Assignment 2 - Code
Mihael Zlatev - 1MI3400543

This report describes the implementation of a python program using Forward Chaining and Backward Chaining approach. The maze may contain obstacles, and explorable cells are logically inferred through logical rules.

Maze representation

The maze is represented via 2D Array with "0" or "1" as it's elements. "0" means that the node is free and there isn't any obstacle, whereas "1" means there is an obstacle. The maze class has the following fields:

- N (number of rows in the 2D array)
- M (number of columns in the 2D array)
- Grid (2D array representing the maze)
- Obstacles (tuples representing where in the maze to populate "1" node)

To construct a new Maze, you need to provide:

1. N – positive integer
2. M – positive integer
3. Obstacles (can be None)

```
class Maze:
    def __init__(self, n, m, obstacles=None):
        if n <= 0 or m <= 0:
            raise ValueError('Invalid maze dimensions. N and M must be greater than 
0')
        self.n = n
        self.m = m
        self.grid = np.zeros((n, m))
        self.set_obstacles(obstacles)
```

Forward chaining

Facts

1. **MazeCell(x,y)** – self.grid[x][y] = 0

2. **Obstacle(x,y)** – grid[x][y] = 1

```
    def set_obstacles(self, obstacles=None):
        if not obstacles:
            return
        for i, j in obstacles:
            if 0 <= i < self.n and 0 <= j < self.m:
                self.grid[i][j] = 1
```

3. **Start(x,y), End(x,y)** – passed as parameters to the searching algorithm

```
solver = KnowledgeBasedMazeSolver(maze)
    path, visited = solver.find_path(start_node, end_node)
```

Rules

1. **ValidMove(x, y) – move is valid when is in the grid and it's not an obstacle**

```
    def is_valid_move(self, row, col):
        return (self.is_in_grid(row, col) and not self.is_obstacle(row, col))

    def is_in_grid(self, row, col):
        return (0 <= row < self.n and 0 <= col < self.m)

    def is_obstacle(self, row, col):
        return (self.grid[row][col] == 1)
```

2. **Neighbours(x1, y1, x2, y2) – node is a tuple of (x,y), get_neighbors(x1, y1) returns all combinations where Neighbours(x1, y1, x2, y2) is true**
   - node is a tuple of (x,y)

```
def get_neighbors(self, node):
        row, col = node
        neighbors = []

        for dx, dy in DIRECTIONS:
            new_row, new_col = row + dx, col + dy
            if self.is_valid_move(new_row, new_col):
                neighbors.append((new_row, new_col))

        return neighbors
```

