Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

ICS 1403 – Introduction To Artificial Intelligence
Theory Assignment – 1
Title: Search Problem Application
Name: A S Tritthik Thilagar
RegNo: 3122237001057
Semester: IV
Year : 2025

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

SEARCH FOR ESCAPE PATH

Problem Description:

A maze is a rectangular grid where each cell is either empty (' ') or a wall ('W'). Given a start and goal cell, the task is to find a path using Depth-First Search (DFS), ensuring that movement is only through empty cells. The program should display the maze, find paths, and print the sequence of moves.

Data Definitions:

- **Maze**: A list of lists representing the grid.
  - `' '` (space) → Empty cell
  - `'W'` → Wall
- **Cell Position**: A tuple `(x, y)` representing row and column index.
- **Path**: A list of cell positions showing the route from start to goal.
- **Neighbors**: Adjacent cells (up, down, left, right) that are empty.

Program Construction

Function: neighbours(x,y)

1. Specification

Returns the list of valid adjacent cells for a given cell (x, y).

2. Examples

```
maze = [
    [" ", "W", " "],
    [" ", " ", "W"],
    ["W", " ", " "]
]
neighbours(1, 1)  # Output: [(1, 0), (2, 1)]
neighbours(0, 0)  # Output: [(1, 0)]
```

3. Trace

For neighbours(1,1):

- Right (1,2): Wall
- Down (2,1): Empty
- Left (1,0): Empty
- Up (0,1): Wall

  Final output: [(1,0), (2,1)]

4. Algorithm Design

1. Define movement directions (right, down, left, up).
2. Filter out positions that are out of bounds or walls.

Function: find_paths

1. Specification

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

Finds paths from the start cell to the goal cell using DFS.

2. Examples

```
find_paths(maze, (0,0), (2,2))
# Output: [(0,0), (1,0), (1,1), (2,1), (2,2)]
```

3. Trace

For find_paths(maze, (0,0), (2,2)):

1. Start at (0,0), move to (1,0).
2. Move to (1,1), then (2,1).
3. Move to (2,2), goal reached.

4. Algorithm Design

1. Initialize a stack with the start cell and an empty path.
2. Use DFS to explore neighbours.
3. If the goal is reached, save the path.

Function: display_path

1. Specification

Displays the maze with the path marked as 'X'.

2. Examples

```
display_path(maze, [(0,0), (1,0), (1,1), (2,1), (2,2)])
```

Output:

X W
X X W
W X X

3. Trace

For display_path(maze, [(0,0), (1,0), (1,1), (2,1), (2,2)]):

- Mark X at (0,0), (1,0), (1,1), (2,1), (2,2).

4. Algorithm Design

1. Create a copy of the maze.
2. Replace path cells with 'X'.
3. Print the updated maze.

Python Implementation:

```
class MazeSolver:
    def __init__(self, maze):
        self.maze = [list(row) for row in maze]
        self.rows = len(maze)
        self.cols = len(maze[0])
        self.paths = []

    def display_maze(self):
        for row in self.maze:
            print(" ".join(row))
        print()

    def neighbors(self, x, y):
```

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

```
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        return [(x + dx, y + dy) for dx, dy in directions
                if 0 <= x + dx < self.rows and 0 <= y + dy < self.cols
and self.maze[x + dx][y + dy] == ' ']

    def find_paths(self, start, goal):
        if self.maze[start[0]][start[1]] == 'W' or
self.maze[goal[0]][goal[1]] == 'W':
            print("Start or goal is inside a wall! No path possible.")
            return

        stack = [(start, [start])]
        visited = set()

        while stack:
            (x, y), path = stack.pop()
            if (x, y) == goal:
                self.paths.append(path)
                continue
            if (x, y) in visited:
                continue
            visited.add((x, y))
            for nx, ny in self.neighbors(x, y):
                if (nx, ny) not in visited:
                    stack.append(((nx, ny), path + [(nx, ny)]))

    def display_path(self, path):
        temp_maze = [row[:] for row in self.maze]
        for x, y in path:
            temp_maze[x][y] = 'P'
        for row in temp_maze:
            print(" ".join(row))
        print("\nSequence of Moves:")
        for i in range(0,len(path)-1):
            current=path[i]
            next=path[i+1]
            print(path[i] , " --> ", path[i+1],end=" := ")
            if(current[0]==next[0]):
                if(current[1]<next[1]):
                    print("Move RIGHT")
                else:
                    print("Move LEFT")
            else:
                if(current[0]<next[0]):
                    print("Move DOWN")
                else:
                    print("Move UP")

    def get_next_path(self):
        if not self.paths:
            return None
        return self.paths.pop(0)
```

