ACI Assignment 1 - Team 148

(Question 1) Explain the environment of the agent:
PEAS DESCRIPTION

Performance Measure:

1. **Efficiency:** Minimize the total distance traveled by the drone.

2. **Completeness:** Ensure each blood bank is visited exactly once before reaching the hospital.

3. **Optimality:** The path chosen should be the one with the least distance from the available paths.

Environment:

1. **(Fully Observable/Partially Observable):** Fully, as the drone has access to the complete map and the distances between nodes (blood banks and hospital) are known.
2. **(Single Agent/Multi Agent):** Single, as the problem statement does not mention other agents or drones that the system must interact with.
3. **(Deterministic/stochastic):** Deterministic, since the outcome of taking an action (moving from one blood bank to another) is known.
4. **(Sequential/Episodic):** Sequential, because the agent's current decision affects the following decisions.
5. **(Static/Dynamic):** Static, since the blood banks and hospital locations do not change over time.
6. **(Discrete/Continuous):** Discrete, as there are a finite number of blood banks and a single hospital, similar to distinct nodes in a graph with defined paths.

Actuators:

1. **Motors/Propellers:** For movement and maintaining flight.
2. **Grippers/Clamps**: For picking up and securing blood supply containers.

Sensors:

1. **Position Sensors:** To determine the drone's current location on the map (not GPS, but based on the initial position and the path taken).
2. **Cargo Sensors:** To confirm that the blood supply has been picked up and is secured.

(Question 2) Define the heuristic and or fitness function for the given algorithms and the given problem.:

1. A* Algorithm Heuristic:

For the A* algorithm, the heuristic should estimate the cost to reach the goal from the current state. In this case, since we want to visit all blood banks exactly once before reaching the hospital, a suitable heuristic could be the "Average Distance to Unvisited Blood Banks (ADB)" from the current node. This heuristic can be admissible if it never overestimates the actual cost to reach the goal from the current node. The ADB heuristic is calculated as follows:

$$h(n) = \frac{\text{(Sum of distances from node n to each unvisited Blood bank)}}{\text{Total Number of Unvisited Blood banks}}$$

This heuristic encourages the drone to choose a path that minimizes the average distance to all remaining blood banks, balancing exploration of closer banks with the need to eventually reach all banks in an efficient manner.

2. Hill Climbing Algorithm Fitness Function:

The Hill Climbing algorithm uses a fitness function to determine the "goodness" of a state. For this problem, the fitness function could be the inverse of the path cost function. Since Hill Climbing seeks to maximize the fitness function, we can define it as the negative of the total distance traveled, because we want to minimize the distance:

$$\text{Fitness (n)} = - \text{Total distance traveled from start to n}$$

The fitness function evaluates the current path's length, and the algorithm will attempt to move to a neighboring state that has a higher fitness (lower total distance traveled). This way, the drone will "climb the hill" by moving towards shorter paths.

(Question 3) Use appropriate data structures and implement search algorithms (informed and local search) to find the path that covers all the blood bank with shortest distance in the city as provided in the graph. The starting point is to be obtained from the user as input.

1. Defining Data structures and helper functions

```
[72]  # Code Block : Set the matrix for transition & cost (as relevant for the given problem)
      graph = {'A': [('B', 5), ('C', 8)],
               'B': [('A', 5), ('C', 7), ('D', 6), ('E', 10), ('H', 8)],
               'C': [('A', 8), ('B', 7), ('F', 12)],
               'D': [('B', 6), ('H', 10)],
               'E': [('B', 10), ('F', 9), ('H', 18)],
               'F': [('C', 12), ('E', 9)]}

      goal_state = 'H'
      visited = {'H': False}
```

```
[73]  # Code Block : Set Initial State (Must handle dynamic inputs)
      unvisited = set(graph.keys())
      for g in graph.keys():
          visited[g] = False
```

```
# function to compute minimum cost between 2 nodes
def compute_min_cost(src, dest):
    initialize()
    find_all_paths(src, dest, visited, path)
    min_index = all_costs.index(min(all_costs)) # find index of the minimum cost among costs os all paths
    return all_costs[min_index]


def find_all_paths(u, d, visited, path): # function to compute all paths that exist between 2 nodes
    visited[u] = True
    path.append(u)
    if u == d:
        # print(path)
        cur_path = path.copy()
        all_paths.append(cur_path)
        final_cost = compute_pathcost(path)
        # print(final_cost)
        all_costs.append(final_cost)
    if u == 'H':
        pass # No more child nodes
    else:
        # Continue until all child nodes are travered (leaf nodes)
        for (m, weight) in graph[u]:
            if visited[m] == False:
                find_all_paths(m, d, visited, path)
    path.pop()
    visited[u] = False
```