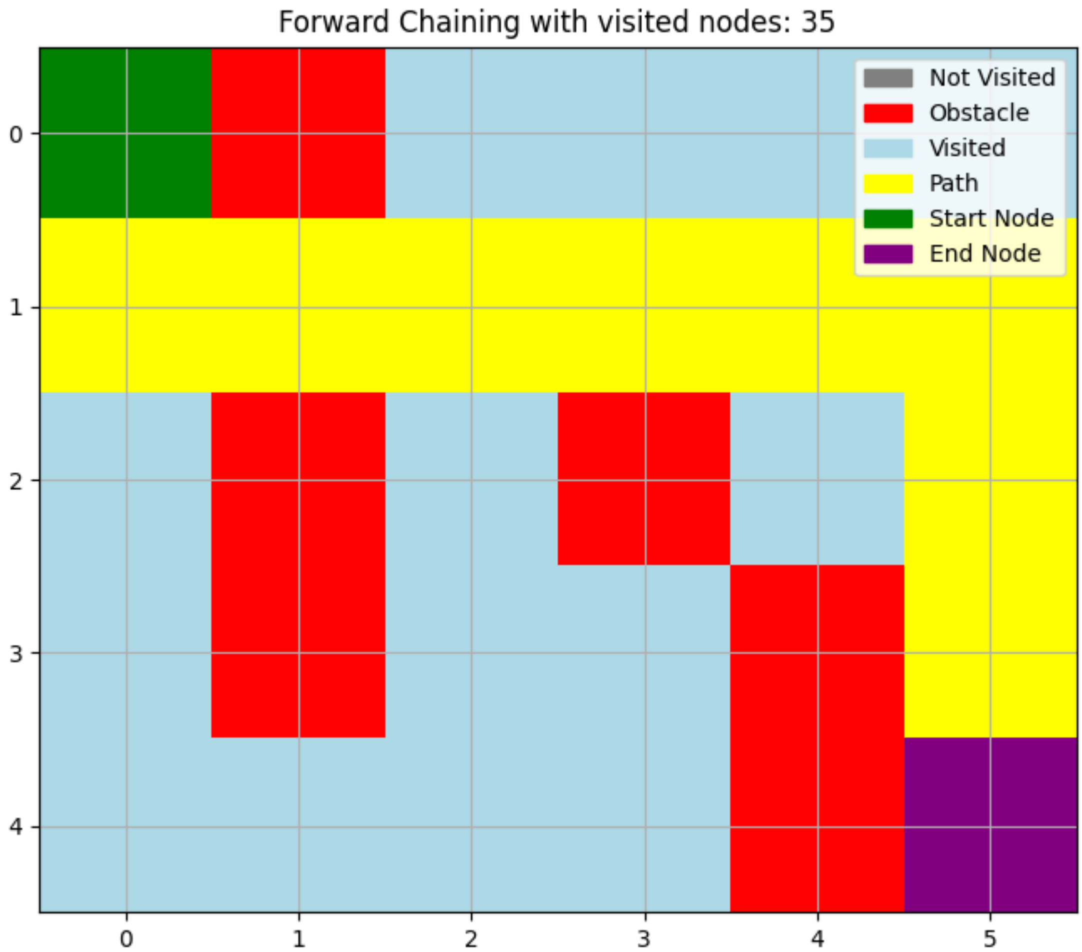
3. **Explorable(x,y) – adding new cells to the path via the Explorable rule and forward chaining**

```
explorable = [(start, [start])]

        while explorable:
            current, path = explorable.pop(0)
            visited.append(current)

            if current == end:
                return path, visited

            for neighbor in self.maze.get_neighbors(current):
                if neighbor not in visited:
                    new_path = path + [neighbor]
                    explorable.append((neighbor, new_path))

        return None, None
```

Example run of Forward Chaining

Result:

Path: [(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (3, 5), (4, 5)]



Backward Chaining

Facts

- Same as forward chaining facts

Rules

1. Predecessor(x1, y1, x2, y2)
get_predecessors(x1, x2) returns all combinations where Predecessor(x2, y2, x1, y2) is true
- node is a tuple of (x,y)

```
    def get_predecessors(self, node):
        row, col = node
        predecessors = []

        for dx, dy in DIRECTIONS:
            new_row, new_col = row - dx, col - dy
            if self.is_valid_move(new_row, new_col):
                predecessors.append((new_row, new_col))

        return predecessors
```

2. Path Existence(x,y) – adding new cells to the path if path exists through the predecessors

```
    def backward_chain(self, current, end, path, visited):
        if end == current:
            return path, path

        if current in visited:
            return None, None

        visited.append(current)

        valid_predecessors = self.maze.get_predecessors(current)
        for predecessor in valid_predecessors:
            if predecessor not in visited:
                new_path =  path + [predecessor]

                result_path, result_visited = self.backward_chain(predecessor, end,
new_path, visited)
                if result_path:
                    return result_path, result_visited

        return None, None
```

The backward chaining algorithm is started from the END node and via the PathExistence rule tries to find a path to START node.

```
return self.backward_chain(current=end, end=start, path=[end_node], visited=[])
```

Example run of Backward Chaining

Result:

Path: [(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (3, 5), (4, 5)]