<https://www.lintcode.com/problem/788/>

**Description**

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, **find the shortest distance for the ball to stop at the destination**. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

1.There is only one ball and one destination in the maze.

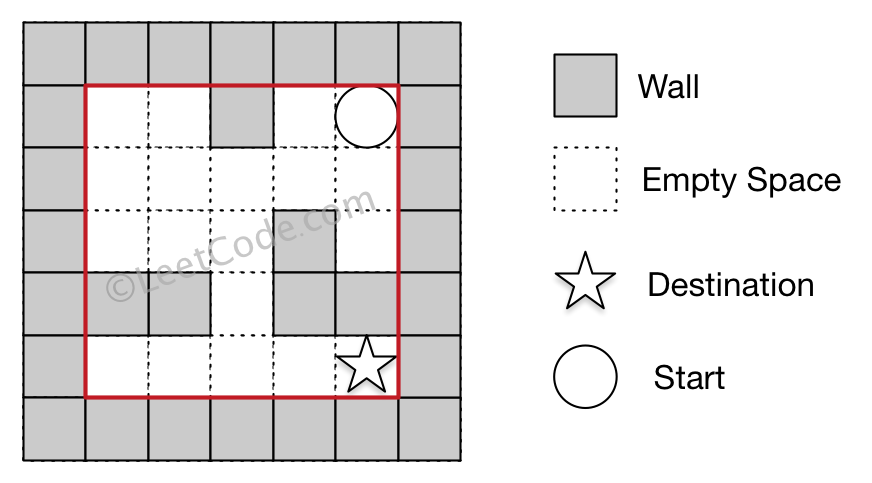
2.Both the ball and the destination exist on an empty space, and they will not be at the same position initially.

3.The given maze does not contain border (like the red rectangle in the example pictures), but you could assume the border of the maze are all walls.

4.The maze contains at least 2 empty spaces, and both the width and height of the maze won't exceed 100.

**Example**

Example 1:



Input:

(rowStart, colStart) = (0,4)

(rowDest, colDest)= (4,4)

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Output:  12

Explanation:

(0,4)->(0,3)->(1,3)->(1,2)->(1,1)->(1,0)->(2,0)->(2,1)->(2,2)->(3,2)->(4,2)->(4,3)->(4,4)

Example 2:

Input:

(rowStart, colStart) = (0,4)

(rowDest, colDest)= (0,0)

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

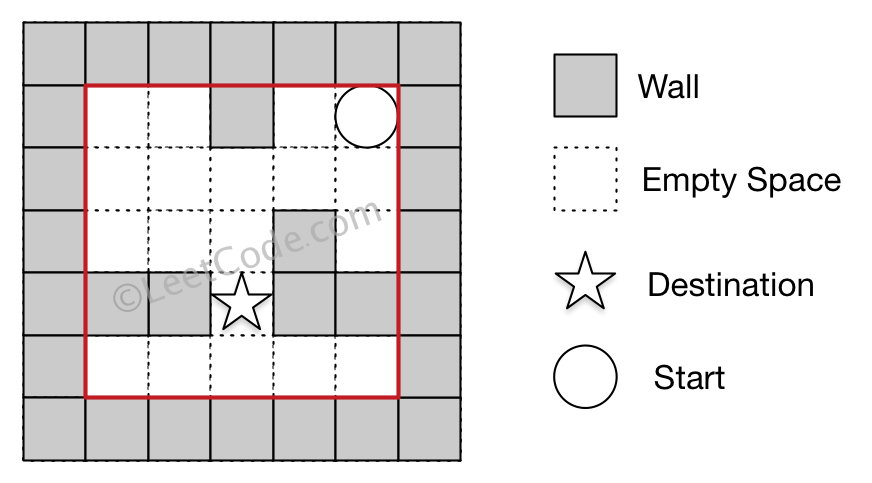
0 0 0 0 0

Output:  6

Explanation:

(0,4)->(0,3)->(1,3)->(1,2)->(1,1)->(1,0)->(0,0)

Example 3:



Input:

(rowStart, colStart) = (0,4)

(rowDest, colDest)= (3,2)

0 0 1 0 0

0 0 0 0 0

0 0 0 1 0

1 1 0 1 1

0 0 0 0 0

Output:  -1

Explanation:

There is no way for the ball to stop at the destination.

