# Ankushdeep Singh_102003174_COE7

## Assignment -4

## Artificial Intelligence

**1Q:**

If the initial and final states are as below and H(n): number of misplaced tiles in the current state $n$ as compared to the goal node need to be considered as the heuristic function. You need to use **Best First Search** algorithm.

Initial:

| | | |
|---|---|---|
| 2 | | 3 |
| 1 | 8 | 4 |
| 7 | 6 | 5 |

Goal:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

**CODE:**

ques_1.py > generate_successor_nodes

```python
from queue import PriorityQueue

# Function to calculate the number of misplaced tiles
def heuristic_misplaced_tiles(current_state, goal_state):
    misplaced_tiles = 0
    for i in range(len(current_state)):
        for j in range(len(current_state[0])):
            if current_state[i][j] != goal_state[i][j]:
                misplaced_tiles += 1
    return misplaced_tiles

# Function to check if the current state is the goal state
def is_goal_state(current_state, goal_state):
    return current_state == goal_state

# Function to generate all possible successor nodes
def generate_successor_nodes(current_state):
    successor_nodes = []
    empty_tile_index = None
    # Find the index of the empty tile (0)
    for i in range(len(current_state)):
        for j in range(len(current_state[0])):
            if current_state[i][j] == 0:
                empty_tile_index = (i, j)
                break

    # Check possible moves: up, down, left, right
    possible_moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    for move in possible_moves:
        new_index = (empty_tile_index[0] + move[0], empty_tile_index[1] + move[1])
        if 0 <= new_index[0] < len(current_state) and 0 <= new_index[1] < len(current_state[0]):
            new_state = [row[:] for row in current_state]
            new_state[empty_tile_index[0]][empty_tile_index[1]] = new_state[new_index[0]][new_index[1]]
            new_state[new_index[0]][new_index[1]] = 0
            successor_nodes.append(new_state)
    return successor_nodes
```

```python

# Best-First Search algorithm
def best_first_search(initial_state, goal_state):
    priority_queue = PriorityQueue()
    priority_queue.put((heuristic_misplaced_tiles(initial_state, goal_state), initial_state))
    visited = set(tuple(map(tuple, initial_state)))  # Keep track of visited states

```

```python
    while not priority_queue.empty():
        current_node = priority_queue.get()[1]

        if is_goal_state(current_node, goal_state):
            return current_node

        successors = generate_successor_nodes(current_node)

        for successor in successors:
            successor_tuple = tuple(map(tuple, successor))
            if successor_tuple not in visited:
                visited.add(successor_tuple)
                priority_queue.put((heuristic_misplaced_tiles(successor, goal_state), successor))
    return None  # No solution found

# Example usage
initial_state = [[2, 0, 3],
                 [1, 8, 4],
                 [7, 6, 5]]

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

print("Initial State:")
for row in initial_state:
    print(row)
print()

solution = best_first_search(initial_state, goal_state)

if solution is None:
    print("No solution found.")
else:
    print("Final State:")
    for row in solution:
        print(row)
```

# OUTPUT:

```
PS E:\SUMMER SEM\AI\LABS\ASS_4> python -u "e:\SUMMER SEM\AI\LABS\ASS_4\ques_1.py"
Initial State:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Final State:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

## 2Q:

If the initial and final states have been changed as below and approach you need to use is **Hill Climbing searching algorithm.** H(n): number of misplaced tiles in the current state *n* as compared to the goal node as the heuristic function for the following states.

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 5 | 4 |
| 7 | 6 | |

Initial State

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

Final State

## CODE:

ques_2.py > generate_neighbors

```python
import random
# Define the initial and final states
initial_state = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
final_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

# Calculate the number of misplaced tiles heuristic function
def misplaced_tiles(state):
    count = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] != final_state[i][j]:
                count += 1
    return count

# Generate random neighbors by swapping two tiles
def generate_neighbors(state):
    neighbors = []
    zero_row, zero_col = find_zero(state)

    # Generate neighbors by swapping with the left tile
    if zero_col > 0:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row][zero_col - 1] = new_state[zero_row][zero_col - 1], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    # Generate neighbors by swapping with the right tile
    if zero_col < len(state[0]) - 1:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row][zero_col + 1] = new_state[zero_row][zero_col + 1], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    # Generate neighbors by swapping with the above tile
    if zero_row > 0:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row - 1][zero_col] = new_state[zero_row - 1][zero_col], new_state[zero_row][zero_col]
        neighbors.append(new_state)
```

ques_2.py > generate_neighbors

