Maze Problem Description

1-Overview

The Maze Problem involves navigating through a maze from a starting point to a goal while avoiding obstacles. This is a classic problem in computer science and artificial intelligence, often used to demonstrate search algorithms and optimization techniques. The goal is to find the best path (shortest, safest, or most efficient) within the constraints of the maze structure.

2-Problem Breakdown

To approach the maze problem, we can generate sub-questions that help structure the solution:

2.1. What is the layout of the maze?

A maze can be represented as a grid (2D array) where:

- **0 represents a free space.**
- **1 represents an obstacle.**
- **Start and goal points are marked.**

2.2. How do we represent the path?

- **Paths can be represented as sequences of grid coordinates (e.g., [(0,0), (0,1), (1,1)]).**
- **Each step corresponds to a valid move (e.g., up, down, left, right).**

2.3. What algorithms can solve the problem?

Common algorithms include:

- **Breadth-First Search (BFS): Finds the shortest path.**
- **Depth-First Search (DFS): Explores all paths but may not find the shortest.**
- **A* Search: Combines path cost and heuristic to efficiently find the best path.**
- **Dijkstra's Algorithm: guarantees the shortest path but is less efficient than A*.**

2.4. How do we handle obstacles?

- **Obstacles block movement. Algorithms must avoid paths that pass through cells marked as obstacles (1 in the grid).**

2.5. How do we evaluate performance?

Performance metrics include:

- **Path length.**
- **Computational efficiency (time/space complexity).**
- **Number of nodes explored.**

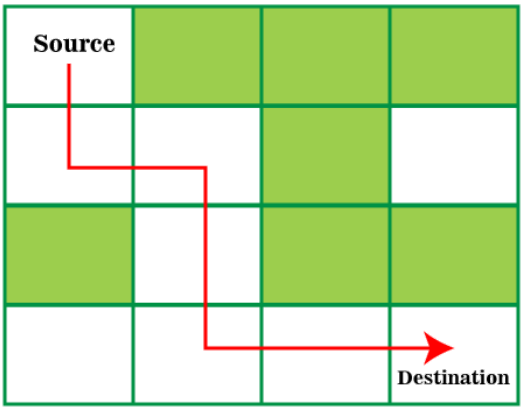
3-Example

3.1. Maze Representation

```
Maze:
[ [0, 1, 0, 0],
  [0, 0, 1, 0],
  [1, 0, 1, 1],
  [0, 0, 0, 0] ]

Start: (0, 0)
Goal: (3, 3)
```

3.2Maze Solution:



Maze

4-Applications of the Maze Problem

- **Robotics:** **Navigating through environments with obstacles.**
- **Video Games:** **Pathfinding for NPCs or characters.**
- **AI Research:** **demonstrating algorithms for search and optimization.**
- **Network Routing:** **Finding optimal data transmission paths.**

Resources:

1. **https://www.geeksforgeeks.org/rat-in-a-maze/?ref=gcse_outind**
2. **https://www.javatpoint.com/rat-in-a-maze-problem-in-java**

Node Class Code

1. Overview:

The Node class implements a basic structure for representing nodes in a tree data structure. It provides methods to calculate the tree depth, manage child nodes, and visualize the levels of the tree. While the implementation covers several key features, there are issues and areas for improvement that need to be addressed.

---

2. Code Breakdown:

1. Initialization (__init__):

```python
class Node:
    def __init__(self, value = None, children:list = [], cost = None, heuristic = None):
        self.value = value
        self.children = children
        self.heuristic = heuristic
        self.cost = cost
```

Purpose: The constructor initializes a `Node` object with the following attributes:

1. `value`: The value associated with the node (e.g., data).
2. `children`: A list of child nodes (defaults to an empty list).
3. `heuristic`: An optional heuristic value, likely for search algorithms.
4. `cost`: An optional cost value, potentially for weighted graphs or trees.

2. Static Method: max depth:

```python
    @staticmethod
    def max_depth(node):
        if node.children is None:
            return 1
        return 1 + max([Node.max_depth(child) for child in node.children])
```

Purpose: Calculates the maximum depth of the tree starting from a given `node`.

3. Method: add children:

```
def add_children(self, children:list):
    self.children.extend(children)
```

4. **Purpose**: Adds a list of child nodes to the current node's children list.
5. **String Representation (__str__):**

```
def __str__(self):
    return f"<{self.value}>"
```

Purpose: Returns a string representation of the `Node` object, showing its `value`

6. **Private Method: `next level`**

```
def __next_level(self, level):
    next_level = []
    for node in level:
        next_level.extend(node.children)
    return next_level
```

Purpose: Given a list of nodes (level), this method computes the next level of the tree by aggregating the children of all nodes in the current level.

7. **Method: is leaf**

```
def is_leaf(self):
    return len(self.children) == 0
```

Purpose: Checks if the node is a leaf (i.e., has no children).

8. **Method: print levels**

```
def print_levels(self):
    height = Node.max_depth(self)
    cur_level = [self.value ]
    for _ in range(height):
        print(cur_level)
        cur_level = self.__next_level(cur_level)
```

Purpose: Prints the values of nodes at each level of the tree, starting from the root.

