Experiment No. 4

1. **Greedy Search:**

Overview:

Greedy Search is a simple algorithm used for traversing or searching tree or graph data structures. At each step, it selects the node that appears to be the best choice based solely on heuristic information, without considering the overall path to that node. Greedy Search is not guaranteed to find the optimal solution, but it's relatively efficient and can be useful in certain scenarios.

Algorithm Steps:

1. **Initialization:**
   - Start with the initial or starting node.
   - Initialize an empty list to store the path.

2. **Traversal:**
   - While the goal node has not been reached and there are still unexplored nodes:
     - Select the node that appears to be the best choice based on a heuristic evaluation.
     - Move to the selected node and add it to the path.

3. **Termination:**
   - Terminate the algorithm when the goal node is reached or when there are no more nodes to explore.

Key Points:

- Greedy Search makes decisions based solely on local information, without considering the overall path.

- It's not guaranteed to find the optimal solution, but it can be efficient in certain scenarios, especially when the search space is large.
- Greedy Search may get stuck in local optima and fail to find the global optimum.

2. A* Search:**

Overview:

A* Search is an informed search algorithm used for traversing or searching tree or graph data structures. It combines the advantages of both uniform cost search (UCS) and greedy search by considering both the cost to reach a node and an estimate of the cost from that node to the goal. A* Search is widely used due to its optimality and efficiency in finding the shortest path.

Algorithm Steps:

1. **Initialization:**
    - Start with the initial or starting node.
    - Initialize an empty priority queue ordered by the sum of the cost to reach a node and the estimated cost from that node to the goal.
    - Initialize the cost to reach the starting node as zero.

2. **Traversal:**
    - While the priority queue is not empty, repeat the following steps:
        - Dequeue a node with the lowest total cost from the priority queue. This node represents the current node being explored.
        - If the dequeued node is the goal node, terminate the algorithm and return the path.
        - Visit and process the current node.
        - For each unvisited neighbouring node of the current node:
            - Calculate the total cost to reach that neighbouring node via the current node.
            - Enqueue the neighbouring node into the priority queue with its total cost.
            - Update the cost to reach the neighbouring node if a lower-cost path is found.

3. **Termination:**

- Terminate the algorithm when the goal node is reached or when the priority queue becomes empty.

Key Points:

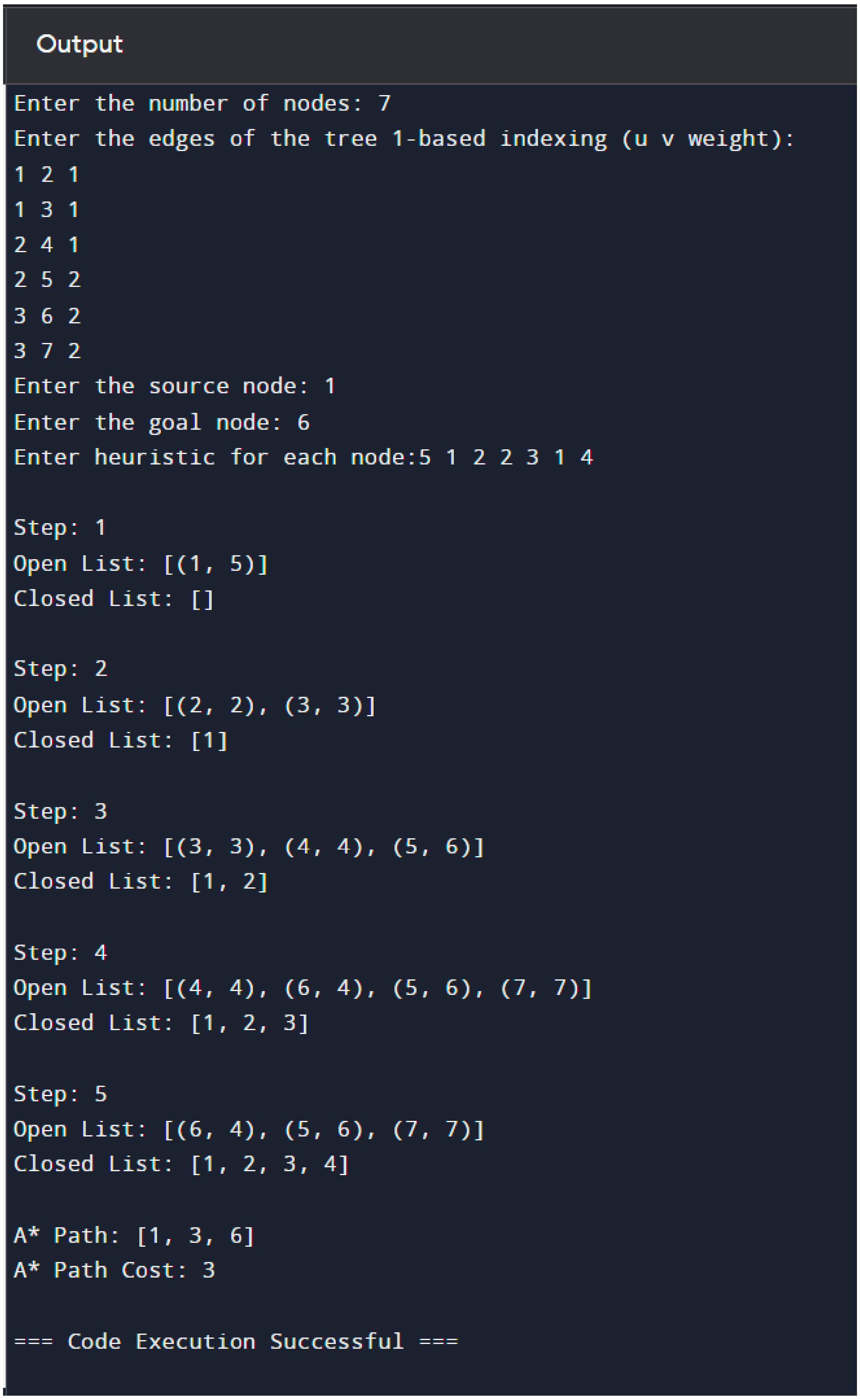
- A* Search is optimal and complete when using consistent heuristic functions.
- It considers both the cost to reach a node and an estimate of the cost from that node to the goal, ensuring efficient and effective pathfinding.
- A* Search guarantees the shortest path from the starting node to the goal node when using admissible heuristic functions.