Min Cost Graph

```
A [['A', 0], ['B', 5], ['C', 8], ['D', 11], ['E', 15], ['F', 20], ['H', 13]]
B [['A', 5], ['B', 0], ['C', 7], ['D', 6], ['E', 10], ['F', 19], ['H', 8]]
C [['A', 8], ['B', 7], ['C', 0], ['D', 13], ['E', 17], ['F', 12], ['H', 15]]
D [['A', 11], ['B', 6], ['C', 13], ['D', 0], ['E', 16], ['F', 25], ['H', 10]]
E [['A', 15], ['B', 10], ['C', 17], ['D', 16], ['E', 0], ['F', 9], ['H', 18]]
F [['A', 20], ['B', 19], ['C', 12], ['D', 25], ['E', 9], ['F', 0], ['H', 27]]
H [['A', 13], ['B', 8], ['C', 15], ['D', 10], ['E', 18], ['F', 27]]
```

Taking Input

DYNAMIC INPUT

IMPORTANT : Dynamic Input must be got in this section. Display the possible states to choose from: This is applicable for all the relevent problems as mentioned in the question.

```
[19] 1 #Code Block : Function & call to get inputs (start/end state)
     2 nodes_in_graph = ','.join(list(graph.keys()))
     3 while(1):
     4   start_node = input(f"Select a node from the available nodes: {nodes_in_graph} \n")
     5   if start_node in graph.keys():
     6     break
     7   else:
     8     print("Please select only from the available nodes")
     9 goal_node = 'H'
```

```
Select a node from the available nodes: A,B,C,D,E,F
D
```

Running Algo1

```
1 #Invoke algorithm 1 (Should Print the solution, path, cost etc., (As mentioned in the problem))
2 print("Given Graph:")
3 for k,v in graph.items():
4   print(k,v)
5 print(f"{start_node=}")
6 print(f"{goal_node=}")
7 a_star_start_time = time.time()
8 astar_path, astar_cost, astar_nodes_expanded = a_star_algo(start_node, goal_node)
9 a_star_start_end_time  = time.time()
10 print("Optimal Path found: ", astar_path)
11 print("Cost of the optimal path: ", astar_cost)
```

```
Given Graph:
A [('B', 5), ('C', 8)]
B [('A', 5), ('C', 7), ('D', 6), ('E', 10), ('H', 8)]
C [('A', 8), ('B', 7), ('F', 12)]
D [('B', 6), ('H', 10)]
E [('B', 10), ('F', 9), ('H', 18)]
F [('C', 12), ('E', 9)]
start_node='D'
goal_node='H'
open_set={'D'}
closed_set=set()
open_set={'H', 'B'}
closed_set={'D'}
open_set={'E', 'A', 'C', 'H'}
closed_set={'B', 'D'}
open_set={'E', 'C', 'H'}
closed_set={'A', 'B', 'D'}
open_set={'F', 'E', 'H'}
closed_set={'A', 'B', 'D', 'C'}
Optimal Path found:  ['D', 'H']
Cost of the optimal path:  10
```

Running Algo 2

```python
#Invoke algorithm 2 (Should Print the solution, path, cost etc., (As mentioned in the problem))
print("Given Graph:")
for k,v in graph.items():
  print(k,v)
print(f"{start_node=}")
print(f"{goal_node=}")
hill_climb_start_time = time.time()
hc_path, hc_cost, hc_nodes_expanded = hill_climbing_search(start_node, goal_node, graph)
hill_climb_end_time  = time.time()
print("Optimal Path found: ", hc_path)
print("Cost of the optimal path: ", hc_cost)
```

```
Given Graph:
A [('B', 5), ('C', 8)]
B [('A', 5), ('C', 7), ('D', 6), ('E', 10), ('H', 8)]
C [('A', 8), ('B', 7), ('F', 12)]
D [('B', 6), ('H', 10)]
E [('B', 10), ('F', 9), ('H', 18)]
F [('C', 12), ('E', 9)]
start_node='D'
goal_node='H'
neighbors=[('B', 6), ('H', 10)]
neighbors=[('A', 5), ('C', 7), ('D', 6), ('E', 10), ('H', 8)]
Optimal Path found:  ['D', 'B', 'H']
Cost of the optimal path:  14
```

(Question 4). Find and print space and time complexity using code in your implementation

A* Algorithm

Space complexity

The space complexity of the A* algorithm is **O(b^d)**, where **b** is the branching factor and **d** is the depth of the search tree. This is because the algorithm needs to store all the nodes that have been visited so far, as well as the nodes that are currently being explored.

Time complexity

The time complexity of the A* algorithm is **O(b^d),** where **b** is the branching factor and **d** is the depth of the search tree. This is because the algorithm needs to explore all possible paths from the start node to the goal node, and the number of possible paths is exponential in the depth of the search tree.

Hill Climbing Algorithm

Space complexity

The space complexity of the Hill Climbing algorithm is **O(b),** where **d** is the depth of the search tree. This is because the algorithm only needs to store the current node and its neighbors.

Time complexity

The time complexity of the Hill Climbing algorithm is **O(b^d),** where **b** is the branching factor and **d** is the depth of the search tree. This is because the algorithm needs to explore all possible neighbors of the current node in order to find the one with the lowest cost.