505. The Maze II

May 23, 2017 | 48.2K views

Average Rating: 4.74 (39 votes)

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.

Example 1:

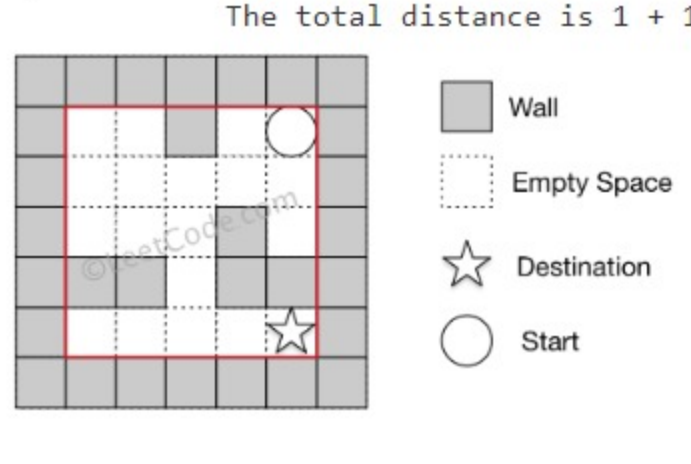
```
Input 1: a maze represented by a 2D array

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: 12

Explanation: One shortest way is : left -> down -> left -> down -> right -> down -> ri
             The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.
```



Example 2:

