25/10/24

LAB-3

Q) Implement A* algorithm using Python

function A* search (problem) returns a solution or failure

node ← a node n with n.state = problem.initial state, n.g = 0

frontier ← a priority queue ordered by ascending g+h, only element 'n'

loop do

if empty?(frontier) then return failure

n ← pop(frontier)

if problem.goalTest(n.state) then return solution(n)

for each action a in a problem actions(n.states) do

n' ← childNode(problem, n, a)

insert(n', g(n') + h(n'), frontier)

OUTPUT:

$$\begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 0, 7, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 0, 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 0 \end{bmatrix}$$

Using Manhattan distance:

Initial state

1 2 3
4 5 6
0 7 8

Goal state

1 2 3
4 5 6
7 8 0

$f(0) = 0 + 2 = 2$

1 2 3
0 5 6
4 7 8

$f(1) = 1 + 3 = 4$

1 2 3
4 5 6
7 0 8

$f(1) = 1 + 1 = 2$

1 2 3
4 5 6
0 7 8

$f(3) = 2 + 2 = 4$

1 2 3
4 0 6
7 5 8

$f(2) = 2 + 2 = 4$

1 2 3
4 5 6
7 8 0

$f(3) = 2 + 0 = 2$

✓

25/10/24

```python
#Number of misplaced tiles
import heapq

class PuzzleState:
    def __init__(self, board, g, h):
        self.board = board  # The current state of the board
        self.g = g  # Cost to reach this node (depth)
        self.h = h  # Heuristic cost (misplaced tiles)
        self.f = g + h  # Total cost (f(n) = g(n) + h(n))

    def __lt__(self, other):
        return self.f < other.f  # For priority queue to sort by f(n)

def print_board(board):
    """Print the current board state."""
    for row in board:
        print(" ".join(str(num) for num in row))
    print()  # Empty line for better readability

def get_blank_position(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:  # Find the blank space (0)
                return (i, j)

def get_successors(state):
    successors = []
    x, y = get_blank_position(state.board)  # Get position of blank tile
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Possible moves
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Valid move
            new_board = [row[:] for row in state.board]  # Copy the current board
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]  # Swap
            successors.append(PuzzleState(new_board, state.g + 1, 0))  # Create new state
    return successors
```

```python
def heuristic_misplaced_tiles(board):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0 and board[i][j] != i * 3 + j + 1:  # Check for misplaced tiles
                misplaced += 1
    return misplaced

def is_goal_state(board):
    return board == [[1, 2, 3],
                     [8, 0, 4],
                     [7, 6, 5]]  # Check if the board is in the goal state

def a_star_search_misplaced_tiles(start_board):
    start_state = PuzzleState(start_board, 0, heuristic_misplaced_tiles(start_board))
    open_set = []
    heapq.heappush(open_set, start_state)
    closed_set = set()

    while open_set:
        current_state = heapq.heappop(open_set)
        # Print current board state and details
        print("Current board state:")
        print_board(current_state.board)
        print(f"g(n): {current_state.g}, h(n): {current_state.h}, f(n): {current_state.f}\n")

        # Check if we've reached the goal
        if is_goal_state(current_state.board):
            print("Goal state reached!")
            return current_state.g  # Return the cost to reach the goal

        closed_set.add(tuple(map(tuple, current_state.board)))

        for successor in get_successors(current_state):
