

# 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
1  from queue import PriorityQueue
2
3  # Function to calculate the number of misplaced tiles
4  def heuristic_misplaced_tiles(current_state, goal_state):
5      misplaced_tiles = 0
6      for i in range(len(current_state)):
7          for j in range(len(current_state[0])):
8              if current_state[i][j] != goal_state[i][j]:
9                  misplaced_tiles += 1
10     return misplaced_tiles
11
12     # Function to check if the current state is the goal state
13     def is_goal_state(current_state, goal_state):
14         return current_state == goal_state
15
16     # Function to generate all possible successor nodes
17     def generate_successor_nodes(current_state):
18         successor_nodes = []
19         empty_tile_index = None
20         # Find the index of the empty tile (0)
21         for i in range(len(current_state)):
22             for j in range(len(current_state[0])):
23                 if current_state[i][j] == 0:
24                     empty_tile_index = (i, j)
25                     break
26
27         # Check possible moves: up, down, left, right
28         possible_moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
29
30         for move in possible_moves:
31             new_index = (empty_tile_index[0] + move[0], empty_tile_index[1] + move[1])
32             if 0 <= new_index[0] < len(current_state) and 0 <= new_index[1] < len(current_state[0]):
33                 new_state = [row[:] for row in current_state]
34                 new_state[empty_tile_index[0]][empty_tile_index[1]] = new_state[new_index[0]][new_index[1]]
35                 new_state[new_index[0]][new_index[1]] = 0
36                 successor_nodes.append(new_state)
37         return successor_nodes
38
39     # Best-First Search algorithm
40     def best_first_search(initial_state, goal_state):
41         priority_queue = PriorityQueue()
42         priority_queue.put((heuristic_misplaced_tiles(initial_state, goal_state), initial_state))
43         visited = set(tuple(map(tuple, initial_state))) # Keep track of visited states
44
```

```

45     while not priority_queue.empty():
46         current_node = priority_queue.get()[1]
47
48         if is_goal_state(current_node, goal_state):
49             return current_node
50
51         successors = generate_successor_nodes(current_node)
52
53         for successor in successors:
54             successor_tuple = tuple(map(tuple, successor))
55             if successor_tuple not in visited:
56                 visited.add(successor_tuple)
57                 priority_queue.put((heuristic_misplaced_tiles(successor, goal_state), successor))
58     return None # No solution found
59
60 # Example usage
61 initial_state = [[2, 0, 3],
62                 [1, 8, 4],
63                 [7, 6, 5]]
64
65 goal_state = [[1, 2, 3],
66              [8, 0, 4],
67              [7, 6, 5]]
68
69 print("Initial State:")
70 for row in initial_state:
71     print(row)
72 print()
73
74 solution = best_first_search(initial_state, goal_state)
75
76 if solution is None:
77     print("No solution found.")
78 else:
79     print("Final State:")
80     for row in solution:
81         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
1 import random
2 # Define the initial and final states
3 initial_state = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
4 final_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
5
6 # Calculate the number of misplaced tiles heuristic function
7 def misplaced_tiles(state):
8     count = 0
9     for i in range(len(state)):
10         for j in range(len(state[i])):
11             if state[i][j] != final_state[i][j]:
12                 count += 1
13     return count
14
15 # Generate random neighbors by swapping two tiles
16 def generate_neighbors(state):
17     neighbors = []
18     zero_row, zero_col = find_zero(state)
19
20     # Generate neighbors by swapping with the left tile
21     if zero_col > 0:
22         new_state = [row[:] for row in state]
23         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]
24         neighbors.append(new_state)
25
26     # Generate neighbors by swapping with the right tile
27     if zero_col < len(state[0]) - 1:
28         new_state = [row[:] for row in state]
29         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]
30         neighbors.append(new_state)
31
32     # Generate neighbors by swapping with the above tile
33     if zero_row > 0:
34         new_state = [row[:] for row in state]
35         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]
36         neighbors.append(new_state)
```