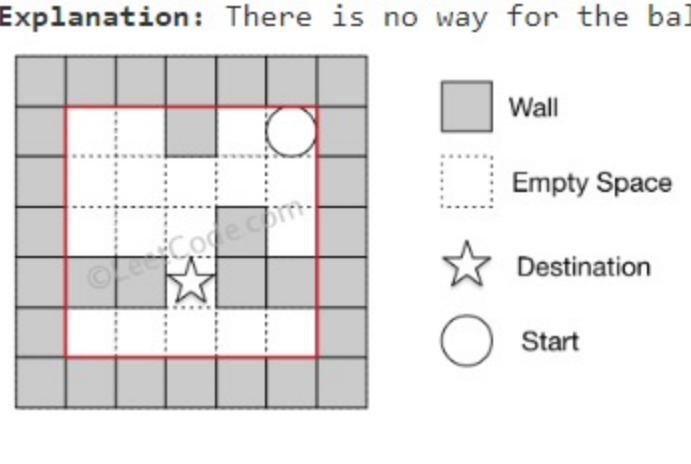
```
Input 1: a maze represented by a 2D array

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: -1

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



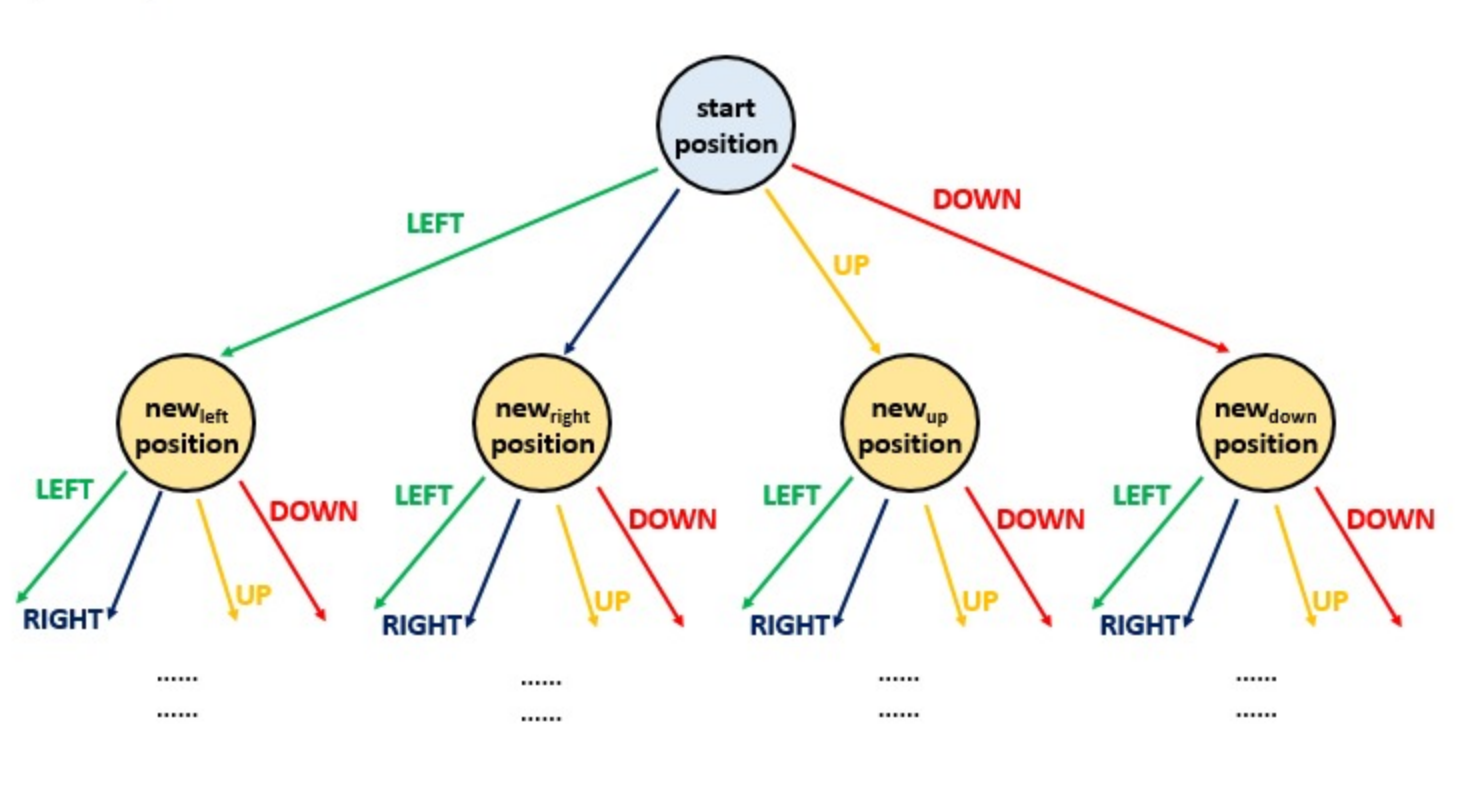
Note:

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 [Accepted]

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. We make use of a recursive function `dfs` for this. From every current position, we try to go as deep as possible into the levels of a tree taking a particular branch traversal direction as possible. When one of the deepest levels is exhausted, we continue the process by reaching the next deepest levels of the tree. In order to travel in the various directions from the current position, we make use of a $dirs$ array. $dirs$ is an array with 4 elements, where each of the elements represents a single step of a one-dimensional movement. For travelling in a particular direction, we keep on adding the appropriate $dirs$ element in the current position till the ball hits a boundary or a wall.

We start with the given $start$ position, and try to explore these directions represented by the $dirs$ array one by one. For every element dir of the $dirs$ chosen for the current travelling direction, we determine how far can the ball travel in this direction prior to hitting a wall or a boundary. We keep a track of the number of steps using $count$ variable.

Apart from this, we also make use of a 2-D $distance$ array. $distance[i][j]$ represents the minimum number of steps required to reach the positon (i, j) starting from the $start$ position. This array is initialized with largest integer values in the beginning.