            successor.h = heuristic_misplaced_tiles(successor.board)
            successor.f = successor.g + successor.h
```

```python
                successor.f = successor.g + successor.h
                if tuple(map(tuple, successor.board)) in closed_set:
                    continue
                heapq.heappush(open_set, successor)

    return None  # No solution found

def get_user_input():
    board = []
    for i in range(3):
        while True:
            row = input(f"Enter row {i + 1} (3 numbers separated by space): ")
            nums = list(map(int, row.split()))
            if len(nums) == 3 and all(0 <= num <= 8 for num in nums):
                board.append(nums)
                break
            else:
                print("Invalid input. Please enter 3 numbers between 0 and 8.")
    return board

print("Tanish M V")
print("1BM22CS302")
print("Number of misplaced tiles")
if __name__ == "__main__":
    start_board = get_user_input()
    steps = a_star_search_misplaced_tiles(start_board)
    print(f"Steps to solve with Misplaced Tiles heuristic: {steps}")
```

```
Tanish M V
1BM22CS302
Enter row 1 (3 numbers separated by space): 2 8 3
Enter row 2 (3 numbers separated by space): 1 6 4
Enter row 3 (3 numbers separated by space): 0 7 5
Current board state:
2 8 3
1 6 4
0 7 5

g(n): 0, h(n): 7, f(n): 7

Current board state:
2 8 3
1 6 4
7 0 5

g(n): 1, h(n): 6, f(n): 7

Current board state:
2 8 3
0 6 4
1 7 5

g(n): 1, h(n): 7, f(n): 8

Current board state:
2 8 3
1 0 4
7 6 5

g(n): 2, h(n): 6, f(n): 8

Current board state:
2 8 3
1 6 4
7 5 0
```

```
g(n): 2, h(n): 6, f(n): 8

Current board state:
0 8 3
2 6 4
1 7 5

g(n): 2, h(n): 7, f(n): 9

Current board state:
2 8 3
1 4 0
7 6 5

g(n): 3, h(n): 6, f(n): 9

Current board state:
2 8 3
1 6 0
7 5 4

g(n): 3, h(n): 6, f(n): 9

Current board state:
2 8 3
6 0 4
1 7 5

g(n): 2, h(n): 7, f(n): 9

Current board state:
2 8 3
0 1 4
7 6 5

g(n): 3, h(n): 6, f(n): 9
```

```
Current board state:
2 0 3
1 8 4
7 6 5

g(n): 3, h(n): 6, f(n): 9

Current board state:
2 8 3
1 0 6
7 5 4

g(n): 4, h(n): 5, f(n): 9

Current board state:
0 2 3
1 8 4
7 6 5

g(n): 4, h(n): 5, f(n): 9

Current board state:
2 8 3
1 5 6
7 0 4

g(n): 5, h(n): 4, f(n): 9

Current board state:
1 2 3
0 8 4
7 6 5

g(n): 5, h(n): 4, f(n): 9
```

```
Current board state:
2 8 3
0 1 6
7 5 4

g(n): 5, h(n): 5, f(n): 10

Current board state:
2 8 3
1 5 6
7 4 0

g(n): 6, h(n): 4, f(n): 10

Current board state:
8 0 3
2 6 4
1 7 5

g(n): 3, h(n): 7, f(n): 10

Current board state:
2 0 3
1 8 6
7 5 4

g(n): 5, h(n): 5, f(n): 10

Current board state:
1 2 3
8 0 4
7 6 5

g(n): 6, h(n): 4, f(n): 10

Goal state reached!
Steps to solve with Misplaced Tiles heuristic: 6
```

```python
#Manhattan distance
import heapq

class PuzzleState:
    def __init__(self, board, g, h):
        self.board = board
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
        return self.f < other.f

def print_board(board):
    for row in board:
        print(" ".join(str(num) for num in row))
    print()

def get_blank_position(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return (i, j)

def get_successors(state):
    successors = []
    x, y = get_blank_position(state.board)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_board = [row[:] for row in state.board]
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
            successors.append(PuzzleState(new_board, state.g + 1, 0))
    return successors