Maze graph builder

1. Method: create a node

```
def create_node(parent, move, target):
    new_val = (parent.value[0] + move[0], parent.value[1] + move[1])
    distance_to_goal = get_distance(new_val, target)
    new_node = Node(new_val, [], 1, distance_to_goal)

    return new_node
```

Purpose: Checks if the node is a leaf (i.e., has no children).

2. Method: in bounds

```
  new_val = node.value
  return 0 <= new_val[1] < width and 0 <= new_val[0] < length
```

This function checks if the given node's coordinates are within the bounds of the maze.

Parameters:

node (Node): The node whose coordinates need to be checked.

width (int): The width of the maze in cells.

length (int): The length of the maze in cells.

Returns:

bool: True if the node's coordinates are within the maze bounds, False otherwise.

3. Method: get children

```
moves = [(0, 1), (1, 0), (-1, 0), (0, -1)]
children = []
for move in moves:
    new_node = create_node(node, move, target)

    if valid_node(new_node, width, length) and maze[new_node.value[0]][new_node.value[1]] == 0:
        children.append(new_node)
        created.add(new_node.value)

return children
```

This function generates child nodes for the given node in the maze based on valid moves (up, down, left, right). It constructs a list of child nodes that are within the bounds of the maze and have not been visited yet.

Parameters:

node (Node): The parent node from which child nodes are generated.

target (tuple): The coordinates of the target node in the maze.

width (int): The width of the maze in cells.

length (int): The length of the maze in cells.

Returns:

list: A list of child nodes that are within the bounds of the maze and have not been visited yet.

4. Method: get graph root

```
que = deque()
root = Node(start, [], 0, 0)
LENGTH = len(maze)
WIDTH = len(maze[0])

que.append(root)

while que:
    current = que.popleft()
    current.children = get_children(current, target, WIDTH, LENGTH, maze)

    for child in current.children:
        que.append(child)

return root
```

This function constructs a graph representation of a maze using the A* search algorithm.

The graph is represented by a tree where each node corresponds to a cell in the maze.

The function starts from the given start node and explores the maze by generating child nodes

based on valid moves (up, down, left, right) until the target node is reached.

Parameters:

maze (list of lists): A 2D list representing the maze. Each cell contains either '0' (empty) or '1' (wall).

start (tuple): The coordinates of the starting node in the maze.

target (tuple): The coordinates of the target node in the maze.

Returns:

Node: The root node of the constructed graph. The graph is represented by the root node and its children.

Maze builder Code

1. Overview:

The provided code implements a procedural maze generation algorithm with the following key functionalities:

- Randomly determines a valid path from a start point to a goal point.
- Dynamically adds complexity by introducing obstacles and additional paths.
- Offers a flexible approach to create mazes of arbitrary size and complexity.

While the code achieves its objectives, there are several inefficiencies and issues that can be addressed to improve its reliability and performance.

---

2. Code Breakdown:

1. Function: get direction

```
def get_direction():
    right = (0, 1)
    left = (0, -1)
    up = (-1, 0)
    down = (1, 0)
    chance = randint(1,100)

    direction = None
    if chance <= 100:  # Go right
        direction = right
    if chance <= 50:  # Go down
        direction = down
    if chance <= 20:  # Go up
        direction = up
    if chance <= 10:  # Go left
        direction = left

    return direction
```

Purpose: Randomly selects a direction (right, left, up, or down) with weighted probabilities.

2. Function: get valid path

```
def get_valid_path(maze, start, goal):
    path = [start]
    width = len(maze[0])
    length = len(maze)

    while start != goal:
        direction = get_direction()
        new_block = (start[0] + direction[0], start[1] + direction[1])

        while not (0 <= new_block[1] < width and 0 <= new_block[0] < length):
            direction = get_direction()
            new_block = (start[0] + direction[0], start[1] + direction[1])

        start = (new_block[0], new_block[1])
        path.append(start)
```

Purpose: Generates a valid path from the start point to the goal point by randomly moving in valid directions within the maze boundaries.

3. Function: get empty maze

```
def get_empty_maze(rows_num, cols_num):
    maze_template = [[1 for i in range(cols_num)] for j in range(rows_num)]
    return maze_template
```

Purpose: Creates an empty maze filled with walls (represented by 1).

4. Function: complicate maze

```
def complicate_maze(maze, rows, cols, path, complexity):
    crosses = complexity
    while crosses > 0:
        cross_point = path[randint(0, len(path)-1)]
        length = complexity
        direction = get_direction()
        while length:
            cross_point = (cross_point[0] + direction[0], cross_point[1] + direction[1])
            if 0 <= cross_point[1] < cols and 0 <= cross_point[0] < rows:
                maze[cross_point[0]][cross_point[1]] = 0
                path.append(cross_point)
                length -= 1
            else:
                break
        crosses -= 1
```

Purpose: Adds additional paths or obstacles to the maze by extending the valid path and carving out new paths.

5. Function: get maze

```
def get_maze(rows, cols, complexity):
    maze_template = get_empty_maze(rows, cols)
    valid_path = get_valid_path(maze_template, (0, 0), (rows-1, cols-1))

    for cell in valid_path:
        maze_template[cell[0]][cell[1]] = 0

    complicate_maze(maze_template, rows, cols, valid_path, complexity)
    return maze_template
```

Purpose: Combines the other functions to generate a complete maze with a valid path and added complexity**.**