Shortest path in a maze – Lee Algorithm

Given a maze in the form of the binary rectangular matrix, find the shortest path's length in a maze from a given source to a given destination.

The path can only be constructed out of cells having value 1, and at any given 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]

[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]

Practice this problem

We have already discussed a backtracking solution in the previous post. The time complexity of the 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.

The **Lee algorithm** is one possible solution for maze routing problems based on Breadth–first search. It always gives an optimal solution, if one exists, but is slow and requires considerable memory. Following is the complete algorithm:

1. Create an empty queue and enqueue the source cell having a distance 0 from the source (itself) and mark it as visited.
2. Loop till queue is empty.
    - Dequeue the front node.
    - If the popped node is the destination node, then return its distance.
    - Otherwise, for each of four adjacent cells of the current cell, enqueue each valid cell with `+1` distance and mark them as visited.
3. If all the queue nodes are processed, and the destination is not reached, then return false.

Note that in BFS, all cells having the shortest path as 1 are visited first, followed by their adjacent cells having the shortest path as `1 + 1 = 2` and so on… So if we reach any node in BFS, its shortest path is one more than the shortest path of the parent. So, the destination cell's first occurrence gives us the result, and we can stop our search there. It is impossible that the shortest path exists from some other cell for which we haven't reached the given node yet. If any such path were possible, we would have already explored it.

Following is the C++, Java, and Python program that demonstrates it:

C++

Java

Python

```python
import sys
from collections import deque

# Below lists detail all four possible movements from a cell
row = [-1, 0, 0, 1]
col = [0, -1, 1, 0]


# Function to check if it is possible to go to position (row, col)
# from the current position. The function returns false if row, col
# is not a valid position or has a value 0 or already visited.
def isValid(mat, visited, row, col):
    return (row >= 0) and (row < len(mat)) and (col >= 0) and (col <
            and mat[row][col] == 1 and not visited[row][col]


# Find the shortest possible route in a matrix `mat` from source `src
# destination `dest`
def findShortestPathLength(mat, src, dest):

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

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

    # base case: invalid input
    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]))
```

```
    visited = [[False for x in range(N)] for y in range(M)]

    # create an empty queue
    q = deque()

    # mark the source cell as visited and enqueue the source node
    visited[i][j] = True

    # (i, j, dist) represents matrix cell coordinates, and their
    # minimum distance from the source
    q.append((i, j, 0))

    # stores length of the longest path from source to destination
    min_dist = sys.maxsize

    # loop till queue is empty
    while q:

        # dequeue front node and process it
        (i, j, dist) = q.popleft()

        # (i, j) represents a current cell, and `dist` stores its
        # minimum distance from the source

        # if the destination is found, update `min_dist` and stop
        if i == x and j == y:
            min_dist = dist
            break

        # check for all four possible movements from the current cell
        # and enqueue each valid movement
        for k in range(4):
            # check if it is possible to go to position
            # (i + row[k], j + col[k]) from current position
            if isValid(mat, visited, i + row[k], j + col[k]):
                # mark next cell as visited and enqueue it
                visited[i + row[k]][j + col[k]] = True
                q.append((i + row[k], j + col[k], dist + 1))

    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]
```

```
92          [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
93      ]
94
95      src = (0, 0)
96      dest = (7, 5)
97
98      min_dist = findShortestPathLength(mat, src, dest)
99
100     if min_dist != -1:
101         print("The shortest path from source to destination has lengt
102     else:
103         print("Destination cannot be reached from source")
104
```

Download Run Code

Output:

```
The shortest path from source to destination has length 12
```

The time complexity of the proposed solution is $O(M \times N)$ and requires $O(M \times N)$ extra space, where `M` and `N` are dimensions of the matrix.

Exercise: Extend the solution to print the shortest path from source to destination.

Matrix, Queue

Algorithm, Breadth-first search, FIFO, Maze, Medium, Must Know