6. Breadth First search grid shortest path

Many problems in graph theory can be represented using a grid. Grids are a form of implicit graph because we can determine a node's neighbours based on our location within the grid.

for example, avoiding obstacles and reaching a destination

→ A common approach to solving graph theory problems on grids is to first convert the grid to a familiar format like adjacency list / matrix.

Empty grid

0	1
2	3
4	5

First label the cells in the grid with numbers $[0, n)$ where

n = # rows × # columns.

→ Assume grid is unweighted and cells connect left, right, up, down.

Adjacency list

0 → [1, 2]
1 → [0, 3]
2 → [0, 3, 4]
3 → [1, 2, 5]
4 → [2, 5]
5 → [3, 4]

Adjacency matrix

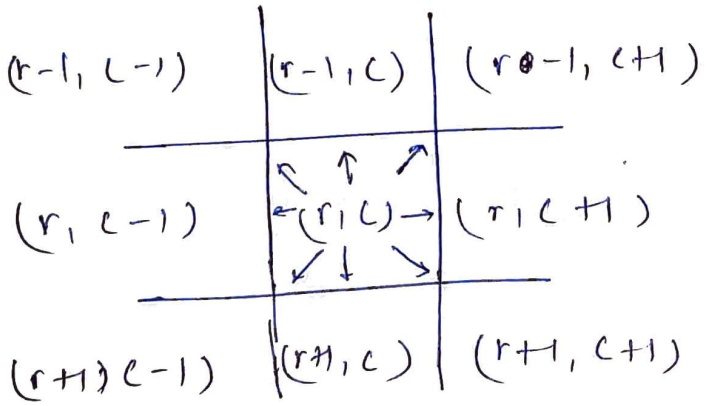
	0	1	2	3	4	5
0	0	1	1	0	0	0
1	1	0	0	1	0	0
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	0	1	0	0	1
5	0	0	0	1	1	0

Once we have an adjacency list / matrix, we can run whatever specialised graph algorithm to solve our problem such as : shortest path, connected components etc.

However, transformations between graph representations can usually be avoided due to structure of a grid.

⇒ Direction vectors

If we are at (r, c), we can add the row vectors [-1, 0], [1, 0], [0, 1], and [0, -1] to reach adjacent cells. (-1, -1), [1, 1], [-1, 1], [1, -1] if diagonal movement allowed.



This makes it very easy to access neighbouring cells from the current row, column position.

```
# Define direction vectors for
# north, south, east, west.

dr = [-1, +1, 0, 0]
dc = [ 0, 0, 1, -1]

for (i=0 ; i<4; i++):

    rr = r + dr[i]
    cc = c + dc[i]

    # skip invalid cells. Assume R and C
    # for number of rows and colums.

    if (rr<0 or cc<0): continue
    if (rr>=R or cc>=C): continue.

    # (rr, cc) is a neighbouring cell of (r,c).
```

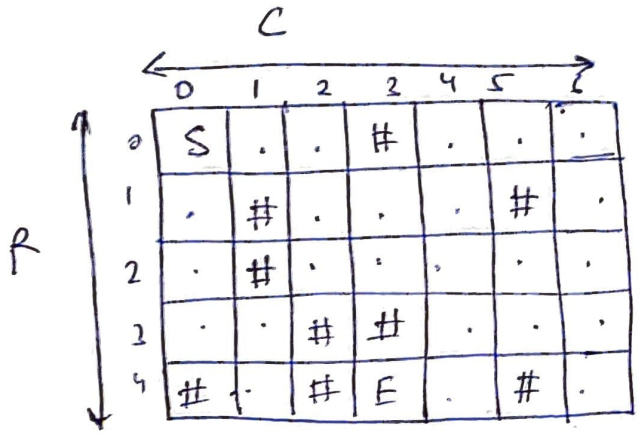
→ Dungeon Problem

You are trapped in a 2D dungeon and need to find quickest way out. Dungeon is composed of unit cubes which may or may not be filled with rocks. It takes one minute to move one unit north, west, east, south. You cannot move diagonally and the maze is surrounded by rocks on all sides.

Is an escape possible? If yes, how long will it take?

Dungeon has a size of R x C and you start at cell 'S' and there is an exit at cell 'E'. A cell full of rock is indicated by '#'. and empty cells are represented by '.'.

ex.

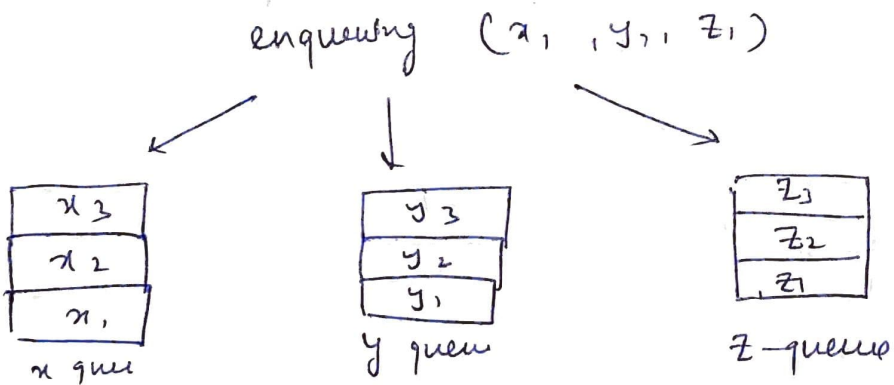


start at S node coordinate by adding (s_r, s_c) to a queue.

Now, start a breadth first search and terminate when E coordinate comes in the queue.

Alternative state representation

Use one queue for each dimension. So, in a 3D grid, we would have one queue for each of x, y, z dimensions.



```
# Global / class scope variables.
R, C = ...   # R = number of rows, C = number of columns.
m = ...      # input character matrix of size R x C
sr, sc = ... # 'S' symbol row & column values.
rq, cq = ... # Empty row queue (RQ) and column
                                    queue (CQ).

# variables used to track the number of steps taken.
move_count = 0
nodes_left_in_layer = 1
nodes_in_next_layer = 0.

# variable to track if 'E' character is reached.
reached_end = false;

# R x C matrix with 'false' to check if position (i,j)
# has been visited.
   visited = ...

# North, south, east, west direction vectors.
dr = [-1, +1, 0, 0]
dc = [0, 0, +1, -1].
```

```
function solve():
    rq.enqueue(sr)
    cq.enqueue(sc)
    visited[sr][sc] = true

    while (rq.size() > 0):          # or cq.size() > 0
        r = rq.dequeue()
        c = rc.dequeue()
        if (m[r][c] == 'E'):
            reached_end = true
            break
        explore_neighbours(r, c)
        nodes_left_in_layer--
        if nodes_left_in_layer == 0:
            nodes_left_in_layer = nodes_in_next_layer
            nodes_in_next_layer = 0
            move_count++
    if reached_end:
        return move_count
    return -1
```

```
function explore_neighbours (r, c):

    for (i = 0; i < 4; i++):

        rr = r + dr[i]
        cc = c + dc[i]

        # skip out of bound locations
        if rr < 0 or cc < 0 : continue
        if rr >= R or cc ≥ R : continue.

        # Skip visited locations or blocked cells
        if visited[rr][cc] : continue
        if m[rr][cc] == '#' : continue.

        rq.enqueu(rr)
        cq.enqueu(cc)
        visited[rr][cc] = true
        nodes_in_next_layer ++.
```