Code:

```
def astar(tree, start, goal, heuristic):
    n = len(tree)

    open_list=[(start,heuristic[start])]
    parent = [-1] * n
    cost = [float('inf')] * n
    cost[start] = 0
    closed_list = []

    step = 1
    found=False

    while open_list:
        open_list=sorted(open_list, key=lambda x: x[1])
        current_node = open_list[0]
        current=current_node[0]
        print("\nStep:", step, "\nOpen List:",open_list, "\nClosed List:", closed_list)
        step += 1
        del open_list[0]
        closed_list.append(current)

        if current == goal:
            found=True
            break

        for neighbor, edge_cost in tree[current]:
            if neighbor in closed_list:
                continue

            cost[neighbor] = cost[current] + edge_cost
            parent[neighbor] = current
            open_list.append((neighbor,heuristic[neighbor]+cost[neighbor]))

    if not found:
        print("\nGoal node not found!!!")
        return None,None

    path = []
    while goal != -1:
        path.append(goal)
        goal = parent[goal]

    path.reverse()
    return path, cost[path[-1]]
```

```
n = int(input("Enter the number of nodes: "))

tree = [[] for _ in range(n+1)]
print("Enter the edges of the tree 1-based indexing (u v weight):")

for _ in range(n-1):
    u, v, weight = map(int, input().split())
    tree[u].append((v, weight))
    tree[v].append((u, weight))

start = int(input("Enter the source node: "))
goal = int(input("Enter the goal node: "))
if start>n:
    print('\nInvalid start node!!!')
    exit(0)

heuristic =[float('inf')]+[int(i) for i in input("Enter heuristic for each node:").split()]

path, path_cost = astar(tree, start, goal, heuristic)

if path:
    print("\nA* Path:", path)
    print("A* Path Cost:",path_cost)
```

Output:



Output

```
Enter the number of nodes: 7
Enter the edges of the tree 1-based indexing (u v weight):
1 2 1
1 3 1
2 4 1
2 5 2
3 6 2
3 7 2
Enter the source node: 1
Enter the goal node: 6
Enter heuristic for each node:5 1 2 2 3 1 4

Step: 1
Open List: [(1, 5)]
Closed List: []

Step: 2
Open List: [(2, 2), (3, 3)]
Closed List: [1]

Step: 3
Open List: [(3, 3), (4, 4), (5, 6)]
Closed List: [1, 2]

Step: 4
Open List: [(4, 4), (6, 4), (5, 6), (7, 7)]
Closed List: [1, 2, 3]

Step: 5
Open List: [(6, 4), (5, 6), (7, 7)]
Closed List: [1, 2, 3, 4]

A* Path: [1, 3, 6]
A* Path Cost: 3

=== Code Execution Successful ===
```