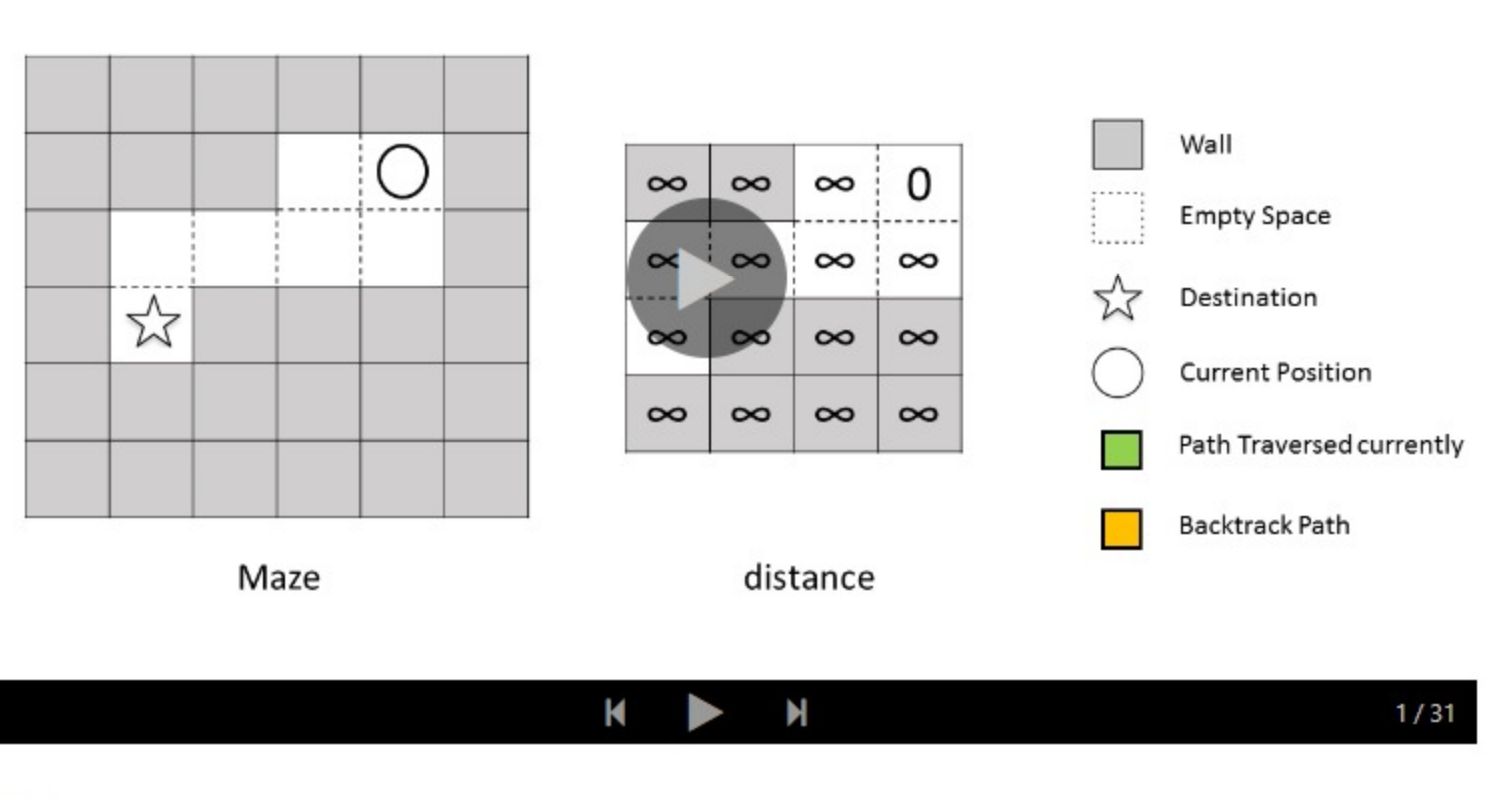
When we reach any position next to a boundary or a wall during the traversal in a particular direction, as discussed earlier, we keep a track of the number of steps taken in the last direction in $count$ variable. Suppose, we reach the position (i, j) starting from the last position (k, l). Now, for this position, we need to determine the minimum number of steps taken to reach this position starting from the $start$ position. For this, we check if the current path takes lesser steps to reach (i, j) than any other previous path taken to reach the same position i.e. we check if $distance[k][l] + count$ is lesser than $distance[i][j]$. If not, we continue the process of traversal from the position (k, l) in the next direction.

If $distance[k][l] + count$ is lesser than $distance[i][j]$, we can reach the position (i, j) from the current route in lesser number of steps. Thus, we need to update the value of $distance[i][j]$ as $distance[k][l] + count$. Further, now we need to try to reach the destination, $dest$, from the end position (i, j), since this could lead to a shorter path to $dest$. Thus, we again call the same function `dfs` but with the position (i, j) acting as the current position.

After this, we try to explore the routes possible by choosing all the other directions of travel from the current position (k, l) as well.

At the end, the entry in distance array corresponding to the destination, $dest$'s coordinates gives the required minimum distance to reach the destination. If the destination can't be reached, the corresponding entry will contain \(\text{Integer.MAX_VALUE}\).

The following animation depicts the process.



Java

