<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)

**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)**

public class Solution {

/\*\*

\* @param maze: the maze

\* @param start: the start

\* @param destination: the destination

\* @return: the shortest distance for the ball to stop at the destination

\*/

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++) {

// Don't understand why assign start\_i, start\_j to new variable x, y?

// Recording start position before start rolling in the maze, for next time start another

// direction's DFS, since we are rolling until hit the wall, so start position if not

// record will change to the stop position, which change the start condition of DFS in this

// time recursion, 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

// 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

while(iter\_x >= 0 && iter\_x < m && iter\_y >= 0 && iter\_y < n && maze[iter\_x][iter\_y] != 1) {

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

// definition (if u -> v has dist[u] + step < dist[v] then we update)

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;

}

}

**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;

while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 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]]});

}

}

}

}

}