```

```
def heuristic_manhattan_distance(board):
    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                target_x = (board[i][j] - 1) // 3
                target_y = (board[i][j] - 1) % 3
                distance += abs(i - target_x) + abs(j - target_y)
    return distance

def is_goal_state(board):
    return board == [[1, 2, 3],
                     [8, 0, 4],
                     [7, 6, 5]]

def a_star_search_manhattan_distance(start_board):
    start_state = PuzzleState(start_board, 0, heuristic_manhattan_distance(start_board))
    open_set = []
    heapq.heappush(open_set, start_state)
    closed_set = set()

    while open_set:
        current_state = heapq.heappop(open_set)
        print("Current board state:")
        print_board(current_state.board)
        print(f"g(n): {current_state.g}, h(n): {current_state.h}, f(n): {current_state.f}\n")

        if is_goal_state(current_state.board):
            print("Goal state reached!")
            return current_state.g

        closed_set.add(tuple(map(tuple, current_state.board)))

        for successor in get_successors(current_state):
            successor.h = heuristic_manhattan_distance(successor.board)
```

```python
                successor.h = heuristic_manhattan_distance(successor.board)
                successor.f = successor.g + successor.h
                if tuple(map(tuple, successor.board)) in closed_set:
                    continue
                heapq.heappush(open_set, successor)

    return None

def get_user_input():
    board = []
    for i in range(3):
        while True:
            row = input(f"Enter row {i + 1} (3 numbers separated by space): ")
            nums = list(map(int, row.split()))
            if len(nums) == 3 and all(0 <= num <= 8 for num in nums):
                board.append(nums)
                break
            else:
                print("Invalid input. Please enter 3 numbers between 0 and 8.")
    return board

print("Tanish M V")
print("1BM22CS302")
print("Manhattan distance")
if __name__ == "__main__":
    start_board = get_user_input()
    steps = a_star_search_manhattan_distance(start_board)
    print(f"Steps to solve with Manhattan Distance heuristic: {steps}")
```

```
Tanish M V
1BM22CS302
Manhattan distance
Enter row 1 (3 numbers separated by space): 2 8 3
Enter row 2 (3 numbers separated by space): 1 6 4
Enter row 3 (3 numbers separated by space): 0 7 5
Current board state:
2 8 3
1 6 4
0 7 5

g(n): 0, h(n): 10, f(n): 10

Current board state:
2 8 3
1 6 4
7 0 5

g(n): 1, h(n): 9, f(n): 10

Current board state:
2 8 3
1 6 4
7 5 0

g(n): 2, h(n): 8, f(n): 10

Current board state:
2 8 3
1 0 4
7 6 5

g(n): 2, h(n): 10, f(n): 12

Current board state:
2 8 3
0 6 4
1 7 5
```

```
g(n): 1, h(n): 11, f(n): 12

Current board state:
2 0 3
1 8 4
7 6 5

g(n): 3, h(n): 9, f(n): 12

Current board state:
2 8 3
1 4 0
7 6 5

g(n): 3, h(n): 9, f(n): 12

Current board state:
0 2 3
1 8 4
7 6 5

g(n): 4, h(n): 8, f(n): 12

Current board state:
2 8 3
1 6 0
7 5 4

g(n): 3, h(n): 9, f(n): 12

Current board state:
2 8 3
1 4 5
7 6 0

g(n): 4, h(n): 8, f(n): 12
```

```
Current board state:
1 2 3
0 8 4
7 6 5

g(n): 5, h(n): 7, f(n): 12

Current board state:
2 8 3
1 0 6
7 5 4

g(n): 4, h(n): 8, f(n): 12

Current board state:
2 8 3
1 4 5
7 0 6

g(n): 5, h(n): 7, f(n): 12

Current board state:
2 0 3
1 8 6
7 5 4

g(n): 5, h(n): 7, f(n): 12

Current board state:
2 8 3
1 5 6
7 0 4

g(n): 5, h(n): 7, f(n): 12
```

```
Current board state:
0 2 3
1 8 6
7 5 4

g(n): 6, h(n): 6, f(n): 12

Current board state:
2 8 3
1 5 6
7 4 0

g(n): 6, h(n): 6, f(n): 12

Current board state:
1 2 3
0 8 6
7 5 4

g(n): 7, h(n): 5, f(n): 12

Current board state:
2 8 0
1 4 3
7 6 5

g(n): 4, h(n): 10, f(n): 14

Current board state:
2 3 0
1 8 6
7 5 4

g(n): 6, h(n): 8, f(n): 14
```

```
Current board state:
1 2 3
7 8 6
0 5 4

g(n): 8, h(n): 6, f(n): 14

Current board state:
1 2 3
8 0 6
7 5 4

g(n): 8, h(n): 6, f(n): 14

Current board state:
2 8 3
1 5 0
7 4 6

g(n): 7, h(n): 7, f(n): 14

Current board state:
2 8 3
6 0 4
1 7 5

g(n): 2, h(n): 12, f(n): 14

Current board state:
2 3 0
1 8 4
7 6 5

g(n): 4, h(n): 10, f(n): 14
```

```
Current board state:
0 8 3
2 6 4
1 7 5

g(n): 2, h(n): 12, f(n): 14

Current board state:
2 8 3
0 1 6
7 5 4

g(n): 5, h(n): 9, f(n): 14

Current board state:
2 8 3
1 4 5
0 7 6

g(n): 6, h(n): 8, f(n): 14

Current board state:
2 8 3
6 4 5
1 7 0

g(n): 4, h(n): 10, f(n): 14

Current board state:
1 2 3
8 0 4
7 6 5

g(n): 6, h(n): 8, f(n): 14

Goal state reached!
Steps to solve with Manhattan Distance heuristic: 6
```