```
ques_2.py > generate_neighbors
37
38     # Generate neighbors by swapping with the below tile
39     if zero_row < len(state) - 1:
40         new_state = [row[:] for row in state]
41         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]
42         neighbors.append(new_state)
43
44     return neighbors
45
46 # Find the position of the zero (blank) tile
47 def find_zero(state):
48     for i in range(len(state)):
49         for j in range(len(state[i])):
50             if state[i][j] == 0:
51                 return i, j
52
53 # Implement the Hill Climbing algorithm
54 def hill_climbing(initial_state):
55     current_state = initial_state
56
57     while True:
58         current_heuristic = misplaced_tiles(current_state)
59
60         # Check if the current state is the goal state
61         if current_heuristic == 0:
62             return current_state
63
64         neighbors = generate_neighbors(current_state)
65         best_neighbor = None
66         best_neighbor_heuristic = current_heuristic
67
68         # Find the neighbor with the lowest heuristic value
69         for neighbor in neighbors:
70             neighbor_heuristic = misplaced_tiles(neighbor)
71             if neighbor_heuristic < best_neighbor_heuristic:
72                 best_neighbor = neighbor
73                 best_neighbor_heuristic = neighbor_heuristic
```

ques\_2.py > hill\_climbing

```
74
75     # If no better neighbor is found, return the current state
76     if best_neighbor is None or best_neighbor_heuristic >= current_heuristic:
77         return current_state
78
79     # Move to the best neighbor and continue the search
80     current_state = best_neighbor
81
82 # Print the initial state
83 print("Initial state:")
84 for row in initial_state:
85     print(row)
86
87 print() # Add a blank line
88
89 # Run the Hill Climbing algorithm
90 result = hill_climbing(initial_state)
91
92 # Print the final state
93 print("Final state:")
94 for row in result:
95     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 > ...
1 import heapq
2 # Define the initial and final states
3 initial_state = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
4 final_state = [[1, 2, 4], [8, 0, 4], [7, 6, 5]]
5
6 # Define the heuristic function: number of correctly placed tiles
7 def correctly_placed_tiles(state):
8     count = 0
9     for i in range(len(state)):
10         for j in range(len(state[i])):
11             if state[i][j] == final_state[i][j]:
12                 count += 1
13     return count
14
15 # Define the A* search algorithm
16 def astar_search(initial_state):
17     # Define a priority queue to store the states
18     priority_queue = []
19     heapq.heappush(priority_queue, (0, initial_state)) # Priority is determined by the sum of heuristic and cost
20     cost = {tuple(map(tuple, initial_state)): 0} # Keep track of the cost to reach each state
21     parent = {tuple(map(tuple, initial_state)): None} # Keep track of the parent state
22
23     while priority_queue:
24         # Pop the state with the lowest priority from the queue
25         _, current_state = heapq.heappop(priority_queue)
26
27         # Check if the current state is the goal state
28         if current_state == final_state:
29             return current_state
30
31         # Generate all possible neighbors
32         zero_row, zero_col = find_zero(current_state)
33         neighbors = generate_neighbors(current_state, zero_row, zero_col)
```

```
ques_3.py > ...
34
35     for neighbor in neighbors:
36         neighbor_cost = cost[tuple(map(tuple, current_state))] + 1
37         if tuple(map(tuple, neighbor)) not in cost or neighbor_cost < cost[tuple(map(tuple, neighbor))]:
38             cost[tuple(map(tuple, neighbor))] = neighbor_cost
39             priority = neighbor_cost + correctly_placed_tiles(neighbor)
40             heapq.heappush(priority_queue, (priority, neighbor))
41             parent[tuple(map(tuple, neighbor))] = current_state
42
43     # No solution found
44     return None
45
46 # Generate possible neighbors by swapping the zero (blank) tile with adjacent tiles
47 def generate_neighbors(state, zero_row, zero_col):
48     neighbors = []
49
50     if zero_col > 0:
51         new_state = [row[:] for row in state]
52         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]
53         neighbors.append(new_state)
54
55     if zero_col < len(state[0]) - 1:
56         new_state = [row[:] for row in state]
57         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]
58         neighbors.append(new_state)
59
60     if zero_row > 0:
61         new_state = [row[:] for row in state]
62         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]
63         neighbors.append(new_state)
64
65     if zero_row < len(state) - 1:
66         new_state = [row[:] for row in state]
67         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]
68         neighbors.append(new_state)
```

ques\_3.py > generate\_neighbors