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

```
maze_data = [
    [" ", " ", " ", " ", " ", "W", " ", "W"],
    ["W", " ", "W", "W", " ", " ", " ", " "],
    ["W", " ", "W", " ", "W", "W", "W", " "],
    [" ", " ", " ", " ", " ", "W", " ", " "],
    ["W", "W", "W", "W", " ", "W", "W", " "],
    ["W", " ", " ", " ", " ", " ", "W", " "],
    [" ", " ", "W", "W", "W", " ", "W", " "],
    ["W", " ", "W", " ", " ", " ", "W", "W"],
    [" ", " ", "W", " ", "W", " ", " ", " "],
    [" ", "W", " ", " ", "W", "W", "W", " "],
    ["W", " ", "W", " ", "W", " ", "W", " "],
    [" ", " ", " ", " ", "W", " ", " ", " "],
]

start = (0, 0)
goal = (11, 5)

solver = MazeSolver(maze_data)
solver.find_paths(start, goal)

print("Maze Representation:")
solver.display_maze()

path = solver.get_next_path()
if path:
    print("Path Found:")
    solver.display_path(path)
else:
    print("No path found!")
```

Sample Output:

Maze Representation:

```
          W   W
W   W W
W   W   W W W
          W
W W W W   W W
W           W
    W W W   W
W   W       W W
    W   W
  W     W W W
W   W   W   W
        W
```

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

Path Found:

```
P P       W   W
W P W W
W P W   W W W
  P P P P W
W W W W P W W
W       P P W
    W W W P W
W   W     P W W
    W   W P P P
  W     W W W P
W   W   W   W P
        W P P P
```

Sequence of Moves:
(0, 0) --> (0, 1) := Move RIGHT
(0, 1) --> (1, 1) := Move DOWN
(1, 1) --> (2, 1) := Move DOWN
(2, 1) --> (3, 1) := Move DOWN
(3, 1) --> (3, 2) := Move RIGHT
(3, 2) --> (3, 3) := Move RIGHT
(3, 3) --> (3, 4) := Move RIGHT
(3, 4) --> (4, 4) := Move DOWN
(4, 4) --> (5, 4) := Move DOWN
(5, 4) --> (5, 5) := Move RIGHT
(5, 5) --> (6, 5) := Move DOWN
(6, 5) --> (7, 5) := Move DOWN
(7, 5) --> (8, 5) := Move DOWN
(8, 5) --> (8, 6) := Move RIGHT
(8, 6) --> (8, 7) := Move RIGHT
(8, 7) --> (9, 7) := Move DOWN
(9, 7) --> (10, 7) := Move DOWN
(10, 7) --> (11, 7) := Move DOWN

(11, 7) --> (11, 6) := Move LEFT
(11, 6) --> (11, 5) := Move LEFT

MAGIC SQUARE

Problem Description:

We have a 3×3 sliding puzzle with tiles numbered 1 to 8 and one empty space. The goal is to reach a specific arrangement using valid moves. A move consists of sliding an adjacent tile into the empty space.

The initial state:
7 2 4
5 6
8 3 1

The goal state:
1 2
3 4 5
6 7 8
We will use A Search* with a misplaced tiles heuristic to find an optimal solution.

Data Definitions:

State Representation:

A 3×3 list (matrix) to represent the puzzle.
The empty space is denoted by 0.
Moves:
A tile adjacent to 0 can slide into it.
Path:
A list of states representing the sequence of moves.

Program Construction:

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

Function: neighbours(s)

1. Specification

Returns all valid next states from state s by sliding adjacent tiles into the empty space.

2. Examples

neighbours([[7, 2, 4], [5, 6, 0], [8, 3, 1]])

Possible Outputs:

Moving 6 down:
7 2 4
5 0 6
8 3 1
Moving 3 up:
7 2 4
5 6 1
8 3 0

3. Trace:

Find the empty space (0).
Identify valid moves (tiles that can slide).
Swap empty space with a tile to generate new states.

