<https://www.lintcode.com/problem/787/>

**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, determine whether the ball could stop at the destination.

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.

**Example 1:**

Input 1: a maze represented by a 2D array

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

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (4, 4)

**Output:** true

**Explanation:** One possible way is : left -> down -> left -> down -> right -> down -> right.

**Example 2:**

Input 1: a maze represented by a 2D array

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

Input 2: start coordinate (rowStart, colStart) = (0, 4)

Input 3: destination coordinate (rowDest, colDest) = (3, 2)

**Output:** false

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

**Note:**

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

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

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.

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

**Attempt 1: 2022-11-26**

**Wrong solution**

public class Solution {

    /\*\*

    \* @param maze: the maze

    \* @param start: the start

    \* @param destination: the destination

    \* @return: whether the ball could stop at the destination

    \*/

    int[] dx = new int[]{0,0,1,-1};

    int[] dy = new int[]{-1,1,0,0};

    public boolean hasPath(int[][] maze, int[] start, int[] destination) {

        if(maze == null || maze.length == 0) {

            return false;

        }

        boolean[][] visited = new boolean[maze.length][maze[0].length];

        return helper(maze, visited, start[0], start[1], destination[0], destination[1]);

    }

    private boolean helper(int[][] maze, boolean[][] visited, int start\_x, int start\_y, int dest\_x, int dest\_y) {

        if(start\_x == dest\_x && start\_y == dest\_y) {

            return true;

        }

        if(start\_x < 0 || start\_x >= maze.length || start\_y < 0 || start\_y >= maze[0].length || visited[start\_x][start\_y]) {

            return false;

        }

        visited[start\_x][start\_y] = true;

        for(int k = 0; k < 4; k++) {

            // rolling until out or hit the wall

                start\_x += dx[k];

                start\_y += dy[k];

            }

            // one step back to the stop position

            start\_x -= dx[k];

            start\_y -= dy[k];

            // start a new dfs from the stop position

            if(helper(maze, visited, start\_x, start\_y, dest\_x, dest\_y)) {

                return true;

            }

        }

        return false;

    }

}

**Solution 1:  DFS (30min, use 'iter' to start current dfs then restore 'iter' back to start new dfs)**

**Correct Solution with reassign start\_i, start\_j to x, y**

**Caution: The difference here is very important, after each time rolling until hit the wall, we must return back to original start position to prepare another direction DFS, same as normal DFS one direction one step, but since we have to rolling until hit the wall in this problem, recording the original position by int x = start\_i, int y = start\_j is very critical**

import java.util.\*;

public class Solution {

/\*\*

\* @param maze: the maze

\* @param start: the start

\* @param destination: the destination

\* @return: whether the ball could stop at the destination

\*/

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

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

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

if(maze == null || maze.length == 0) {

return false;

}

boolean[][] visited = new boolean[maze.length][maze[0].length];

return helper(maze, visited, start[0], start[1], destination[0], destination[1]);

}

private boolean helper(int[][] maze, boolean[][] visited, int start\_x, int start\_y, int dest\_x, int dest\_y) {

if (start\_x == dest\_x && start\_y == dest\_y) {

return true;

}

if (start\_x < 0 || start\_x >= maze.length || start\_y < 0 || start\_y >= maze[0].length || visited[start\_x][start\_y]) {

return false;

}

visited[start\_x][start\_y] = true;

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 = start\_x;

int iter\_y = start\_y;

// Rolling until out or hit the wall

iter\_x += dx[k];

iter\_y += dy[k];

}

// One step back to the stop position

iter\_x -= dx[k];

iter\_y -= dy[k];

// start a new dfs from the stopped position

if (helper(maze, visited, iter\_x, iter\_y, dest\_x, dest\_y)) {

return true;

}

}

return false;

}

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};

boolean result = so.hasPath(maze, start, destination);

System.out.println(result);