```
69     return neighbors
70
71
72 # Find the position of the zero (blank) tile
73 def find_zero(state):
74     for i in range(len(state)):
75         for j in range(len(state[i])):
76             if state[i][j] == 0:
77                 return i, j
78
79 # Print the initial state
80 print("Initial state:")
81 for row in initial_state:
82     print(row)
83
84 print()
85
86 # Run the A* search algorithm
87 result = astar_search(initial_state)
88
89 # Print the result
90 if result is not None:
91     print("Final state:")
92     for row in result:
93         print(row)
94 else:
95     print("Pattern not found.")
```

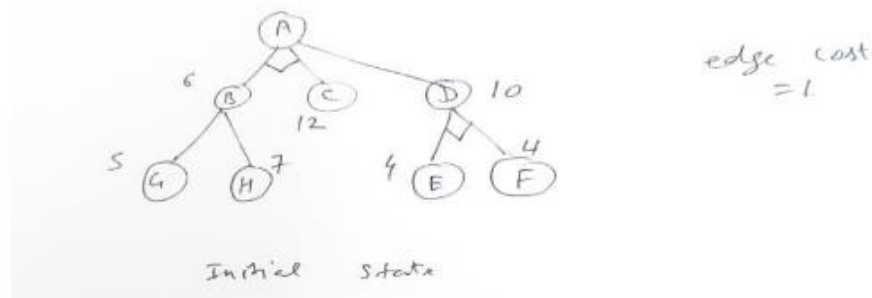
## OUTPUT:

```
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 > ...
1  import heapq
2  # Define the search tree
3  search_tree = {}
4      'A': {'B': 1, 'C': 1},
5      'B': {'G': 1, 'H': 1},
6      'C': {},
7      'D': {'E': 1, 'F': 1},
8      'E': {},
9      'F': {},
10     'G': {},
11     'H': {}
12 }
13
14 # Define the heuristic costs
15 heuristic_costs = {
16     'A': 0,
17     'B': 5,
18     'C': 12,
19     'D': 10,
20     'E': 4,
21     'F': 4,
22     'G': 0,
23     'H': 7
24 }
25
26 # Define the AO* search algorithm
27 def ao_star_search(start, goal):
28     # Define a priority queue to store the nodes
29     priority_queue = []
30     heapq.heappush(priority_queue, (0, start)) # Priority is determined by the estimated total cost
31     g_cost = {start: 0} # Keep track of the cost to reach each node
32     parent = {start: None} # Keep track of the parent node
33
34     while priority_queue:
35         # Pop the node with the lowest priority from the queue
36         _, current_node = heapq.heappop(priority_queue)
37

```

```

ques_4.py > ao_star_search
38     # Check if the current node is the goal
39     if current_node == goal:
40         return construct_path(parent, current_node)
41
42     # Explore the neighbors of the current node
43     for neighbor in search_tree[current_node]:
44         # Calculate the estimated total cost for the neighbor
45         total_cost = g_cost[current_node] + search_tree[current_node][neighbor] + heuristic_costs[neighbor]
46
47         # Update the cost and parent if the neighbor has not been visited or has a lower cost
48         if neighbor not in g_cost or total_cost < g_cost[neighbor]:
49             g_cost[neighbor] = g_cost[current_node] + search_tree[current_node][neighbor]
50             heapq.heappush(priority_queue, (total_cost, neighbor))
51             parent[neighbor] = current_node
52
53             # Print intermediate state
54             print("Intermediate state:")
55             path = construct_path(parent, neighbor)
56             print(' -> '.join(path))
57             print()
58
59     # No path found
60     return None
61
62 # Helper function to construct the path from the start to the goal
63 def construct_path(parent, goal):
64     path = [goal]
65     current = goal
66     while current != None:
67         current = parent[current]
68         if current is not None:
69             path.append(current)
70     path.reverse()
71     return path

```

```

ques_4.py > ...
72
73 # Define the start and goal nodes
74 start_node = 'A'
75 goal_node = 'G'
76
77 # Run the AO* search algorithm
78 path = ao_star_search(start_node, goal_node)
79
80 # Print the result
81 if path is not None:
82     print("Final path found:")
83     print(' -> '.join(path))
84 else:
85     print("No path found.")
86

```

## 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> 

```