Dr, D. Y. Patil College of Engineering and Innovation, Varale, Talegaon

Department of Computer Engineering

Class: TE Computer A.Y. 2024-25 Sem-II

Subject: Laboratory Practice-II (Part-I-Artificial Intelligence)

Lab Assignment - 03

Objective: To implement the Greedy Search Algorithm for solving problems such as Selection Sort, Minimum Spanning Tree, Shortest Path, and Job Scheduling.

Introduction

Greedy algorithms are a way to solve problems by making the best choice at each step without worrying about the bigger picture. The idea is that choosing the best option in the moment will lead to the best overall solution. However, this approach doesn't always guarantee the best result. Greedy algorithms work well when:

- **Greedy Choice Property:** The best solution can be built by picking the best choices at each step.
- **Optimal Substructure:** The problem can be broken down into smaller problems that can be solved independently.

Applications of Greedy Algorithm

1. Selection Sort

Theory - Selection Sort follows a greedy approach by selecting the smallest element from the unsorted part and swapping it with the first element of the unsorted array. The process repeats until the array is sorted.

Algorithm

1. Start with an unsorted array.
2. Find the smallest element in the unsorted part.
3. Swap it with the first element of the unsorted part.
4. Move the boundary between sorted and unsorted parts.
5. Repeat until the entire array is sorted.

Flowchart

Start → Find min element → Swap with first → Move boundary → Repeat → Sorted list → End

Code (Python)

```
arr = [64, 25, 12, 22, 11]

for i in range(len(arr)):
    min_idx = i
    for j in range(i+1, len(arr)):
        if arr[j] < arr[min_idx]:
            min_idx = j
    arr[i], arr[min_idx] = arr[min_idx], arr[i]
print("Sorted array:", arr)
```

Time Complexity: $O(n^2)$

---

2. Minimum Spanning Tree using Kruskal's Algorithm

Theory - A Minimum Spanning Tree (MST) of a graph is a subset of edges that connects all vertices with the minimum total edge weight. Kruskal's algorithm sorts all edges in increasing order and selects the smallest edge that does not form a cycle.

Algorithm

1. Sort all edges in ascending order of weight.
2. Pick the smallest edge and check if adding it forms a cycle.
3. If no cycle is formed, add it to the MST.
4. Repeat until MST has (V-1) edges.

Flowchart

Start → Sort edges → Pick min edge → Check cycle → Add to MST → Repeat (V-1) edges → End

Code (Python)

```
from heapq import heapify, heappop

graph = [(1, 2, 1), (2, 3, 2), (1, 3, 3)]

heapify(graph)

mst = []

while graph and len(mst) < 2:
    weight, u, v = heappop(graph)
```

```
    mst.append((u, v, weight))

print("MST:", mst)
```

Time Complexity: O(E log E)

3. Minimum Spanning Tree using Prim's Algorithm

Theory - Prim's algorithm is another approach to finding the Minimum Spanning Tree (MST). It starts from an arbitrary node and grows the MST by always adding the smallest edge that connects a new vertex.

Algorithm

1. Start with any vertex and add it to the MST.
2. Select the smallest edge that connects a new vertex.
3. Repeat until all vertices are included.

Flowchart

Start → Select starting node → Find min edge → Add to MST → Repeat until all nodes are covered → End

Code (Python)

```
import heapq

def prim(graph, start):
    mst = []
    visited = set([start])
    edges = [(cost, start, v) for v, cost in graph[start]]
    heapq.heapify(edges)
    while edges:
        cost, u, v = heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            mst.append((u, v, cost))
            for next_v, next_cost in graph[v]:
                if next_v not in visited:
```

```
                heapq.heappush(edges, (next_cost, v, next_v))
    return mst
```

Time Complexity: O(E log V)

4. Dijkstra’s Algorithm (Single-Source Shortest Path)

Theory - Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It uses a priority queue to always expand the nearest unvisited node.

Algorithm

1. Assign initial distances as infinity except for the source (0).
2. Pick the smallest unvisited node and update its neighbors.
3. Repeat until all nodes are visited.

Flowchart

Start → Initialize distances → Select min distance node → Update neighbors → Mark visited → Repeat → End

Code (Python)

```
import heapq

graph = {0: [(1, 4), (2, 1)], 1: [(3, 1)], 2: [(1, 2), (3, 5)], 3: []}

dist = {0: 0, 1: float('inf'), 2: float('inf'), 3: float('inf')}

heap = [(0, 0)]

while heap:

    d, node = heapq.heappop(heap)

    for neighbor, weight in graph[node]:

        if d + weight < dist[neighbor]:

            dist[neighbor] = d + weight

            heapq.heappush(heap, (dist[neighbor], neighbor))

print("Shortest distances:", dist)
```

Time Complexity: O((V + E) log V)

---

5. Job Scheduling Problem

Theory - In the Job Scheduling Problem, each job has a deadline and a profit. The goal is to schedule jobs in such a way that maximizes profit. The greedy approach sorts jobs by profit and assigns them to the latest available time slot.

Algorithm

1. Sort jobs in decreasing order of profit.
2. Assign each job to the latest available slot before its deadline.
3. Repeat until all jobs are scheduled or no slots are left.

Flowchart

Start → Sort jobs → Check deadline → Assign slot → Maximize profit → Repeat → End

Code (Python)

```
jobs = [(3, 100), (1, 50), (2, 10)]  # (deadline, profit)

jobs.sort(key=lambda x: x[1], reverse=True)

schedule = [None] * max(job[0] for job in jobs)

for deadline, profit in jobs:

    for i in range(deadline - 1, -1, -1):

        if schedule[i] is None:

            schedule[i] = profit

            break

print("Maximized Profit:", sum(filter(None, schedule)))
```

Time Complexity: O(n log n)

Conclusion

Greedy algorithms provide efficient solutions to many optimization problems. However, they do not always guarantee the global optimal solution. By analysing different cases, we can determine when the greedy approach is suitable.