Find the shortest path in a maze

Given a maze in the form of a binary rectangular matrix, find the shortest path's length in the maze from a given source to a given destination. The path can only be constructed out of cells having value 1, and at any moment, we can only move one step in one of the four directions.

The valid moves are:

Go Top: (x, y) ——> (x – 1, y)
Go Left: (x, y) ——> (x, y – 1)
Go Down: (x, y) ——> (x + 1, y)
Go Right: (x, y) ——> (x, y + 1)

For example, consider the following binary matrix. If source = `(0, 0)` and destination = `(7, 5)`, the shortest path from source to destination has length 12.

[1 1 1 1 1 0 0 1 1 1]
[0 1 1 1 1 1 0 1 0 1]
[0 0 1 0 1 1 1 0 0 1]
[1 0 1 1 1 0 1 1 0 1]
[0 0 0 1 0 0 0 1 0 1]

[0 1 1 1 1 1 1 1 0 0]

[1 1 1 1 1 0 0 1 1 1]

[0 0 1 0 0 1 1 0 0 1]

Practice this problem

To find the maze's shortest path, search for all possible paths in the maze from the starting position to the goal position until all possibilities are exhausted. We can easily achieve this with the help of backtracking. The idea is to start from the given source cell in the matrix and explore all four paths possible and recursively check if they will lead to the destination or not. Then update the minimum path length whenever the destination cell is reached. If a path doesn't reach the destination or explored all possible routes from the current cell, backtrack. To make sure that the path is simple and doesn't contain any cycles, keep track of cells involved in the current path in a matrix, and before exploring any cell, ignore the cell if it is already covered in the current path.

Following is the C++, Java, and Python implementation of the idea:

C++

Java

Python

```
import sys


# Check if it is possible to go to (x, y) from the current position.
# function returns false if the cell is invalid, has value 0 or alrea
def isSafe(mat, visited, x, y):
    return 0 <= x < len(mat) and 0 <= y < len(mat[0]) and \
            not (mat[x][y] == 0 or visited[x][y])


# Find the shortest possible route in a matrix `mat` from source cell
# to destination cell `dest`.

# `min_dist` stores the length of the longest path from source to a d
# found so far, and `dist` maintains the length of the path from a so
```

```python

    # if the destination is found, update `min_dist`
    if (i, j) == dest:
        return min(dist, min_dist)

    # set (i, j) cell as visited
    visited[i][j] = 1

    # go to the bottom cell
    if isSafe(mat, visited, i + 1, j):
        min_dist = findShortestPath(mat, visited, i + 1, j, dest, min

    # go to the right cell
    if isSafe(mat, visited, i, j + 1):
        min_dist = findShortestPath(mat, visited, i, j + 1, dest, min

    # go to the top cell
    if isSafe(mat, visited, i - 1, j):
        min_dist = findShortestPath(mat, visited, i - 1, j, dest, min

    # go to the left cell
    if isSafe(mat, visited, i, j - 1):
        min_dist = findShortestPath(mat, visited, i, j - 1, dest, min

    # backtrack: remove (i, j) from the visited matrix
    visited[i][j] = 0

    return min_dist


# Wrapper over findShortestPath() function
def findShortestPathLength(mat, src, dest):

    # get source cell (i, j)
    i, j = src

    # get destination cell (x, y)
    x, y = dest

    # base case
    if not mat or len(mat) == 0 or mat[i][j] == 0 or mat[x][y] == 0:
        return -1

    # `M × N` matrix
    (M, N) = (len(mat), len(mat[0]))

    # construct an `M × N` matrix to keep track of visited cells
    visited = [[False for _ in range(N)] for _ in range(M)]

    min_dist = findShortestPath(mat, visited, i, j, dest)

    if min_dist != sys.maxsize:
        return min_dist
    else:
        return -1
```

```
if __name__ == '__main__':

    mat = [
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
    ]

    src = (0, 0)
    dest = (7, 5)

    min_dist = findShortestPathLength(mat, src, dest)

    if min_dist != -1:
        print("The shortest path from source to destination has lengt
    else:
        print("Destination cannot be reached from source")

```

Download Run Code

Output:

The shortest path from source to destination has length 12

The time complexity of the above backtracking solution will be higher since all paths need to be traveled. However, since it is the shortest path problem, Breadth–first search (BFS) would be an ideal choice. If BFS is used to solve this problem, we travel level by level. So the destination node's first occurrence gives us the result, and we can stop our search there. The BFS approach is discussed here.

Backtracking, Matrix

Maze, Medium, Recursive