4. Algorithm Design

Locate 0 in the 3×3 grid.
Define up, down, left, right moves.
Swap 0 with a neighboring tile and return new states.

Class: SearchProblem:

1. Specification

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

Defines the puzzle problem with a start state, goal state, and the neighbours() function.

Class: Searcher (A Search Implementation)*:

1. Specification
Implements A* search using a priority queue with the misplaced tiles heuristic.

2. Algorithm Design
Use a priority queue (min-heap) for efficient path selection.
Define cost function:
Path cost (g): Number of moves.
Heuristic (h): Count of misplaced tiles.
Total cost (f = g + h).
Store visited states to avoid cycles.

Class: PathFinder (Modified Search for Multiple Paths)

1. Specification
Returns multiple solutions by continuing search after finding one.

Python Implementation:

```
from heapq import heappop, heappush
import itertools

def print_board(state):
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()

class EightPuzzle:
    def __init__(self, start, goal):
        self.start = start
        self.goal = goal
        self.goal_pos = {num: (r, c) for r, row in enumerate(goal) for
c, num in enumerate(row)}

    def find_empty(self, state):
```

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

```
        for r, row in enumerate(state):
            for c, num in enumerate(row):
                if num == 0:
                    return r, c
        return None

    def neighbors(self, state):
        r, c = self.find_empty(state)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        neighbors = []
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            if 0 <= nr < 3 and 0 <= nc < 3:
                new_state = [list(row) for row in state]
                new_state[r][c], new_state[nr][nc] = new_state[nr][nc],
new_state[r][c]
                neighbors.append((tuple(tuple(row) for row in
new_state), (nr, nc)))
        return neighbors

    def heuristic(self, state):
        return sum(1 for r, row in enumerate(state) for c, num in
enumerate(row) if num != 0 and (r, c) != self.goal_pos[num])

    def solve(self):
        pq = []
        counter = itertools.count()
        start_tuple = tuple(tuple(row) for row in self.start)
        heappush(pq, (0, next(counter), start_tuple, [], 0))
        visited = set()

        while pq:
            _, _, state, path, cost = heappop(pq)
            if state == tuple(tuple(row) for row in self.goal):
                return path + [state]
            if state in visited:
                continue
            visited.add(state)

            for neighbor, (nr, nc) in self.neighbors(state):
                if neighbor not in visited:
                    heappush(pq, (cost + self.heuristic(neighbor) + 1,
next(counter), neighbor, path + [state], cost + 1))
        return None

# Test Case
start_state = [
    [7, 2, 4],
    [5, 6, 0],
    [8, 3, 1]
]
```

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

```
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

ep = EightPuzzle(start_state, goal_state)
solution = ep.solve()

if solution:
    for i, step in enumerate(solution):
        print(f"Step {i}:")
        print_board(step)
else:
    print("No solution found!")
```

TESTING:

Step 0:
7 2 4
5 6
8 3 1

Step 1:
7 2 4
5   6
8 3 1

Step 2:
7 2 4
5 3 6
8   1

Step 3:
7 2 4
5 3 6
8 1

Step 4:
7 2 4
5 3
8 1 6

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

Step 5:
7 2 4
5   3
8 1 6

Step 6:
7 2 4
** 5 3**
8 1 6

Step 7:
** 2 4**
7 5 3
8 1 6

Step 8:
2   4
7 5 3
8 1 6

Step 9:
2 4
7 5 3
8 1 6

Step 10:
2 4 3
7 5
8 1 6

Step 11:
2 4 3
7   5
8 1 6

Step 12:
2 4 3
7 1 5

8   6

Step 13:
2 4 3
7 1 5
** 8 6**

Step 14:
2 4 3
** 1 5**
7 8 6

Step 15:
2 4 3
1   5
7 8 6

Step 16:
2   3
1 4 5
7 8 6

Step 17:
** 2 3**
1 4 5
7 8 6

Step 18:
1 2 3
** 4 5**
7 8 6

Step 19:
1 2 3
4   5
7 8 6

Step 20:

Date: 5-3-2025
Reg No: 3122237001057
Name: A.S.TRITTHIK THILAGAR

1 2 3
4 5
7 8 6

Step 21:
1 2 3
4 5 6
7 8