```java
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;
        dfs(maze, start, distance);
        return distance[dest[0]][dest[1]] == Integer.MAX_VALUE ? -1 : distance[dest[0]][dest[1]];
    }

    public void dfs(int[][] maze, int[] start, int[][] distance) {
        int[][] dirs={{0,1}, {0,-1}, {-1,0}, {1,0}};
        for (int[] dir: dirs) {
            int x = start[0] + dir[0];
            int y = start[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[start[0]][start[1]] + count < distance[x - dir[0]][y - dir[1]]) {
                distance[x - dir[0]][y - dir[1]] = distance[start[0]][start[1]] + count;
                dfs(maze, new int[]{x - dir[0],y - dir[1]}, distance);
            }
        }
    }
}
```

Complexity Analysis

- Time complexity : $O(m*n*\max(m,n))$. Complete traversal of maze will be done in the worst case. Here, m and n refers to the number of rows and columns of the maze. Further, for every current node chosen, we can travel upto a maximum depth of $\max(m,n)$ in any direction.
- Space complexity : $O(mn)$. $distance$ array of size $m*n$ is used.

Approach #2 Using Breadth First Search [Accepted]

Algorithm

Instead of making use of Depth First Search for exploring the search space, we can make use of Breadth First Search as well. In this, instead of exploring the search space on a depth basis, we traverse the search space(tree) on a level by level basis i.e. we explore all the new positions that can be reached starting from the current position first, before moving onto the next positions that can be reached from these new positions.

In order to make a traversal in any direction, we again make use of $dirs$ array as in the DFS approach. Again, whenever we make a traversal in any direction, we keep a track of the number of steps taken while moving in this direction in $count$ variable as done in the last approach. We also make use of $distance$ array initialized with very large values in the beginning. $distance[i][j]$ again represents the minimum number of steps required to reach the position (i, j) from the $start$ position.

This approach differs from the last approach only in the way the search space is explored. In order to reach the new positions in a Breadth First Search order, we make use of a $queue$, which contains the new positions to be explored in the future. We start from the current position (k, l), try to traverse in a particular direction, obtain the corresponding $count$ for that direction, reaching an end position of (i, j) just near the boundary or a wall. If the position (i, j) can be reached in a lesser number of steps from the current route than any other previous route checked, indicated by $distance[k][l] + count < distance[i][j]$, we need to update $distance[i][j]$ as $distance[k][l] + count$.

After this, we add the new position obtained, (i, j) to the back of a $queue$, so that the various paths possible from this new position will be explored later on when all the directions possible from the current position (k, l) have been explored. After exploring all the directions from the current position, we remove an element from the front of the $queue$ and continue checking the routes possible through all the directions now taking the new position(obtained from the $queue$) as the current position.

Again, the entry in distance array corresponding to the destination, $dest$'s coordinates gives the required minimum distance to reach the destination. If the destination can't be reached, the corresponding entry will contain \(\text{Integer.MAX_VALUE}\).

Java

