Project Report

1. Introduction

The goal of this project was to develop a program which is able to solve the maximum flow min cost problem. The graph is extracted from a file containing its list of nodes, arcs, upper bound capacities and costs.

2. Code Structure

- **Graph Class:** This class represents the graph and contains methods to add edges, find the maximum flow, find the minimum cut, and calculate the minimum cost.
    - The __init__ method initializes the graph with the number of nodes and creates an adjacency list to store the edges and their properties.

```
class Graph:
    def __init__(self, num_nodes):
        self.num_nodes = num_nodes
        self.adj_list = defaultdict(dict)
        self.flow_values = defaultdict(int)
```

    - The add_edge method adds an edge to the graph by updating the adjacency list with the capacity and cost.

```
def add_edge(self, u, v, capacity, cost):
    self.adj_list[u][v] = {'capacity': capacity, 'cost': cost}
    self.adj_list[v][u] = {'capacity': 0, 'cost': -cost}  # Residual edge
```

    - The dijkstra method implements Dijkstra's algorithm to find the shortest path from the source node to the sink node.

```
def dijkstra(self, source, sink):
    distance = [float('inf')] * self.num_nodes
    distance[source] = 0
    prev = [-1] * self.num_nodes
    visited = [False] * self.num_nodes

    while True:
        min_distance = float('inf')
        u = -1
        for v in range(self.num_nodes):
            if not visited[v] and distance[v] < min_distance:
                min_distance = distance[v]
                u = v

        if u == -1:
            break

        visited[u] = True

        for v in self.adj_list[u]:
            edge = self.adj_list[u][v]
            if edge['capacity'] > 0 and distance[u] + edge['cost'] < distance[v]:
                distance[v] = distance[u] + edge['cost']
                prev[v] = u

    return distance, prev
```

- The augment_path method updates the flow values and capacities along the augmenting path found by Dijkstra's algorithm.

```
def augment_path(self, source, sink, prev):
    v = sink
    path_capacity = float('inf')
    while v != source:
        u = prev[v]
        edge = self.adj_list[u][v]
        path_capacity = min(path_capacity, edge['capacity'])
        v = u

    v = sink
    while v != source:
        u = prev[v]
        self.flow_values[(u, v)] += path_capacity
        self.flow_values[(v, u)] -= path_capacity
        self.adj_list[u][v]['capacity'] -= path_capacity
        self.adj_list[v][u]['capacity'] += path_capacity
        v = u
```

- The max_flow method repeatedly applies Dijkstra's algorithm and augmenting paths until no more paths can be found, returning the maximum flow value.

```
def max_flow(self, source, sink):
    while True:
        distance, prev = self.dijkstra(source, sink)
        if prev[sink] == -1:
            break
        self.augment_path(source, sink, prev)

    max_flow_value = sum(self.flow_values[(source, v)] for v in self.adj_list[source])
    return max_flow_value
```

- The find_min_cut method uses a breadth-first search to find the minimum cut in the graph, returning a list of arcs that form the cut.

```
def find_min_cut(self, source):
    visited = [False] * self.num_nodes
    queue = [source]
    visited[source] = True

    while queue:
        u = queue.pop(0)
        for v in self.adj_list[u]:
            if self.adj_list[u][v]['capacity'] > 0 and not visited[v]:
                visited[v] = True
                queue.append(v)

    min_cut = []
    for u in range(self.num_nodes):
        for v in self.adj_list[u]:
            if visited[u] and not visited[v]:
                min_cut.append((u, v))

    return min_cut
```

- The min_cost_max_flow method combines the max_flow and find_min_cut methods to calculate both the maximum flow value and the minimum cost value.

```
def min_cost_max_flow(self, source, sink):
    max_flow_value = self.max_flow(source, sink)
    min_cost_value = sum(self.flow_values[(u, v)] * self.adj_list[u][v]['cost']
                         for u in self.adj_list for v in self.adj_list[u])

    return max_flow_value, min_cost_value
```

- **Helper Function:**
  - read_graph_from_file: This function reads the graph data from a text file and creates an instance of the Graph class with the corresponding nodes, arcs, capacities, and costs.

```
def read_graph_from_file(file_path):
    with open(file_path, 'r') as file:
        num_nodes, num_arcs, source_node, sink_node = map(int, file.readline().split())
        graph = Graph(num_nodes)

        for _ in range(num_arcs):
            u, v, capacity, cost = map(int, file.readline().split())
            graph.add_edge(u, v, capacity, cost)

        # Add the (sink, source) arc if not already present
        if sink_node not in graph.adj_list[source_node]:
            graph.add_edge(sink_node, source_node, 0, 0)

    return num_nodes, source_node, sink_node, graph
```