**Attempt 1: 2022-11-27**

**Solution 1:  Find minimum distance using BFS [Dijkstra's algorithm] (120min)**

**Use int array distance which exactly same as definition of Dijkstra, check shortest path depends on comparison between original distance and new calculated distance (if u -> v has dist[u] + step < dist[v] then we update)**

import java.util.\*;

public class Solution {

public int shortestDistance(int[][] maze, int[] start, int[] destination) {

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

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

int m = maze.length;

int n = maze[0].length;

// Strictly follow Dijkstra definition, tracking each distance

int[][] distances = new int[m][n];

for(int[] distance : distances) {

Arrays.fill(distance, Integer.MAX\_VALUE);

}

// {x, y, distance}

PriorityQueue<int[]> minPQ = new PriorityQueue<int[]>((a, b) -> a[2] - b[2]);

minPQ.offer(new int[] {start[0], start[1], 0});

while(!minPQ.isEmpty()) {

int[] cur = minPQ.poll();

int cur\_x = cur[0];

int cur\_y = cur[1];

int cur\_distance = cur[2];

if(cur\_x == destination[0] && cur\_y == destination[1]) {

return cur\_distance;

}

distances[cur\_x][cur\_y] = cur\_distance;

for(int k = 0; k < 4; k++) {

// Why we need new variable iter\_x, iter\_y (assign start\_x, start\_y value to them

// initially) instead of directly use start\_x, start\_y ?

// Because in this way we won't change start\_x, start\_y value when we require to

// use it in next for loop iteration (k from 0 to 3) when change to another direction

// to attempt a potential path.

// In more detail, since we are rolling until hit the wall or out the board, if directly

// modify on start\_x, start\_y, the start position will change to the stop position,

// which suppose no change when we attempt on another direction in next for loop

// iteration, actually we do this in an implicit way in normal DFS as directly start

// four directions DFS in for loop, which not modify the start position.

// Refer to below, we don't change the start position value as {i, j}, only pass in

// new value based on different direction choice {i + dx[k], j + dy[k]}

// https://leetcode.com/problems/flood-fill/

// for(int k = 0; k < 4; k++) {

// dfs(i + dx[k], j + dy[k], image, visited, iniColor, newColor);

// }}

int iter\_x = cur\_x;

int iter\_y = cur\_y;

int step = 0;

// Keep rolling till hit the wall

iter\_x += dx[k];

iter\_y += dy[k];

step++;

}

// One step back to the stop position

iter\_x -= dx[k];

iter\_y -= dy[k];

step--;

// Get current rolling result(after it stopped) distance and update to PQ and visited 2D array

int new\_distance = cur\_distance + step;

// Comparison new distance with original value to make sure we need to update based on Dijkstra

// "relaxation" (if u -> v has dist[u] + step < dist[v] then we update), and since we only update

// node's distance strictly when this "relaxation" inequality satisfied, so not every node will be

// added into minPQ, and not every node's distance will be updated, the process won't be infinite.

// Also since every node in minPQ reserves a chance to be updated when above "relaxation" inequality

// satisfied, the traditional BFS "visited" array not satisfy the requirement as it will block

// potential future update on already visited node's distance update, so we ban the "visited" array

// in Dijkstra traversal, the minPQ plus "relaxation" inequality helps avoid duplicate visit.

if(distances[iter\_x][iter\_y] > new\_distance) {

distances[iter\_x][iter\_y] = new\_distance;

minPQ.offer(new int[] {iter\_x, iter\_y, new\_distance});

}

}

}

return -1;

}

public static void main(String[] args) {

Solution so = new Solution();

int[][] maze = new int[][]{{0,0,1,0,0},{0,0,0,0,0},{0,0,0,1,0},{1,1,0,1,1},{0,0,0,0,0}};

int[] start = new int[]{0,4};

int[] destination = new int[]{4,4};

int result = so.shortestDistance(maze, start, destination);

System.out.println(result);

}

}

**Refer to**

<https://www.lintcode.com/problem/788/solution/57320>

**方法：Dijkstra 算法 + 堆优化**

我们可以使用 Dijkstra 算法直接求出从起始位置到终点位置的最短路。这里不会详细介绍 Dijkstra 算法的实现，只会描述如何建立这道题对应的图。

对于迷宫中的任意一个空地 0，即为 x，它可以往四个方向滚动，假设它往上下左右分别可以滚动到位置 p, q, r, s，那么可以从 x 向 p, q, r, s 分别连一条权值为经过空地个数的边，注意这条边是单向边，因为从 x 可以滚动到位置 p 不代表从 p 一定可以滚动到位置 x。

在连完所有的边之后，我们以起始位置为源，使用 Dijkstra 算法计算出其到所有其它位置的最短路长度，也就得到了从起始位置到目的地最少经过的空地个数。

我们可以使用堆（优先队列）优化 Dijkstra 算法，减少其时间复杂度。

**题解代码**

public class Solution {

    public int shortestDistance(int[][] maze, int[] start, int[] dest) {

        int[][] distance = new int[maze.length][maze[0].length];

        for (int[] row: distance)

            Arrays.fill(row, Integer.MAX\_VALUE);

        distance[start[0]][start[1]] = 0;

        dijkstra(maze, start, distance);

        return distance[dest[0]][dest[1]] == Integer.MAX\_VALUE ? -1 : distance[dest[0]][dest[1]];