```java
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;
         int[][] dirs={{0, 1} ,{0, -1}, {-1, 0}, {1, 0}};
        Queue < int[] > queue = new LinkedList < > ();
        queue.add(start);
        while (!queue.isEmpty()) {
            int[] s = queue.remove();
            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.add(new int[] {x - dir[0], y - dir[1]});
                }
            }
        }
        return distance[dest[0]][dest[1]] == Integer.MAX_VALUE ? -1 : distance[dest[0]][dest[1]];
    }
```

Complexity Analysis

- Time complexity : $O(m*n*max(m,n))$. Time complexity : $O(m*n*\max(m,n))$. Complete traversal of maze will be done in the worst case. Here, m and n refers to the number of rows and columns of the maze. Further, for every current node chosen, we can travel upto a maximum depth of $\max(m,n)$ in any direction.
- Space complexity : $O(mn)$. $queue$ size can grow upto $m*n$ in the worst case.

Approach #3 Using Dijkstra Algorithm [Accepted]

Algorithm

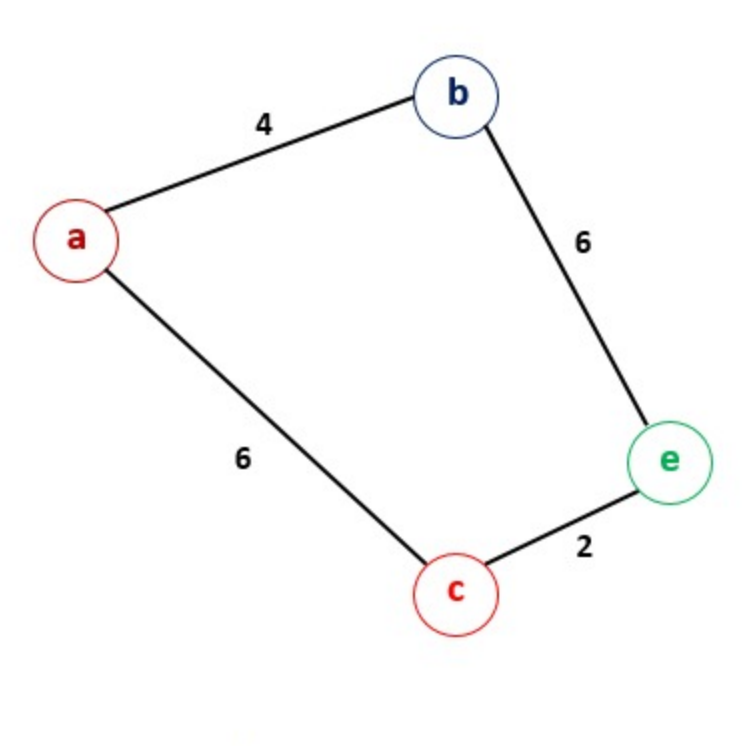
Before we look into this approach, we take a quick overview of Dijkstra's Algorithm.

Dijkstra's Algorithm is a very famous graph algorithm, which is used to find the shortest path from any $start$ node to any $destination$ node in the given weighted graph(the edges of the graph represent the distance between the nodes).

The algorithm consists of the following steps:

1. Assign a tentative distance value to every node: set it to zero for our $start$ node and to infinity for all other nodes.
2. Set the $start$ node as $current$ node. Mark it as visited.
3. For the $current$ node, consider all of its neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one to all the neighbors. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the $current$ node as visited. A visited node will never be checked again.
5. If the $destination$ node has been marked visited or if the smallest tentative distance among all the nodes left is infinity(indicating that the $destination$ can't be reached), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new $current$ node, and go back to step 3.

The working of this algorithm can be understood by taking two simple examples. Consider the first set of nodes as shown below.

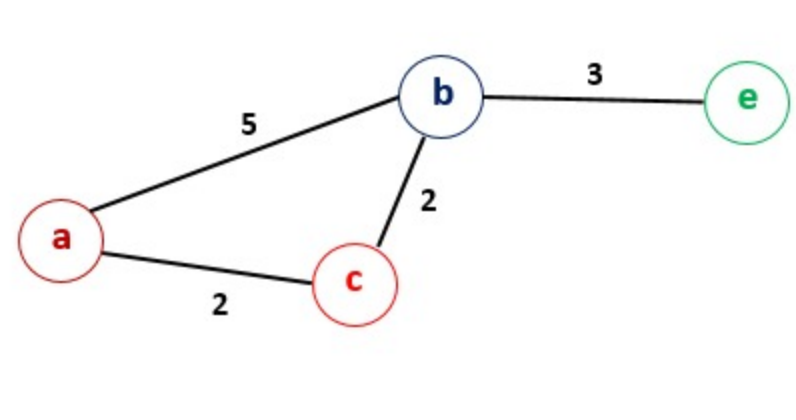


Suppose that the node b is at a shorter distance from the $start$ node a as compared to c, but the distance from a to the $destination$ node, e, is shorter through the node c itself. In this case, we need to check if the Dijkstra's algorithm works correctly, since the node b is considered first while selecting the nodes being at a shorter distance from a. Let's look into this.

1. Firstly, we choose a as the $start$ node, mark it as visited and update the $distance_b$ and $distance_c$ values. Here, $distance_i$ represents the distance of node i from the $start$ node.
2. Since $distance_b < distance_c$, b is chosen as the next node for calculating the distances. We mark b as visited. Now, we update the $distance_e$ value as $distance_b + weight_{be}$.
3. Now, c is obviously the next node to be chosen as per the conditions of the assumptions taken above. (For path to e through c to be shorter than path to e through c, $distance_c < distance_b + weight_{be}$. From c, we determine the distance to node e. Since $distance_c + weight_{ce}$ is shorter than the previous value of $distance_e$, we update $distance_e$ with the correct shorter value.
4. We choose e as the current node. No other distances need to be updated. Thus, we mark e as visited. $distance_e$ now gives the required shortest distance.

The above example proves that even if a locally closer node is chosen as the current node first, the ultimate shortest distance to any node is calculated correctly.

Let's take another example to demonstrate that the visited node needs not be chosen again as the current node.



Suppose a is the $start$ node and e is the $destination$ node. Now, suppose we visit b first and mark it as visited, but later on we find that another path exists through c to b, which makes the $distance_b$ shorter than the previous value. But, because of this, we need to consider b as the current node again, since it would affect the value of $distance_e$. But, if we observe closely, such a situation would never occur, because for $weight_{ac} + weight_{cb}$ to be lesser than $weight_{ab}$, $weight_{ac} < weight_{ab}$ in the first place. Thus, b would never be marked $visited$ before c, which contradicts the first assumption. This proves that the $visited$ node needs not be chosen as the current node again.

The given problem is also a shortest distance finding problem with a slightly different set of rules. Thus, we can make use of Dijkstra's Algorithm to determine the minimum number of steps to reach the destination.

The methodology remains the same as the DFS or BFS Approach discussed above, except the order in which the current positions are chosen. We again make use of a $distance$ array to keep a track of the minimum number of steps needed to reach every position from the $start$ position. At every step, we choose a position which hasn't been marked as visited and which is at the shortest distance from the $start$ position to be the current position. We mark this position as visited so that we don't consider this position as the current position again.

From the current position, we determine the number of steps required to reach all the positions possible travelling from the current position(in all the four directions possible till hitting a wall/boundary). If it is possible to reach any position through the current route with a lesser number of steps than the earlier routes considered, we update the corresponding $distance$ entry. We continue the same process for the other directions as well for the current position.

In order to determine the current node, we make use of `minDistance` function, in which we traverse over the whole $distance$ array and find out an unvisited node at the shortest distance from the $start$ node.

At the end, the entry in $distance$ array corresponding to the $destination$ position gives the required minimum number of steps. If the destination can't be reached, the corresponding entry will contain \(\text{Integer.MAX_VALUE}\).

Java