```python

    # Generate neighbors by swapping with the below tile
    if zero_row < len(state) - 1:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row + 1][zero_col] = new_state[zero_row + 1][zero_col], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    return neighbors

# Find the position of the zero (blank) tile
def find_zero(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

# Implement the Hill Climbing algorithm
def hill_climbing(initial_state):
    current_state = initial_state

    while True:
        current_heuristic = misplaced_tiles(current_state)

        # Check if the current state is the goal state
        if current_heuristic == 0:
            return current_state

        neighbors = generate_neighbors(current_state)
        best_neighbor = None
        best_neighbor_heuristic = current_heuristic

        # Find the neighbor with the lowest heuristic value
        for neighbor in neighbors:
            neighbor_heuristic = misplaced_tiles(neighbor)
            if neighbor_heuristic < best_neighbor_heuristic:
                best_neighbor = neighbor
                best_neighbor_heuristic = neighbor_heuristic
```

ques_2.py > hill_climbing

```python

        # If no better neighbor is found, return the current state
        if best_neighbor is None or best_neighbor_heuristic >= current_heuristic:
            return current_state

        # Move to the best neighbor and continue the search
        current_state = best_neighbor

# Print the initial state
print("Initial state:")
for row in initial_state:
    print(row)

print()  # Add a blank line

# Run the Hill Climbing algorithm
result = hill_climbing(initial_state)

# Print the final state
print("Final state:")
for row in result:
    print(row)
```

**OUTPUT:**

```
> python -u "e:\SUMMER SEM\AI\LABS\ASS_4\ques_2.py"
Initial state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Final state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
PS E:\SUMMER SEM\AI\LABS\ASS_4>
```

## 3Q:

Apply A* **searching algorithm** by taking H(n): number of correctly placed tiles in the current state *n* as compared to the goal node. as the heuristic function.

Initial:

| | | |
|---|---|---|
| 2 | | 3 |
| 1 | 8 | 4 |
| 7 | 6 | 5 |

Goal:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

## CODE:

ques_3.py > ...

```python
import heapq
# Define the initial and final states
initial_state = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
final_state = [[1, 2, 4], [8, 0, 4], [7, 6, 5]]

# Define the heuristic function: number of correctly placed tiles
def correctly_placed_tiles(state):
    count = 0
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == final_state[i][j]:
                count += 1
    return count

# Define the A* search algorithm
def astar_search(initial_state):
    # Define a priority queue to store the states
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state))  # Priority is determined by the sum of heuristic and cost
    cost = {tuple(map(tuple, initial_state)): 0}  # Keep track of the cost to reach each state
    parent = {tuple(map(tuple, initial_state)): None}  # Keep track of the parent state

    while priority_queue:
        # Pop the state with the lowest priority from the queue
        _, current_state = heapq.heappop(priority_queue)

        # Check if the current state is the goal state
        if current_state == final_state:
            return current_state

        # Generate all possible neighbors
        zero_row, zero_col = find_zero(current_state)
        neighbors = generate_neighbors(current_state, zero_row, zero_col)
```

ques_3.py > ...

```python

        for neighbor in neighbors:
            neighbor_cost = cost[tuple(map(tuple, current_state))] + 1
            if tuple(map(tuple, neighbor)) not in cost or neighbor_cost < cost[tuple(map(tuple, neighbor))]:
                cost[tuple(map(tuple, neighbor))] = neighbor_cost
                priority = neighbor_cost + correctly_placed_tiles(neighbor)
                heapq.heappush(priority_queue, (priority, neighbor))
                parent[tuple(map(tuple, neighbor))] = current_state

    # No solution found
    return None

# Generate possible neighbors by swapping the zero (blank) tile with adjacent tiles
def generate_neighbors(state, zero_row, zero_col):
    neighbors = []

    if zero_col > 0:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row][zero_col - 1] = new_state[zero_row][zero_col - 1], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    if zero_col < len(state[0]) - 1:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row][zero_col + 1] = new_state[zero_row][zero_col + 1], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    if zero_row > 0:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row - 1][zero_col] = new_state[zero_row - 1][zero_col], new_state[zero_row][zero_col]
        neighbors.append(new_state)

    if zero_row < len(state) - 1:
        new_state = [row[:] for row in state]
        new_state[zero_row][zero_col], new_state[zero_row + 1][zero_col] = new_state[zero_row + 1][zero_col], new_state[zero_row][zero_col]
        neighbors.append(new_state)
```

ques_3.py > generate_neighbors

```python

    return neighbors

# Find the position of the zero (blank) tile
def find_zero(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

# Print the initial state
print("Initial state:")
for row in initial_state:
    print(row)

print()

