**Report**

**Summary:**

**Ford-Fulkerson algorithm(my approach):** Used to find the maximum flow in a flow network. This algorithm utilizes an adjacency list representation for the graph, where each edge is represented by an object containing information such as the destination vertex, capacity, flow, and whether the edge is reversed. The program reads input from a file, constructs the graph, and iteratively performs a breadth-first search (BFS) to find augmenting paths in the residual graph. The augmenting paths are then used to update the flow along the edges, and the process continues until no more augmenting paths are found. The algorithm includes few methods for displaying the graph, flow network, and maximum flow for small graphs. The main method serves as the entry point, reading the input file specified as a command-line argument and executing the Ford-Fulkerson algorithm.

**Preflow-Push algorithm(My approach):**

Preflow-Push algorithm: for finding the maximum flow in a flow network. The algorithm maintains additional information for each vertex, such as height, excess flow, and the index of the current edge. The graph is represented using an adjacency list, and each edge includes a reference to its reverse edge. The push operation is applied to vertices with excess flow, and if no admissible edges are found, the vertex is relabeled by increasing its height.The algorithm includes few methods for displaying the graph, flow network, and maximum flow for small graphs. The main method serves as the entry point, reading the input file specified as a command-line argument and executing the Preflow-Push algorithm. The algorithm maintains the excess flow at each vertex and iteratively applies push and relabel operations until no vertices have excess flow. Finally the Max flow is calculated.

**Data structures used:**

My Ford-Fulkerson algorithm uses these two main data structures:

1. Graph Representation:

- The graph is represented using an adjacency list, which is stored in a HashMap<Integer, List<Edge>> named adjList. The key in the map represents a vertex, and the corresponding value is a list of outgoing edges from that vertex.

- The Edge class is used to represent edges in the graph. It contains the following attributes:

- to: The destination vertex of the edge.

- capacity: The maximum capacity of the edge.

- flow: The current flow through the edge.

- isReversed: A boolean indicating whether the edge is the reverse of another edge in the graph.

2. Queue and Set for BFS:

- In the getPath method, a breadth-first search (BFS) is performed to find augmenting paths in the residual graph. The algorithm uses the following data structures for BFS:

- Queue<Integer> named queue: It stores the vertices to be visited in the BFS order.

- Set<Integer> named visited: It keeps track of visited vertices to avoid revisiting them.

- Map<Integer, Float[]> named parentMap: It maintains a mapping of each vertex to its parent in the BFS tree, along with additional information about the edge leading to that vertex.

**Summary of methods:**

**Edge Class**

The Edge class represents a connection within the graph and possesses the following attributes:

- from: Denotes the source vertex of the edge.

- to: Identifies the destination vertex of the edge.

- capacity: Reflects the maximum capacity of the edge.

- flow: Indicates the current flow through the edge. This is consistently set to 0 for each backward edge.

- isReversed: A boolean flag indicating whether the edge is in a reversed state.

- revEdge: A reference pointing to the reverse edge in the graph. In the case of a forward edge, this reference directs to the corresponding backward edge and vice versa.

**Graph Class**

**Constructor (Graph(int n))**

- Initializes the Graph object with a specified number of vertices (n).

- Creates an adjacency list (adjList) using a HashMap, where each vertex maps to a list of its outgoing edges.

- Sets the sink to the number of vertices (n).

**addEdge(int from, int to, float capacity, float flow)**

- Introduces an edge to the graph with the given source (from), destination (to), capacity, and initial flow.

- Establishes both the forward and reverse edges, updating their attributes accordingly.

- Populates the adjacency list.

**getPath()**

- Utilizes Breadth-First Search (BFS) to identify a path from the source to the sink with available capacity.

- Returns a list of edges that represents the discovered path.

- Run Time: O(V + E)

- Space complexity: O(V)

**augment(List<Edge> path, float bottleNeck)**

- Updates the flow in the graph along the specified path with the given bottleneck value.

- Run Time: O(E), as the maximum length of a path is bounded by E.

**findMaxFlow(Graph graph)**

- Enacts the primary Ford-Fulkerson algorithm.

- Discovers augmenting paths with a run time of O(V + E).

- Continually updates the graph until no more augmenting paths are found.

- If the maximum flow is denoted as C, the total run time is O(C \* (V + E)), where C is a constant.

**Time and space complexity analysis of FordFulkerson algorithm:**

1. **Breadth-First Search (BFS) in getPath method:**

- In the worst case, BFS takes O(V + E) time, where V is the number of vertices and E is the number of edges.

2. **Augmentation in augment method:**

- The time complexity of the augmentation step depends on the length of the augmenting path.

- In the worst case, the length of the path is O(V).

3. **Finding Max Flow in findMaxFlow method**:

- The number of iterations (K) depends on the specific characteristics of the graph and the flow values.

- In the worst case, if the algorithm converges slowly, K could be relatively large.

The overall time complexity is O(K \* (V + E)), where K is the number of iterations, V is the number of vertices, and E is the number of edges.

**Space Complexity for FordFulkerson:**

1. Graph Representation:

- The space complexity for representing the graph is O(V + E), where V is the number of vertices and E is the number of edges.

2. BFS Queue and Sets in getPath method:

- The space complexity for BFS includes the queue (Queue<Integer>) and the visited set (Set<Integer>).

- In the worst case, the space complexity is O(V).

3. Parent Map in getPath method:

- The parent map (Map<Integer, Float[]>) stores information about the parent of each vertex during BFS.

- The space complexity is O(V).