3. Explanation of Algorithms

- **Maximum Flow Algorithm:** The function uses a modified version of Dijkstra's algorithm to find the maximum flow in the graph. It iteratively finds the shortest path from the source node to the sink node, updates the flow values and capacities along the path, and continues until no more paths can be found. The maximum flow value is the sum of the flow values leaving the source node.
- **Minimum Cut Algorithm:** The function uses a BFS (breadth-first search) starting from the source node to identify all nodes reachable from the source. It then iterates through all edges in the graph and checks if one end is reachable while the other end is not. If so, the edge is part of the minimum cut.
- **Maximum Flow Minimum Cost Algorithm:** The function combines the maximum flow and minimum cost calculations. It uses the same approach as the maximum flow algorithm but additionally calculates the minimum cost by multiplying the flow values with the corresponding edge costs.

4. Compilation

The code was written in a single Jupyter Notebook file.

1/ Change path to the file with a graph.

```
file_path = 'graph.txt'
num_nodes, source_node, sink_node, graph = read_graph_from_file(file_path)
```

File format, as requested in the project description:

7 10 0 6
0 1 16 58
0 3 13 40
1 2 5 46
1 4 10 45
2 3 5 58
2 4 8 33
3 2 10 60
3 5 15 46
4 6 25 72
5 6 6 68

The first line contains 4 numbers : numNodes numArcs sourceNode sinkNode, where numN odes is the number of nodes of the graph, numArcs is its number of arcs, sourceNode is s, the source node of the flow and sinkNode is t, the sink of the flow

Then, each line contains the description of an arc under the form: emanatingNode, terminatingNode, maxCapacity, cost. This denotes the arc (emanatingNode,terminatingNode) whose upper bound capacity is maxCapacity and whose cost is cost.

2/ Implementation of maximum flow algorithm for a graph given by a text file. The result is the max flow value and the list of flow values traversing each arc.

```
max_flow_value = graph.max_flow(source_node, sink_node)
flow_values = graph.flow_values
print("Maximum Flow:", max_flow_value)
print("Flow Values Traversing Each Arc:", flow_values)
```

```
Maximum Flow: 24
Flow Values Traversing Each Arc: defaultdict(<class 'int'>, {(5, 6): 6, (6, 5): -6, (3, 5): 6, (5, 3): -6, (0, 3): 13, (3, 0): -13, (4, 6): 18, (6, 4): -18, (1, 4): 10, (4, 1): -10, (0, 1): 11, (1, 0): -11, (2, 4): 8, (4, 2): -8, (3, 2): 7, (2, 3): -7, (1, 2): 1, (2, 1): -1, (0, 6): 0})
```

3/ Implementation of algorithm computing a minimum cut for a graph given by a text file. The result is the list of arcs that form a minimum cut.

```
min_cut = graph.find_min_cut(source_node)
print("Minimum Cut:", min_cut)
```

```
Minimum Cut: [(0, 6), (1, 4), (2, 4), (5, 6)]
```

4/ Implementation of maximum flow minimum cost algorithm for a graph given by a text file. The result is the max flow value, the minimum cost value, and the list of flow values traversing each arc.

```
max_flow_value, min_cost_value = graph.min_cost_max_flow(source_node, sink_node)
print("Maximum Flow:", max_flow_value)
print("Minimum Cost:", min_cost_value)
print("Flow Values Traversing Each Arc:", flow_values)
```

```
Maximum Flow: 24
Minimum Cost: 8636
Flow Values Traversing Each Arc: defaultdict(<class 'int'>, {(5, 6): 6, (6, 5): -6, (3, 5): 6, (5, 3): -6, (0, 3):
13, (3, 0): -13, (4, 6): 18, (6, 4): -18, (1, 4): 10, (4, 1): -10, (0, 1): 11, (1, 0): -11, (2, 4): 8, (4, 2): -8,
(3, 2): 7, (2, 3): -7, (1, 2): 1, (2, 1): -1, (0, 6): 0, (6, 0): 0})
```