# Run the A* search algorithm
result = astar_search(initial_state)

# Print the result
if result is not None:
    print("Final state:")
    for row in result:
        print(row)
else:
    print("Pattern not found.")
```

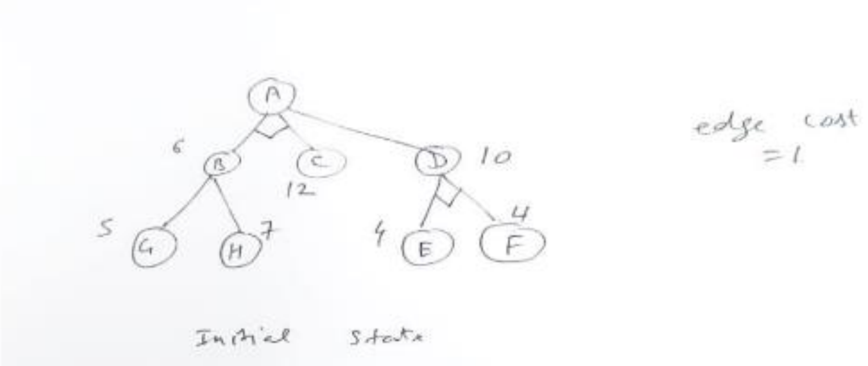
**OUTPUT:**

```
PS E:\SUMMER SEM\AI\LABS\ASS_4> python -u "e:\SUMMER SEM\AI\LABS\ASS_4\ques_3.py
"
Initial state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Pattern not found.
PS E:\SUMMER SEM\AI\LABS\ASS_4> 
```

# 4Q:

Apply **AO* searching algorithm** on the following search tree.




# CODE:

```
ques_4.py > ...
import heapq
# Define the search tree
search_tree = {
    'A': {'B': 1, 'C': 1},
    'B': {'G': 1, 'H': 1},
    'C': {},
    'D': {'E': 1, 'F': 1},
    'E': {},
    'F': {},
    'G': {},
    'H': {}
}

# Define the heuristic costs
heuristic_costs = {
    'A': 0,
    'B': 5,
    'C': 12,
    'D': 10,
    'E': 4,
    'F': 4,
    'G': 0,
    'H': 7
}

# Define the AO* search algorithm
def ao_star_search(start, goal):
    # Define a priority queue to store the nodes
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))  # Priority is determined by the estimated total cost
    g_cost = {start: 0}  # Keep track of the cost to reach each node
    parent = {start: None}  # Keep track of the parent node

    while priority_queue:
        # Pop the node with the lowest priority from the queue
        _, current_node = heapq.heappop(priority_queue)

```

ques_4.py > ao_star_search

```python
        # Check if the current node is the goal
        if current_node == goal:
            return construct_path(parent, current_node)

        # Explore the neighbors of the current node
        for neighbor in search_tree[current_node]:
            # Calculate the estimated total cost for the neighbor
            total_cost = g_cost[current_node] + search_tree[current_node][neighbor] + heuristic_costs[neighbor]

            # Update the cost and parent if the neighbor has not been visited or has a lower cost
            if neighbor not in g_cost or total_cost < g_cost[neighbor]:
                g_cost[neighbor] = g_cost[current_node] + search_tree[current_node][neighbor]
                heapq.heappush(priority_queue, (total_cost, neighbor))
                parent[neighbor] = current_node

                # Print intermediate state
                print("Intermediate state:")
                path = construct_path(parent, neighbor)
                print(' -> '.join(path))
                print()

    # No path found
    return None

# Helper function to construct the path from the start to the goal
def construct_path(parent, goal):
    path = [goal]
    current = goal
    while current != None:
        current = parent[current]
        if current is not None:
            path.append(current)
    path.reverse()
    return path
```

ques_4.py > ...

```python

# Define the start and goal nodes
start_node = 'A'
goal_node = 'G'

# Run the AO* search algorithm
path = ao_star_search(start_node, goal_node)

# Print the result
if path is not None:
    print("Final path found:")
    print(' -> '.join(path))
else:
    print("No path found.")
```

# OUTPUT:

```
PS E:\SUMMER SEM\AI\LABS\ASS_4> python -u "e:\SUMMER SEM\AI\LABS\ASS_4\ques_4.py"
Intermediate state:
A -> B

Intermediate state:
A -> C

Intermediate state:
A -> B -> G

Intermediate state:
A -> B -> H

Final path found:
A -> B -> G
PS E:\SUMMER SEM\AI\LABS\ASS_4>
```