NAME-RAJSHREE
ROLL - BTECH/60129/23
LAB-AI/ML

Question 1.

```
// BFS
from collections import deque


graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['E'],
    'C': [],
    'D': ['G'],
    'E': [],
    'G': []
}

def bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()
    traversal = []

    while queue:
        node, path = queue.popleft()

        if node not in visited:
            visited.add(node)
            traversal.append(node)

            if node == goal:
                return path, traversal

            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))

    return None, traversal



path_bfs, order_bfs = bfs(graph, 'S', 'G')

print("BFS Traversal Order:", order_bfs)
print("BFS Path:", path_bfs)
```

Output:

```
BFS Traversal Order: ['S', 'A', 'B', 'C', 'D', 'E', 'G']
BFS Path: ['S', 'A', 'D', 'G']
```

Question 2.

```
// dfs
graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['E'],
    'C': [],
    'D': ['G'],
    'E': [],
    'G': []
}

def dfs(graph, node, goal, visited, path, traversal):
    visited.add(node)
    traversal.append(node)
    path.append(node)


    if node == goal:
        return True

    for neighbor in graph[node]:
        if neighbor not in visited:
            if dfs(graph, neighbor, goal, visited, path, traversal):
                return True


    path.pop()
    return False



visited = set()
path = []
traversal = []

dfs(graph, 'S', 'G', visited, path, traversal)

print("DFS Traversal Order:", traversal)
print("DFS Path:", path)

Output:

DFS Traversal Order: ['S', 'A', 'C', 'D', 'G']
DFS Path: ['S', 'A', 'D', 'G']

Question 3:
// best first search
import heapq


graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['E'],
```

```python
    'C': [],
    'D': ['G'],
    'E': [],
    'G': []
}


heuristic = {
    'S': 6,
    'A': 4,
    'B': 5,
    'C': 7,
    'D': 2,
    'E': 6,
    'G': 0
}

def best_first_search(graph, heuristic, start, goal):
    # Priority queue: (heuristic_value, node, path)
    pq = []
    heapq.heappush(pq, (heuristic[start], start, [start]))

    visited = set()
    traversal = []

    while pq:
        h_value, node, path = heapq.heappop(pq)

        if node not in visited:
            visited.add(node)
            traversal.append(node)

            # Goal test
            if node == goal:
                return path, traversal

            # Expand neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    heapq.heappush(
                        pq,
                        (heuristic[neighbor], neighbor, path + [neighbor])
                    )

    return None, traversal


path, traversal = best_first_search(graph, heuristic, 'S', 'G')

print("Best-First Traversal Order:", traversal)
print("Best-First Path:", path)
```

Output:

```
Best-First Traversal Order: ['S', 'A', 'D', 'G']
Best-First Path: ['S', 'A', 'D', 'G']
```

Question 4.

```
# A* Search Implementation with Traversal Order (no expanding print)

# Define the graph edges and costs
graph = {
    'S': {'A': 1, 'B': 2},
    'A': {'C': 3, 'D': 1},
    'B': {'E': 4},
    'C': {},
    'D': {'G': 2},
    'E': {},
    'G': {}
}

# Define the heuristic values h(n)
heuristic = {
    'S': 6,
    'A': 4,
    'B': 5,
    'C': 7,
    'D': 2,
    'E': 6,
    'G': 0
}

import heapq

def a_star_search(start, goal):
    open_list = []
    heapq.heappush(open_list, (heuristic[start], start, [start], 0))
    closed_list = set()
    traversal_order = []  # To keep track of node expansion order

    while open_list:
        f, current, path, g = heapq.heappop(open_list)

        if current in closed_list:
            continue

        closed_list.add(current)
        traversal_order.append(current)

        if current == goal:
            return traversal_order, path, g

        for neighbor, cost in graph[current].items():
            if neighbor not in closed_list:
                g_new = g + cost
                f_new = g_new + heuristic[neighbor]
                heapq.heappush(open_list, (f_new, neighbor, path + [neighbor],
g_new))

    return traversal_order, None, float('inf')  # If goal not reachable

# Run A* search
traversal_order, optimal_path, total_cost = a_star_search('S', 'G')

# Format optimal path with single quotes (default Python list formatting already
uses single quotes)
formatted_path = optimal_path

print("Traversal order:", traversal_order)
print("Optimal path:", formatted_path)
print("Total path cost:", total_cost)
```