    }

    public void dijkstra(int[][] maze, int[] start, int[][] distance) {

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

        PriorityQueue < int[] > queue = new PriorityQueue < > ((a, b) -> a[2] - b[2]);

        queue.offer(new int[]{start[0],start[1],0});

        while (!queue.isEmpty()) {

            int[] s = queue.poll();

            if(distance[s[0]][s[1]] < s[2])

                continue;

            for (int[] dir: dirs) {

                int x = s[0] + dir[0];

                int y = s[1] + dir[1];

                int count = 0;

                    x += dir[0];

                    y += dir[1];

                    count++;

                }

                if (distance[s[0]][s[1]] + count < distance[x - dir[0]][y - dir[1]]) {

                    distance[x - dir[0]][y - dir[1]] = distance[s[0]][s[1]] + count;

                    queue.offer(new int[]{x - dir[0], y - dir[1], distance[x - dir[0]][y - dir[1]]});

                }

            }

        }

    }

}

**Refer to**

<http://www.cnblogs.com/grandyang/p/6725380.html>

上面这种解法的 DFS 形式之前是可以通过 OJ 的，但是现在被卡时间过不了了，代码可以参见[评论区第二十三楼](https://www.cnblogs.com/grandyang/p/6725380.html#4511214)。我们还可以使用迪杰斯特拉算法 Dijkstra Algorithm 来做，LeetCode 中能使用到此类高级算法的时候并不多，[Network Delay Time](http://www.cnblogs.com/grandyang/p/8278115.html) 就是一次。该算法是主要是在有向权重图中计算单源最短路径，即单个点到任意点到最短路径。因为这里起点只有一个，所以适用，然后迷宫中的每个格子都可以看作是图中的一个结点，权重可以都看成是1，那么就可以当作是有向权重图来处理。**Dijkstra 算法的核心是松弛操作 Relaxtion，当有对边 (u, v) 是结点u到结点v，如果 dist(v) > dist(u) + w(u, v)，那么 dist(v) 就可以被更新，这是所有这些的算法的核心操作。**Dijkstra 算法是以起点为中心，向外层层扩展，直到扩展到终点为止。那么看到这里，你可能会有个疑问，**到底 Dijkstra 算法和 BFS 算法究竟有啥区别。这是个好问题，二者在求最短路径的时候很相似，但却还是有些区别的。首先 Dijkstra 算法是求单源点的最短路径，图需要有权重，而且权重值不能为负，这道题中两点之间的距离可以看作权重，而且不会为负，满足要求。而 BFS 算法是从某点出发按广度优先原则依次访问连通的结点，图可以无权重。另外一点区别就是，BFS 算法是将未访问的邻居压入队列，然后再将未访问邻居的未访问过的邻居入队列再依次访问，而 Dijkstra 算法是在剩余的未访问过的结点中找出权重最小的并访问，这就是为什么要用一个优先队列（最小堆）来代替普通的 queue，这样就能尽量先更新离起点近的位置的 dp 值，优先队列里同时也存了该点到起点的距离，这个距离不一定是最短距离，可能还能松弛。但是如果其 dp 值已经小于优先队列中保存的距离，那么就不必更新其周围位置的距离了，因为优先队列中保存的这个距离值不是最短的，使用它来更新周围的 dp 值没有意义。这相当于利用了松弛操作来进行剪枝，大大提高了运算效率，之后就是类似于之前的 BFS 的操作了，遍历其周围的四个位置，尝试去更新其 dp 值。**最后还是跟之前一样，如果遍历到了终点，就不要再排入队列了，因为已经得到需要的结果了，参见代码如下：

**总结一点：就算已经访问过的节点也需要保持更新，所以不需要 visited 数组记录节点是否访问过，但是需要 Priority Queue 来对起点到当前节点的总权重排序**

class Solution {

public:

int m = maze.size(), n = maze[0].size();

vector<vector<int>> dists(m, vector<int>(n, INT\_MAX));

vector<vector<int>> dirs{{0,-1},{-1,0},{0,1},{1,0}};

auto cmp = [](vector<int>& a, vector<int>& b) {

return a[2] > b[2];

};

priority\_queue<vector<int>, vector<vector<int>>, decltype(cmp) > pq(cmp);

pq.push({start[0], start[1], 0});

dists[start[0]][start[1]] = 0;

while (!pq.empty()) {

auto t = pq.top(); pq.pop();for (auto dir : dirs) {

int x = t[0], y = t[1], dist = dists[t[0]][t[1]];

x += dir[0];

y += dir[1];

++dist;

}

x -= dir[0];

y -= dir[1];

--dist;

if (dists[x][y] > dist) {

dists[x][y] = dist;

if (x != destination[0] || y != destination[1]) pq.push({x, y, dist});

}

}

}

int res = dists[destination[0]][destination[1]];

return (res == INT\_MAX) ? -1 : res;

}

};

**Refer to**

[L490.Lint787.The Maze (Ref.L505)](note://AD1A9342CF124ABF91D761877F82C2A5)

[L743.Network Delay Time](note://17E4C21F015F4605AAD4D6878F0D0A67)

[Dijkstra Shortest Path Algorithm - A Detailed and Visual Introduction](note://80857119213E49EC840091BB3F7E4356)