ADVANCED ALGORITHM LAB 01

NETWORK FLOW

Concept: Ford-Fulkerson Algorithm

Objective: Find the maximum flow from a source node s to a sink node t in a flow network.

Core Idea:

1. Initialize all flow to 0.
2. While there's a path from s to t (in the residual graph) with available capacity:
    - Find such a path (using DFS or BFS).
    - Determine the minimum capacity (bottleneck) on that path.
    - Augment the flow along the path by the bottleneck value.
    - Update the residual graph:
        - Subtract flow from the forward edges.
        - Add flow to the reverse edges (to allow flow cancellation if needed).

Algorithm Terminology

- **Residual Graph**: A graph that shows the remaining capacity for each edge.
- **Augmenting Path**: A path from s to t in the residual graph with available capacity.
- **Bottleneck Capacity**: Minimum capacity on that path.

Time Complexity:

- If DFS is used: O(E * max_flow) (not polynomial, but practical for small capacities).
- Use BFS (i.e., **Edmonds-Karp**) to make it polynomial: O(V * E^2)

Step-by-Step Example

Imagine a graph:

Source (S) --10--> A --4--> B --10--> Sink (T)
\ /
--5--> C --8--> D --2--/

You try finding paths from S to T, augment them, update residual capacities, and repeat until no more paths exist.

Q1: Basic Ford-Fulkerson Implementation

Graph:
Nodes: {0, 1, 2, 3, 4, 5}
Edges with capacities:
0 → 1 : 16
0 → 2 : 13
1 → 2 : 10
2 → 1 : 4
1 → 3 : 12
3 → 2 : 9
2 → 4 : 14
4 → 3 : 7
3 → 5 : 20
4 → 5 : 4

Task: Implement Ford-Fulkerson using DFS. Output the max flow from source node 0 to sink node 5.
Hint: Use adjacency dictionary, DFS path search, and residual graph logic.

Q 2: Urban Water Distribution
Scenario: Water needs to be pumped from two treatment plants to various city zones through intermediate reservoirs.

Graph Nodes:

Treatment Plants: TP1 (0), TP2 (1)
Reservoirs: R1 (2), R2 (3), R3 (4)
Zones: Z1 (5), Z2 (6), Z3 (7)

Edges (capacity in kilolitres/hour):
0 → 2 : 20
0 → 3 : 15
1 → 3 : 10
1 → 4 : 5
2 → 5 : 10
3 → 5 : 5
3 → 6 : 10
4 → 6 : 10
4 → 7 : 5

Hint: Convert to single-source and sink:

- Add super source S (8) → 0,1
- Add super sink T (9) ← 5,6,7

Tasks:

1. Implement and find max water supply/hour.
2. After removing edge 1 → 4, re-run and compare the results.
3. Which reservoirs are overloaded (i.e., receive more than one input)? Suggest how to rebalance.

Q 3 Flight Slot Scheduling

Scenario: Allocate the maximum number of flights between a set of airports, given runway limits and corridor limits.

Nodes:

Source (S = 0)
Airports: A1 (1), A2 (2), A3 (3), A4 (4)
Sink (T = 5)

Edges (flights/hour):
0 → 1 : 10
0 → 2 : 5
1 → 3 : 4
2 → 3 : 6
2 → 4 : 3
3 → 5 : 8
4 → 5 : 5

Tasks:

- Determine max number of flights scheduled/hour.
- Remove edge 2 → 4, find new max flow.
- If edge capacities are interpreted as **price per slot**, how would you modify Ford-Fulkerson to **minimize cost** instead of just max flow?

Appendix

You can refer to the code skeleton given here (optional)

```
# Ford-Fulkerson using DFS – Code Skeleton

from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(dict)  # graph[u][v] = capacity

    def add_edge(self, u, v, w):
        self.graph[u][v] = w
        if v not in self.graph or u not in self.graph[v]:  # Ensure reverse edge exists with 0 capacity
            self.graph[v][u] = 0

    def _dfs(self, s, t, visited, path):
        visited.add(s)
        if s == t:
            return True
        for neighbor in self.graph[s]:
            if neighbor not in visited and self.graph[s][neighbor] > 0:
                path[neighbor] = s
                if self._dfs(neighbor, t, visited, path):
                    return True
        return False

    def ford_fulkerson(self, source, sink):
        parent = {}
        max_flow = 0
```

```
        while True:
            visited = set()
            path = {}

            # TODO: Call DFS to check if an augmenting path exists
            found_path = self._dfs(source, sink, visited, path)
            if not found_path:
                break

            # TODO: Find bottleneck capacity (minimum residual capacity on the path)
            flow = float('inf')
            v = sink
            while v != source:
                u = path[v]
                flow = min(flow, self.graph[u][v])
                v = u

            # TODO: Update residual capacities along the path
            v = sink
            while v != source:
                u = path[v]
                self.graph[u][v] -= flow
                self.graph[v][u] += flow
                v = u

            max_flow += flow
            print(f"Augmented path flow = {flow}, Current max flow = {max_flow}")

        return max_flow


# --- Sample Usage ---
if __name__ == "__main__":
    g = Graph(6)  # example with 6 nodes (numbered 0 to 5)

    # TODO: Add your graph edges here. Example:
    # g.add_edge(0, 1, 16)
    # g.add_edge(0, 2, 13)
    # g.add_edge(1, 2, 10)
    # g.add_edge(2, 1, 4)
```

```
# g.add_edge(1, 3, 12)
# g.add_edge(3, 2, 9)
# g.add_edge(2, 4, 14)
# g.add_edge(4, 3, 7)
# g.add_edge(3, 5, 20)
# g.add_edge(4, 5, 4)

source = 0
sink = 5

print("Maximum flow is:", g.ford_fulkerson(source, sink))
```