```java
public class Solution {
    public int shortestDistance(int[][] maze, int[] start, int[] dest) {
        int[][] distance = new int[maze.length][maze[0].length];
        boolean[][] visited = new boolean[maze.length][maze[0].length];
        for (int[] row: distance)
            Arrays.fill(row, Integer.MAX_VALUE);
        distance[start[0]][start[1]] = 0;
        dijkstra(maze, distance, visited);
        return distance[dest[0]][dest[1]] == Integer.MAX_VALUE ? -1 : distance[dest[0]][dest[1]];
    }
    public int[] minDistance(int[][] distance, boolean[][] visited) {
        int[] min={-1,-1};
        int min_val = Integer.MAX_VALUE;
        for (int i = 0; i < distance.length; i++) {
            for (int j = 0; j < distance[0].length; j++) {
                if (!visited[i][j] && distance[i][j] < min_val) {
                    min = new int[] {i, j};
                    min_val = distance[i][j];
                }
            }
        }
        return min;
    }
    public void dijkstra(int[][] maze, int[][] distance, boolean[][] visited) {
        int[][] dirs={{0,1},{0,-1},{-1,0},{1,0}};
        while (true) {
            int[] s = minDistance(distance, visited);
            if (s[0] < 0)
```

Complexity Analysis

- Time complexity : $O((mn)^2)$. Complete traversal of maze will be done in the worst case and function `minDistance` takes $O(mn)$ time.
- Space complexity : $O(mn)$. $distance$ array of size $m*n$ is used.

Approach #4 Using Dijkstra Algorithm and Priority Queue[Accepted]

Algorithm

In the last approach, in order to choose the current node, we traversed over the whole $distance$ array and found out an unvisited node at the shortest distance from the $start$ node. Rather than doing this, we can do the same task much efficiently by making use of a Priority Queue, $queue$. This priority queue is implemented internally in the form of a heap. The criteria used for heapifying is that the node which is unvisited and at the smallest distance from the $start$ node, is always present on the top of the heap. Thus, the node to be chosen as the current node, is always present at the front of the $queue$.

For every current node, we again try to traverse in all the possible directions. We determine the minimum number of steps(till now) required to reach all the end points possible from the current node. If any such end point can be reached in a lesser number of steps through the current path than the paths previously considered, we need to update its $distance$ entry.

Further, we add an entry corresponding to this node in the $queue$, since its $distance$ entry has been updated and we need to consider this node as the competitors for the next current node choice. Thus, the process remains the same as the last approach, except the way in which the pick out the current node(which is the unvisited node at the shortest distance from the $start$ node).

Java

```java
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]]) {
```

Complexity Analysis

- Time complexity : $O(mn*log(mn))$. Complete traversal of maze will be done in the worst case giving a factor of mn. Further, `poll` method is a combination of heapifying($O(log(n))$) and removing the top element($O(1)$) from the priority queue, and it takes $O(n)$ time for n elements. In the current case, `poll` introduces a factor of $log(mn)$.
- Space complexity : $O(mn)$. $distance$ array of size $m*n$ is used and $queue$ size can grow upto $m*n$ in worst case.

Analysis written by: @vinod23

Rate this article: