Previous (/articles/design-in-memory-file-system/) Next (/articles/the-maze-ii/)

490. The Maze (/problems/the-maze/)

Average Rating: 4.60 (10 votes)

May 22, 2017 | 8.2K views

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.

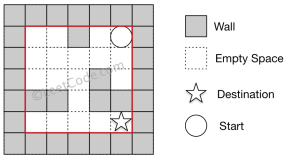
Example 1

```
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.
```



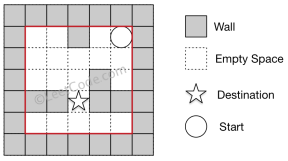
Example 2

```
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.
```



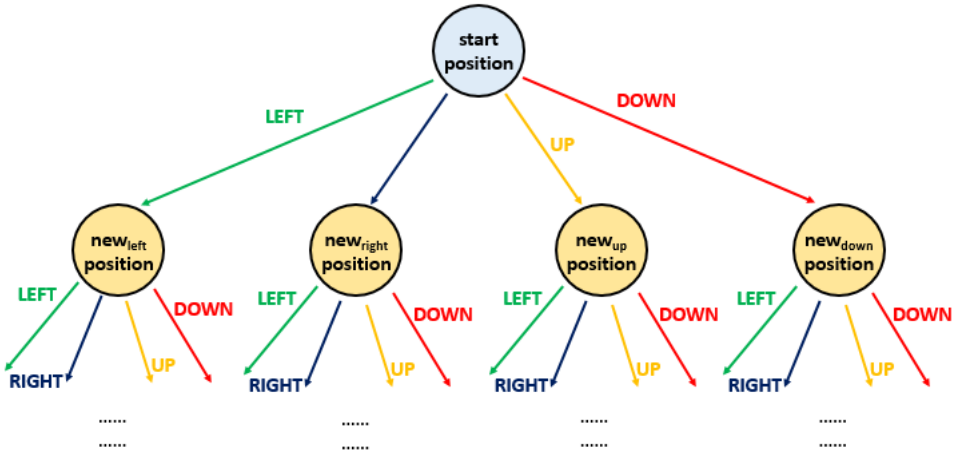
Note:

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.

Solution

Approach #1 Depth First Search [Time Limit Exceeded]

We can view the given search space in the form of a tree. The root node of the tree represents the starting position. Four different routes are possible from each position i.e. left, right, up or down. These four options can be represented by 4 branches of each node in the given tree. Thus, the new node reached from the root traversing over the branch represents the new position occupied by the ball after choosing the corresponding direction of travel.

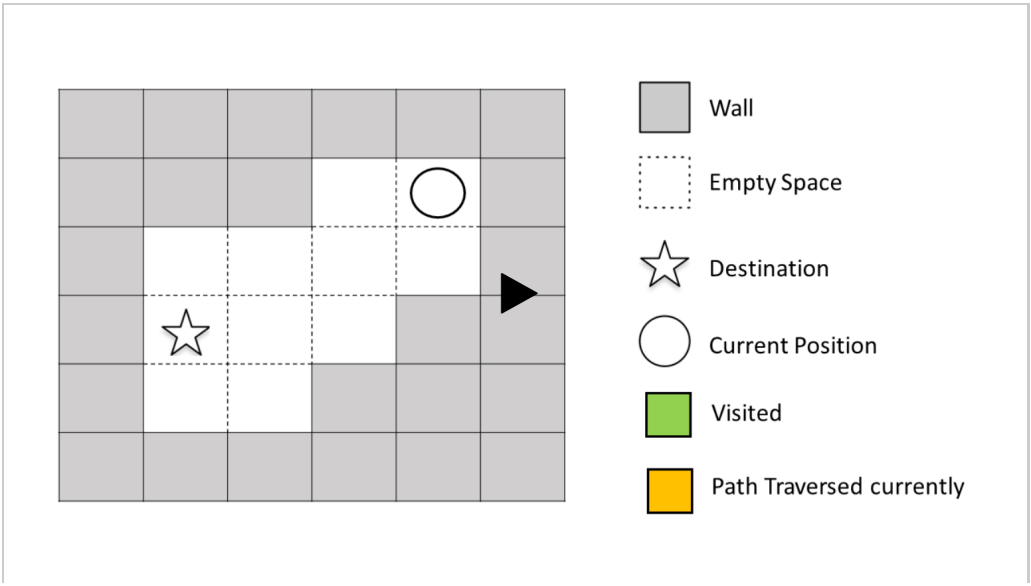


In order to do this traversal, one of the simplest schemes is to undergo depth first search. In this case, we choose one path at a time and try to go as deep as possible into the levels of the tree before going for the next path. In order to implement this, we make use of a recursive function `dfs(maze, start, desination, visited)`. This function takes the given $maze$ array, the $start$ position and the $destination$ position as its arguments along with a $visited$ array. $visited$ array is a 2-D boolean array of the same size as that of $maze$. A True value at $visited[i][j]$ represents that the current position has already been reached earlier during the path traversal. We make use of this array so as to keep track of the same paths being repeated over and over. We mark a True at the current position in the $visited$ array once we reach that particular positon in the $maze$.

From every $start$ position, we can move continuously in either left, right, upward or downward direction till we reach the boundary or a wall. Thus, from the $start$ position, we determine all the end points which can be reached by choosing the four directions. For each of the cases, the new endpoint will now act as the new start point for the traversals. The destination, obviously remains unchanged. Thus, now we call the same function four times for the four directions, each time with a new start point obtained previously.

If any of the function call returns a True value, it means we can reach the desination.

The following animation depicts the process:



Java Copy

```java
public class Solution {
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        boolean[][] visited = new boolean[maze.length][maze[0].length];
        return dfs(maze, start, destination, visited);
    }
    public 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 = start[1] + 1, l = start[1] - 1, u = start[0] - 1, d = start[0] + 1;
        while (r < maze[0].length && maze[start[0]][r] == 0) // right
            r++;
        if (dfs(maze, new int[] {start[0], r - 1}, destination, visited))
            return true;
        while (l >= 0 && maze[start[0]][l] == 0) //left
            l--;
        if (dfs(maze, new int[] {start[0], l + 1}, destination, visited))
            return true;
        while (u >= 0 && maze[u][start[1]] == 0) //up
            u--;
        if (dfs(maze, new int[] {u + 1, start[1]}, destination, visited))
            return true;
        while (d < maze.length && maze[d][start[1]] == 0) //down
            d++;
        if (dfs(maze, new int[] {d - 1, start[1]}, destination, visited))
            return true;
```

Complexity Analysis

- Time complexity : $O(mn)$. Complete traversal of maze will be done in the worst case. Here, m and n refers to the number of rows and coloumns of the maze.
- Space complexity : $O(mn)$. $visited$ array of size $m * n$ is used.

Approach #2 Breadth First Search [Accepted]

Algorithm

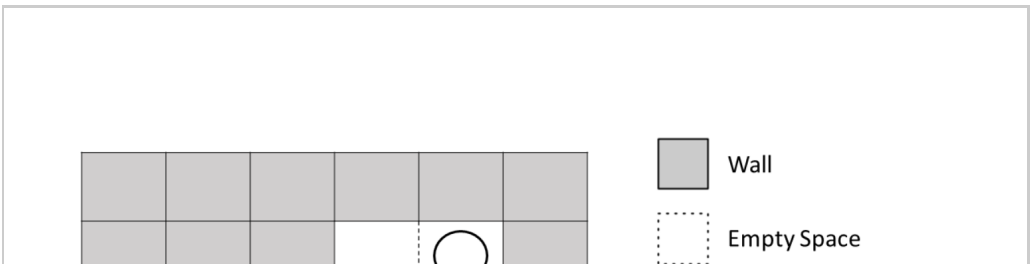
The same search space tree can also be explored in a Depth First Search manner. In this case, we try to explore the search space on a level by level basis. i.e. We try to move in all the directions at every step. When all the directions have been explored and we still don't reach the destination, then only we proceed to the new set of traversals from the new positions obtained.

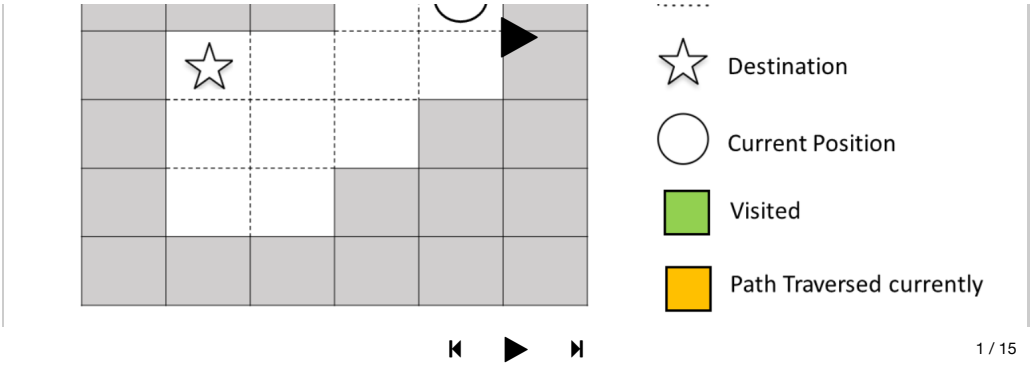
In order to implement this, we make use of a $queue$. We start with the ball at the $start$ position. For every current position, we add all the new positions possible by traversing in all the four directions(till reaching the wall or boundary) into the $queue$ to act as the new start positions and mark these positions as True in the $visited$ array. When all the directions have been covered up, we remove a position value, s, from the front of the $queue$ and again continue the same process with s acting as the new $start$ position.

Further, in order to choose the direction of travel, we make use of a dir array, which contains 4 entries. Each entry represents a one-dimensional direction of travel. To travel in a particular direction, we keep on adding the particular entry of the $dirs$ array till we hit a wall or a boundary. For a particular start position, we do this process of dir addition for all all the four directions possible.

If we hit the destination position at any moment, we return a True directly indicating that the $destination$ position can be reached starting from the $start$ position.

The following animation depicts the process:





Java Copy

```java
public class Solution {
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        boolean[][] visited = new boolean[maze.length][maze[0].length];
        int[][] dirs={{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;
            for (int[] dir: dirs) {
                int x = s[0] + dir[0];
                int y = s[1] + dir[1];
                while (x >= 0 && y >= 0 && x < maze.length && y < maze[0].length && maze[x][y] == 0)
{
                    x += dir[0];
                    y += dir[1];
                }
                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;
    }
}
```

Complexity Analysis

- Time complexity : $O(mn)$. Complete traversal of maze will be done in the worst case. Here, m and n refers to the number of rows and coloumns of the maze.
- Space complexity : $O(mn)$. $visited$ array of size $m * n$ is used and $queue$ size can grow upto $m * n$ in worst case.

Analysis written by: @vinod23 (https://leetcode.com/vinod23)

Rate this article: (/ratings/107/143/?return=/articles/the-maze/) (/ratings/107/143/?return=/articles/the-maze/) (/rat

Previous (/articles/design-in-memory-file-system/) Next (/articles/the-maze-ii/)

Join the conversation

Signed in as **tan7**.

Post a Reply

H (https://discuss.leetcode.com/user/hendisantika)

hendisantika commented last week

Hi @devashish2008 (https://discuss.leetcode.com/uid/164752), could you give me how many possibilities that has in the given maze?
Thanks

H (https://discuss.leetcode.com/user/han_v)

Han_V commented 3 weeks ago

I've implemented BFS as my first idea yet still I don't understand why tests make DFS fail due to TL. BFS in this case is more efficient yet they have same worst case time complexity so to be honest I don't necessarily agree with this decision.

M (https://discuss.leetcode.com/user/mr-crazy)

Mr.Crazy commented 7 months ago

@vinod23 (https://discuss.leetcode.com/uid/1362)
My dfs solution just got accepted.
Share my code and comments:

```
// dfs solution
class Solution {
    private int[] dr;
    private int[] dc;
    private int[][] MAZE;
    private int R;
    private int C;
    private int[] dest;
    private boolean res;
    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        dr = new int[]{-1, 1, 0, 0}; // u d l r
        dc = new int[]{0, 0, -1, 1}; // u d l r
        MAZE = maze;
        R = maze.length;
        C = maze[0].length;
        dest = destination;
        res = false;

        // doesn't care the initial direction parameter
        dfs(-1, start, new boolean[R][C]);
        return res;
    }

    private void dfs(int dir, int[] start, boolean[][] visited) {
        int r = start[0], c = start[1];
        if (Arrays.equals(start, dest)) {
            res = true;
            return;
        }
        if (visited[r][c]) return;
        visited[r][c] = true;
        // up down left right
        for (int i = 0; i < 4; ++i) {
            if (i == dir) continue; // skip the direction that will hit the wall again
            int x = r, y = c;
            while (isValid(new int[]{x + dr[i], y + dc[i]})) {
                x += dr[i];
                y += dc[i];
            }
            dfs(i, new int[]{x, y}, visited);
        }
    }

    // return true if the coord is valid and maze[coord] == 0;
    private boolean isValid(int[] coord) {
        int r = coord[0];
        int c = coord[1];
        if (r < 0 || r >= R || c < 0 || c >= C) return false;
        if (MAZE[r][c] == 1) return false;
        return true;
    }
}
```


(https://discuss.leetcode.com/user/yuely_y)

yuely_y commented 8 months ago

The time complexity analysis is should be O(mn(m+n)).
Determining the position after a left/right movement is O(m), and O(n) after a up/down movement.


(https://discuss.leetcode.com/user/vinod23)

vinod23 commented 11 months ago

@devashish2008 (https://discuss.leetcode.com/uid/164752) DFS and your backtracking approach are very similar, both are using recursion. BFS and backtracking have same complexity but I think BFS would be more efficient as it is iterative while other one is recursive.


(https://discuss.leetcode.com/user/vinod23)

vinod23 commented 11 months ago

@guoleisun (https://discuss.leetcode.com/uid/226461) We aren't walking step by step. But, after every step we're trying to check if we've reached a wall. Only that end position is considered at last, which is just adjacent to the wall.


(https://discuss.leetcode.com/user/guoleisun)

guoleisun commented 11 months ago

Is this correct? It seems you are walking step by step, which is not true. You can walk more than one step along one direction as long as didn't reach the wall.

D

devashish2008 commented 11 months ago

Have tried doing this problem by backtracking method. Java solution below :
(https://discuss.leetcode.com/user/devashish2008)

```
class Coordibate
{
    int row,col;
    public  Coordibate(int row,int col)
    {
        this.row = row;
        this.col = col;
    }

}
public class Maze {

    public static boolean traverse(int maze[][],Coordibate src,Coordibate dest,boolean visite
d[][])
    {
        if(src.row == maze.length || src.col == maze.length || src.row == -1 || src.col == -1
) return false;
        if(maze[src.row][src.col] == 1 ) return false;
        if(visited[src.row][src.col] == true) return false;
        if(src.row == dest.row && src.col == dest.col)
        {
            System.out.print("found");
            return true;
        }

        visited[src.row][src.col] = true;

        return traverse(maze,new Coordibate(src.row-1,src.col),dest,visited) ||
                traverse(maze,new Coordibate(src.row+1,src.col),dest,visited) ||
                traverse(maze,new Coordibate(src.row,src.col-1),dest,visited) ||
                traverse(maze,new Coordibate(src.row,src.col+1),dest,visited);
    }

    public static void main(String args[])
    {
        int maze [][] = {
                {0,0,1,0,0},
                {0,0,0,0,0},
                {0,1,0,1,0},
                {1,1,0,1,1},
                {0,0,0,0,0}
        };

        boolean visited [][] = new boolean[maze.length][maze.length];
       for(int i=0;i<maze.length;i++)
       {
           for(int j=0;j<maze.length;j++)
               visited[i][j] = false;
       }

        System.out.println(Maze.traverse(maze,new Coordibate(0,4),new Coordibate(4,4),visite
d));

    }
```

Complexity of this solution should be O(mn).
So, which approach is efficient - BFS or backtracking ? Or, both the approaches look similar ? Suggestions/Thoughts...

View original thread (https://discuss.leetcode.com/topic/90182)