Output:

```
Traversal order: ['S', 'A', 'D', 'G']
Optimal path: ['S', 'A', 'D', 'G']
Total path cost: 4
```

Question 5.

```
// hill climbing

# Hill Climbing Algorithm Implementation

# Define the graph edges (we only need connectivity to find neighbors)
graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['E'],
    'C': [],
    'D': ['G'],
    'E': [],
    'G': []
}

# Heuristic values h(n)
heuristic = {
    'S': 6,
    'A': 4,
    'B': 5,
    'C': 7,
    'D': 2,
    'E': 6,
    'G': 0
}

def hill_climbing(start, goal):
    current = start
    traversal_path = [current]
    visited_sequence = [current]
    while current != goal:
        neighbors = graph[current]
        if not neighbors:
            # No neighbors to move to
            break

        # Select neighbor with lowest heuristic
        next_node = min(neighbors, key=lambda n: heuristic[n])

        if heuristic[next_node] >= heuristic[current]:
            # No improvement, local maxima or plateau
            break

        # Move to the selected neighbor
        current = next_node
        traversal_path.append(current)
        visited_sequence.append(current)

    # Observation
    observation = ""
    if current == goal:
        observation = "Goal reached successfully."
    elif not graph[current]:
        observation = "Reached a dead end."
    else:
        observation = "Stuck at a local maximum or plateau."

    return visited_sequence, current, traversal_path, observation

# Run Hill Climbing
visited_sequence, final_state, traversal_path, observation = hill_climbing('S',
'G')

# Print results
print("Sequence of states visited:", visited_sequence)
print("Final state reached:", final_state)
print("Traversal path:", traversal_path)
print("Observation:", observation)
```

```
Output:
Sequence of states visited: ['S', 'A', 'D', 'G']
Final state reached: G
Traversal path: ['S', 'A', 'D', 'G']
Observation: Goal reached successfully.
```

Question 6.

```
import math
import random

# Graph edges
graph = {
    'S': ['A', 'B'],
    'A': ['C', 'D'],
    'B': ['E'],
    'C': [],
    'D': ['G'],
    'E': [],
    'G': []
}

# Heuristic values
heuristic = {
    'S': 6,
    'A': 4,
    'B': 5,
    'C': 7,
    'D': 2,
    'E': 6,
    'G': 0
}

def simulated_annealing(start, goal, initial_temp=10, cooling_rate=0.9,
min_temp=0.1, max_iterations=100):
    current = start
    traversal_path = [current]
    visited_sequence = [current]
    T = initial_temp
    iterations = 0

    while T > min_temp and iterations < max_iterations:
        iterations += 1
        neighbors = graph[current]
        if not neighbors:
            break  # Dead end

        # Randomly pick a neighbor to consider
        next_node = random.choice(neighbors)

        delta_h = heuristic[next_node] - heuristic[current]

        if delta_h < 0:
            # Improvement, accept move
            current = next_node
        else:
            # Worse move, accept with probability
            prob = math.exp(-delta_h / T)
            if random.random() < prob:
                current = next_node
            else:
                # Stay at current
                pass

        traversal_path.append(current)
        visited_sequence.append(current)

        if current == goal:
            break

        # Cool down temperature
        T *= cooling_rate

    # Observation about the search
    if current == goal:
        observation = "Goal reached successfully."
    elif not graph[current]:
        observation = "Reached a dead end."
    else:
        observation = "Stopped due to low temperature; may have avoided local
maxima."

    return visited_sequence, current, traversal_path, observation

# Run Simulated Annealing
visited_sequence, final_state, traversal_path, observation =
simulated_annealing('S', 'G')

# Print results
print("Sequence of states visited:", visited_sequence)
print("Final solution reached:", final_state)
print("Traversal path:", traversal_path)
print("Observation:", observation)
```

output:

```
Sequence of states visited: ['S', 'A', 'C']
Final solution reached: C
Traversal path: ['S', 'A', 'C']
Observation: Reached a dead end.
```