}

}

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/graph-search/490.the-maze>

class Solution {

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

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

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

return dfs(maze, start, destination, visited);

}

private boolean dfs(int[][] maze, int[] start, int[] destination, boolean[][] visited) {

if (visited[start[0]][start[1]]) return false;

if (start[0] == destination[0] && start[1] == destination[1]) {

return true;

}

visited[start[0]][start[1]] = true;

int R = maze.length;

int C = maze[0].length;

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

int r = start[0] + dr[i];

int c = start[1] + dc[i];

r = r + dr[i];

c = c + dc[i];

}

if (!visited[r - dr[i]][c - dc[i]]) {

if (dfs(maze, new int[]{r - dr[i], c - dc[i]}, destination, visited)) {

return true;

}

}

}

return false;

}

}

**Solution 2: BFS (30min)**

import java.util.\*;

public class Solution {

/\*\*

\* @param maze: the maze

\* @param start: the start

\* @param destination: the destination

\* @return: whether the ball could stop at the destination

\*/

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

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

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

Queue <int[]> queue = new LinkedList();

queue.add(start);

visited[start[0]][start[1]] = true;

while (!queue.isEmpty()) {

int[] s = queue.remove();

if (s[0] == destination[0] && s[1] == destination[1]) {

return true;

}

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

int x = s[0];

int y = s[1];

// Rolling until out or hit the wall

x += dx[k];

y += dy[k];

}

x -= dx[k];

y -= dy[k];

if (!visited[x][y]) {

queue.add(new int[] {x, y});

visited[x][y] = true;

}

}

}

return false;

}

public static void main(String[] args) {

//Solution so = new Solution();

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};

boolean result = so.hasPath(maze, start, destination);

System.out.println(result);

}

}

**Refer to**

<https://wentao-shao.gitbook.io/leetcode/graph-search/490.the-maze>

public class Solution {

public boolean hasPath(int[][] maze, int[] start, int[] destination) {

boolean[][] visited = new boolean[maze.length][maze[0].length];

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

Queue <int[]> queue = new LinkedList();

queue.add(start);

visited[start[0]][start[1]] = true;

while (!queue.isEmpty()) {

int[] s = queue.remove();

if (s[0] == destination[0] && s[1] == destination[1])

return true;

// The furthest point in four directions

for (int[] dir: dirs) {

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

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

// move Furthest straight-line distance

x += dir[0];

y += dir[1];

}

// Furthest (x,y)

if (!visited[x - dir[0]][y - dir[1]]) {

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

visited[x - dir[0]][y - dir[1]] = true;

}

}

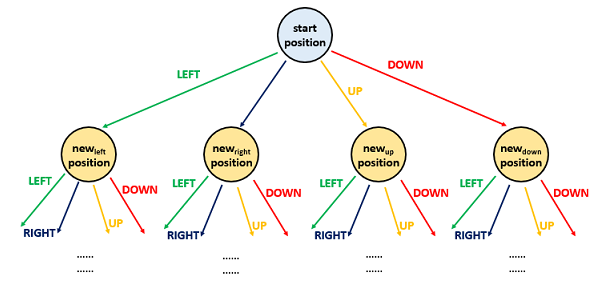
}

return false;

}

}

**方法：广度优先搜索**



我们可以用搜索树的形式来展开搜索空间。如下图所示，根节点代表起始位置，每个节点有 4 个孩子，表示 4 种不同的路线：左、右、上、下。经过某条路线到达一个新的节点，就表示在迷宫中选择某个方向滚动直到停止。我们可以使用广度优先搜索对整颗搜索树进行遍历。注意在一般的广度优先搜索中，我们不会经过同一个节点超过一次，但在这道题目中，只要从起始位置到当前节点的步数 count 小于之前记录的最小步数 distance[i, j]，我们就会把 (i, j) 再次加入队列中。

**Refer to**

[L505.Lint788.The Maze II (Ref.L490)](note://6441348532EF41BEB2DDC34801D7AA20)