LeetCode 286

https://leetcode.com/problems/walls-and-gates/description/

Yifeng Zeng

Description

286. Walls and Gates

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

-1 - A wall or an obstacle.

0 - A gate.

INF - Infinity means an empty room. We use the value 231 - 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
```

Idea Report

Finding a path or calculating a distance in a matrix problem, can be represent as a finding shortest path in an undirected graph problem. Each cell in the matrix is a node, and a up/down/left/right connection to the neighbouring cell is an edge. We are filling each empty room with the distance to its nearest gate, so my primitive idea is to start BFS search at each empty room until we find a gate. But this might take a lot of duplicated search if the number of rooms are much larger than the number of gates. Since this is undirected graph, we can also start search from a gate and update the distance for each room as we go. If a room that we currently searching already has a shorter distance, we can skip this room because there is another path making the distance shorter. One implememtation is that we collecting all the gates and spread out to search from all gates.

Code:

```
class Solution {
    // BFS AC
    public void wallsAndGates(int[][] rooms) {
        Deque<int[]> q = new LinkedList<>();
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) {
                    q.offer(new int[]{i, j});
                }
            }
        }

        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};
        int dist = 1;
        while (!q.isEmpty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                int[] cur = q.poll();
                for (int j = 0; j < dx.length; j++) {
                    int r = cur[0] + dx[j];
                    int c = cur[1] + dy[j];
                    if (isValid(rooms, r, c) && rooms[r][c] > dist) {
                        rooms[r][c] = dist;
                        q.offer(new int[]{r, c});
                    }
                }
            }
            dist++;
```

```
            }
        }

        private boolean isValid(int[][] rooms, int r, int c) {
            if (0 <= r && r < rooms.length && 0 <= c && c < rooms[0].length) {
                return rooms[r][c] != -1;
            }
            return false;
        }
    }
```

Another implementation is just to start BFS search one-by-one, once a room has a smaller distance, that means it can go to another gate with a shorter distance, so we skip that room.

Code:

```
class Solution {
    // BFS AC
    public void wallsAndGates(int[][] rooms) {
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) {
                    bfsHelper(rooms, i, j);
                }
            }
        }
    }

    private void bfsHelper(int[][] rooms, int r0, int c0) {
        int[] dx = {0, 0, 1, -1};
        int[] dy = {1, -1, 0, 0};
        Deque<int[]> q = new LinkedList<>();
        int dist = 0;
        q.offer(new int[]{r0, c0});
        while (!q.isEmpty()) {
            int size = q.size();
            dist++;
            for (int s = 0; s < size; s++) {
                int[] cur = q.poll();
                for (int d = 0; d < dx.length; d++) {
                    int r = cur[0] + dx[d];
                    int c = cur[1] + dy[d];
                    if (isValid(rooms, r, c, dist)) {
                        q.offer(new int[]{r, c});
                        rooms[r][c] = dist;
                    }
                }
            }
```

```
            }
        }


        private boolean isValid(int[][] rooms, int r, int c, int dist) {
            if (0 <= r && r < rooms.length && 0 <= c && c < rooms[0].length) {
                return rooms[r][c] > dist;
            }
            return false;
        }
    }
```

Because we can skip a room with a smaller distance value, then we can also do the search using DFS. The recursion termination condiction is the current room's distance value is smaller the current searched distance.

Code:

```
class Solution {
    public void wallsAndGates(int[][] rooms) {
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) {
                    dfsHelper(rooms, i, j, 0);
                }
            }
        }
    }

    private void dfsHelper(int[][] rooms, int r, int c, int dist) {
        if (r < 0 || r >= rooms.length || c < 0 || c >= rooms[0].length) {
            return;
        }
        if (dist != 0 && rooms[r][c] <= dist) {
            return;
        }

        rooms[r][c] = dist++;

        dfsHelper(rooms, r + 1, c, dist);
        dfsHelper(rooms, r - 1, c, dist);
        dfsHelper(rooms, r, c + 1, dist);
        dfsHelper(rooms, r, c - 1, dist);
    }
}
```

Summary

- Calculating shortest distance in a graph by using BFS.
- Searching in a matrix can be represent as a search in graph.
- Use isValid() value to modulize the code.
- Use int[] dx, dy and a for loop to simplify the coding of 4 or 8 directions.