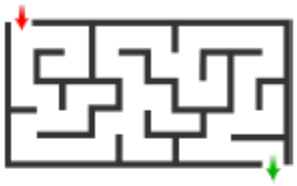
Maze Solver

1. Problem Defenation:

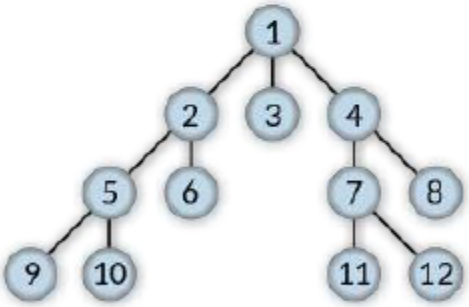
You are expected to solve a 2-D maze using BFS. A maze is path typically, from start node 'S' to Goal node 'E'.



Input: 2D maze represented as a string.

Output: the full path from Start node to End node (Goal Node) and direct path to go from Start to End directly.

Example: let's say the end node is 6.



Full Path: 1, 2, 3, 4, 5, 6.

Path: 1, 2, 6.

2. Introduction:

Breadth-first search algorithm:

It is an algorithm for searching or traverse a tree or a graph, It starts from the first node the root and discover its children in the next level of the tree and after discovering the whole children and finish this level it discovers the next level and so on until discovering the whole tree or graph.

It's applications:

- ❖ Traverse a tree or graph.
- ❖ Search for an element in graph or tree.
- ❖ Find the shortest path or the Minimum spanning tree.
- ❖ GPS navigation system.

2. The algorithm:

2.1. Breadth-first search

2.1.1. The main steps of the algorithm:

- Step 1: initialize a queue and enqueue the starting node.
- Step 2: initialize an empty list.
- Step 3: check if the queue is empty exit.
- Step 4: dequeue Node form the queue and append it on the list.
- Step 5: check if the Node is your final state exit.
- Step 6: enqueue all the Node's children in the queue.
- Step 7: repeat from step 3.

2.1.2. The implementation of the algorithm:

```python
class Node:
    id = None
    up = None
    down = None
    left = None
    right = None
    previousNode = None

    def __init__(self, value):
        self.value = value


class SearchAlgorithms:
    """ * DON'T change Class, Function or Parameters Names and Order
        * You can add ANY extra functions,
          classes you need as long as the main
          structure is left as is """
    path = []  # Represents the correct path from start node to the goal node.
    fullPath = []  # Represents all visited nodes from the start node to the goal node.
    maze = []  # The 2D array Map
    rows = 0  # Number of rows
    columns = 0  # Number of columns
```

```python
def __init__(self, mazeStr):
    """ mazeStr contains the full board
     The board is read row wise,
    the nodes are numbered 0-based starting
    the leftmost node"""
    self.maze = [j.split(',') for j in mazeStr.split(' ')]
    # get number of columns
    for i in mazeStr:
        if i == ',':
            self.columns += 1
        if i == ' ':
            break
    # get number of rows
    for i in mazeStr:
        if i == ' ':
            self.rows += 1
    pass
```

```
def BFS(self):
    """Implement Here"""
    queue = []
    visited = []
    x = 0
    y = 0
    for i in range(self.rows):
        for j in range(self.columns):
            if self.maze[i][j] == 'S':
                x = i
                y = j
    currentNode = self.maze[x][y]
    n = Node(currentNode)
    n.id = (x, y)
    queue.append(n)
    while currentNode != 'E':
        n = queue.pop(0)
        x = n.id[0]
        y = n.id[1]
        self.fullPath.append((x * (self.columns + 1)) + y)
        currentNode = self.maze[x][y]
        if x + 1 <= self.rows:
            if (x + 1, y) not in visited:
                n.up = (x + 1, y)
                if self.maze[x + 1][y] != '#':
                    n1 = Node(self.maze[x + 1][y])
                    n1.id = (x + 1, y)
                    n1.previousNode = n
                    queue.append(n1)
                    visited.append((x + 1, y))
```

```
    if x - 1 >= 0:
        if (x - 1, y) not in visited:
            n.down = (x - 1, y)
            if self.maze[x - 1][y] != '#':
                n1 = Node(self.maze[x - 1][y])
                n1.id = (x - 1, y)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x - 1, y))

    if y - 1 >= 0:
        if (x, y - 1) not in visited:
            n.left = (x, y - 1)
            if self.maze[x][y - 1] != '#':
                n1 = Node(self.maze[x][y - 1])
                n1.id = (x, y - 1)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x, y - 1))
    if y + 1 <= self.columns:
        if (x, y + 1) not in visited:
            n.right = (x, y + 1)
            if self.maze[x][y + 1] != '#':
                n1 = Node(self.maze[x][y + 1])
                n1.id = (x, y + 1)
                queue.append(n1)
                n1.previousNode = n
                visited.append((x, y + 1))
    visited.append((x, y))
self.path.append((n.id[0] * (self.columns + 1)) + n.id[1])
```

```
    self.path.append((n.id[0]*(self.columns+1))+n.id[1])
    while n.value != 'S':
        n = n.previousNode
        self.path.append((n.id[0]*(self.columns+1))+n.id[1])
    self.path.reverse()
    return self.fullPath, self.path
```

2.1.3. Sample run (the output):

```
**BFS**
 Full Path is: [0, 7, 1, 14, 2, 21, 9, 22, 16, 10, 29, 17, 11, 18, 4, 25, 19, 5, 32, 26, 20, 6, 31]
 Path: [0, 1, 2, 9, 16, 17, 18, 25, 32, 31]
```

3. Discussion

Breadth-first search algorithm:

- In Class SearchAlgorithms initialize:
    1. Path list that will have the shortest path from start to end.
    2. FullPath list that will have all visited nodes from stat to end.
    3. Maze 2D array that will carry the whole maze in a 2D array form.
    4. Rows and columns have the number of rows and columns in the 2D array.
- In Class SearchAlgorithms write a function _init_ that will make 2 things:
    1. Take the string maze and covert it to 2D array and put it in the maze the 2D array initialized before.
    2. Count the number of Rows and columns and put it in the variables initialized before.

- In Class SearchAlgorithms write a function BFS that will make the BFS algorithms Steps:
    1. First initialize a queue that will be the queue used to implement the BFS and visited list that will carry the visited nodes to prevent processing on them again and x and y that's carry the row and column of the Start node by searching for it on the 2D array maze and currentNode that will carry the value of the node will be proceeds on and n that will carry the node itself by creating and object from class Node and set its value and it's id which is a pair of the node's position (row and column) then append this node in the queue.
    2. Start a loop that will stop after reaching the End and in the loop dequeue from the queue node then put it in n. Then set x and y with n's position stored in the id. Then append in the fullPath the ID of this node in the string by calculating it from x and y. Then set the currentNode with this node value.
    3. Enqueue the nodes from this node to all available moves in the 4 dimensions (Left, Right, Up, Down) and append them in the visited list and append the node itself.
    4. Finally, outside the loop append the last point in the fullPath. Then loop from the end node and append it and its previous node until reaching the the start and append all this node in the path then reverse it.