286. Walls and Gates

Medium Breadth-First Search Array Matrix

Leetcode Link

Problem Description

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 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.

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.

Intuition

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.

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

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

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

1. **Initialization**: The rooms grid is scanned for all gates (0 values rooms). Their coordinates are added to a deque, a double-ended 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 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, 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.
   - This room's coordinates are then added to the queue for subsequent exploring of its neighbors.
4. **Distance Increment**: Once we've explored all rooms from the current level, we increment d by 1 to reflect the increased distance for the next level of rooms.

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.

By using this approach, the solution efficiently updates the rooms with their minimum distances to the nearest gate in place, avoiding the creation of additional data structures and ensuring optimal space complexity.

Example Walkthrough

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

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

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

Initialization:

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

BFS Implementation:

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

Neighbor Checks:

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

The grid now looks like this:

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

Distance Increment:

- Increment d to 2 for the next level of rooms.

Continue BFS:

- Now, (1, 2) is the only room in the queue. Its neighbors are (1, 1), (1, 3), (0, 2), and (2, 2).
  - (1, 1) is updatable and becomes 2.
  - (1, 3) is out of bounds.
  - (0, 2) is a gate and already has the shortest distance.
  - (2, 2) is updatable and becomes 2.

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

The grid now looks like this:

```
INF  -1  0
 2  INF  1
INF  -1  2
```

Distance Increment:

- Increment d to 3.

Next BFS level:

- 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.

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:

```
INF  -1  0
 3    2    1
INF  -1  2
```

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

```python
from collections import deque

class Solution:
    def wallsAndGates(self, rooms):
        """
        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.

        :type rooms: List[List[int]]
        """

        # Dimensions of the rooms matrix
        num_rows, num_cols = len(rooms), len(rooms[0])

        # Define the representation of an infinite distance (empty room)
        INF = 2**31 - 1

        # Initialize a queue and populate it with the coordinates of all gates
        queue = deque([(row, col) for row in range(num_rows) for col in range(num_cols) if rooms[row][col] == 0])

        # Initialize distance from gates
        distance = 0

        # Perform a breadth-first search (BFS) from the gates
        while queue:
            # Increase the distance with each level of BFS
            distance += 1

            # Process nodes in the current level
            for _ in range(len(queue)):
                i, j = queue.popleft()

                # Explore the four possible directions from the current cell
                for delta_row, delta_col in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    new_row, new_col = i + delta_row, j + delta_col

                    # Check if the new position is within bounds and is an empty room
                    if 0 <= new_row < num_rows and 0 <= new_col < num_cols and rooms[new_row][new_col] == INF:
                        # Update the distance for the room
                        rooms[new_row][new_col] = distance

                        # Add the new position to the queue to process its neighbors
                        queue.append((new_row, new_col))
```

Java Solution

```java
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;

        // create a queue to hold the gate positions
        Deque<int[]> queue = new LinkedList<>();

        // find all gates (represented by 0) and add their positions to the queue
        for (int i = 0; i < numRows; ++i) {
            for (int j = 0; j < numCols; ++j) {
                if (rooms[i][j] == 0) {
                    queue.offer(new int[] {i, j});
                }
            }
        }

        // distance from the gate
        int distance = 0;
        // array to facilitate exploration in 4 directions: up, right, down, left
        int[] directions = {-1, 0, 1, 0, -1};

        // perform BFS starting from each gate
        while (!queue.isEmpty()) {
            ++distance;
            for (int i = queue.size(); i > 0; --i) {
                // get and remove the position from the front of the queue
                int[] position = queue.poll();
                for (int j = 0; j < 4; ++j) { // explore all 4 directions
                    int newRow = position[0] + directions[j];
                    int newCol = position[1] + directions[j + 1];
                    // if the new position is within bounds and is an empty room
                    // (denoted by Integer.MAX_VALUE), then update the distance
                    // and add the new position to the queue for further BFS
                    if (newRow >= 0 && newRow < numRows && newCol >= 0 && newCol < numCols && rooms[newRow][newCol] == Integer.MAX_
                        rooms[newRow][newCol] = distance;
                        queue.offer(new int[] {newRow, newCol});
                    }
                }
            }
        }
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    void wallsAndGates(vector<vector<int>>& rooms) {
        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

        // Find all gates and enqueue their positions
        for (int row = 0; row < rows; ++row) {
            for (int col = 0; col < cols; ++col) {
                if (rooms[row][col] == 0) {
                    toVisit.emplace(row, col);
                }
            }
        }

        // Distance from gate(s)
        int distance = 0;

        // Use BFS to traverse the grid
        while (!toVisit.empty()) {
            ++distance;
            size_t levelSize = toVisit.size();
            for (size_t i = 0; i < levelSize; ++i) {
                auto position = toVisit.front(); // get the front element of the queue
                toVisit.pop(); // remove it from the queue

                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];

                    // Check for valid position and if the room has not been visited or is not a wall
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && rooms[newRow][newCol] == INT_MAX) {
                        rooms[newRow][newCol] = distance; // set distance from nearest gate
                        toVisit.emplace(newRow, newCol); // enqueue the valid position
                    }
                }
            }
        }
    }
};
```

Typescript Solution

```typescript
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

    // Find all gates and enqueue their positions
    for (let row = 0; row < rows; row++) {
        for (let col = 0; col < cols; col++) {
            if (rooms[row][col] === 0) {
                toVisit.push([row, col]);
            }
        }
    }

    // Distance from gate(s)
    let distance = 0;

    // Use BFS to traverse the grid
    while (toVisit.length > 0) {
        const levelSize = toVisit.length;
        distance++;
        for (let i = 0; i < levelSize; i++) {
            const position = toVisit.shift(); // get the front element of the queue
            if (position) {
                const [currentRow, currentCol] = position;

                // Explore all 4 adjacent rooms
                for (let j = 0; j < 4; j++) {
                    const newRow = currentRow + directions[j];
                    const newCol = currentCol + directions[j + 1];

                    // Check for valid position and if the room has not been visited or is not a wall
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && rooms[newRow][newCol] === Number.MAX_SAFE_I
                        rooms[newRow][newCol] = distance; // set the distance from the nearest gate
                        toVisit.push([newRow, newCol]); // enqueue the valid position
                    }
                }
            }
        }
    }
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is O(m * n), where m represents the number of rows and n the number of columns in the 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 write a smaller distance. Hence, the time complexity is linear in the size of the input matrix.

Space Complexity

The space complexity of the given code is also O(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.