4. Path List in getPath method:

- The list representing the augmenting path contributes to the space complexity.

- In the worst case, the length of the path is O(V).

The overall space complexity is O(V + E), where V is the number of vertices and E is the number of edges.

**PreFlow Push algorithm**

**Data Structures used :**

**Edge Class:**

Represents an edge in the graph.

**Attributes:**

to: Destination vertex of the edge.

capacity: Maximum flow that the edge can carry.

flow: Current flow through the edge.

isReversed: Indicates whether the edge is part of the residual graph.

otherEdge: Points to the corresponding edge

**Graph Class:**

Represents the graph and contains methods for graph operations.

**Attributes:**

adjList: Adjacency list using a HashMap to store a list of edges for each vertex.

vertexHeight: Array to store the height of each vertex in the graph.

vertexExcessFlow: Array to store the excess flow at each vertex.

vertexCurrentEdge: Array to keep track of the current edge being considered for each vertex.

sink: The sink vertex, representing the endpoint of the flow.

**Summary of methods:**

**Edge Class:**

Edge()

- Constructor for the Edge class.

- Initializes an edge with properties like to, capacity, flow, isReversed, and otherEdge.

**Graph Class:**

Graph(int n)

- Constructor for the Graph class.

- Initializes a graph with a specified number of vertices (n).

addEdge(int from, int to, float capacity, float flow)

- Adds an edge to the graph.

- Connects vertices from and to with a specified capacity and initial flow.

getEdgeIndex(int fromNode, int toNode, boolean isRev)

- Gets the index of a specific edge.

- Finds the index of the edge between fromNode and toNode, considering whether it's reversed (isRev).

showGraph()

- Displays the current state of the graph.

- Shows the height, excess flow, and details of each vertex and its edges.

resetExcessFlow()

- Resets excess flow in each vertex.

- Updates the excess flow for each vertex based on the current flow in the graph.

showFlow()

- Displays the flow network in the graph.

- Shows the capacity of edges in the flow network.

showMaxFlow()

- Displays the maximum flow in the graph.

- Shows the flow through edges in the graph, indicating the maximum flow achieved.

updateEdge(int from, int to, boolean isRev, float newCap, float newFlow)

- Updates the properties of a specific edge.

- Modifies the capacity and flow of the edge between from and to, considering whether it's reversed (isRev).

**preflowPush Class:**

push(Graph graph, int from)

- Implements the push operation in the preflow-push algorithm.

- Moves excess flow from the from vertex to its neighbors.

**findMaxFlow(Graph graph)**

- Finds the maximum flow in the graph using the preflow-push algorithm.

- Applies the preflow-push algorithm to determine the maximum flow in the graph.

**readFile(String fileName)**

- Reads a file to set up the initial graph state.

- Initializes the graph based on the information provided in the input file.

**main(String[] args)**

- The main function that orchestrates the preflow-push algorithm.

- Reads the input file, sets up the graph, and finds the maximum flow using the preflow-push algorithm.

**Time Complexity analysis for PreFlow Push algorithm:**

Certainly! Let's simplify the time and space complexity analysis in a more student-friendly way.

Time Complexity:

push(Graph graph, int from):

- This method looks at each outgoing edge from a vertex and adjusts the flow. It takes time proportional to the number of edges coming out from that vertex.

- Time Complexity: O(Number of Edges)

findMaxFlow(Graph graph):

- This is the main algorithm that keeps pushing flow until no more excess flow can be pushed or relabeling is required. In the worst case, it might take time proportional to the square of the number of vertices times the number of edges.

- Time Complexity: O(Number of Vertices^2 \* Number of Edges)

readFile(String fileName):

- Reads the file to create the graph. The time taken is related to the number of vertices and edges in the graph.

- Time Complexity: O(Number of Vertices + Number of Edges)

- The algorithm's time complexity is mainly affected by the number of vertices and edges. It might take more time for graphs with many vertices and edges.

**Space Complexity:**

Graph Class:

- The space needed to store the graph information, including vertices, edges, and various arrays.

- Space Complexity: O(Number of Vertices + Number of Edges)

push(Graph graph, int from):

- Requires a small constant amount of extra space, so we say it's O(1).

- Space Complexity: O(1)

findMaxFlow(Graph graph):

- Needs space to store a list of vertices with excess flow.

- Space Complexity: O(Number of Vertices)

readFile(String fileName):

- The space needed to store the graph information read from the file.

- Space Complexity: O(Number of Vertices + Number of Edges)

- The space complexity is determined by the amount of memory needed to store the graph and related information.

Q) We do not require the flow network to be acyclic (i.e., a directed graph without cycle).

What is the effect of cycles on the two algorithms,Ford-Fulkerson’s algorithm and preflow-push algorithm, respectively? Does this affect the correctness of the algorithms?

Exp:

In the context of Ford-Fulkerson's algorithm and the Preflow-Push algorithm, the presence of cycles in the flow network does not compromise the correctness of the algorithms. However, certain implementation considerations are crucial to ensure their proper functioning.

**Ford-Fulkerson Algorithm:**

The Ford-Fulkerson algorithm employs a Breadth-First Search (BFS) to find an augmented path from the source (s) to the sink (t). Since BFS always identifies the shortest path between two vertices, the existence of cycles in the network does not impact the algorithm's correctness. The key lies in maintaining proper tracking of parents and visited nodes during BFS to prevent the algorithm from getting stuck in loops. Furthermore, it is essential to note that the absence of edges with negative capacity contributes to the algorithm's robustness.

**Preflow-Push Algorithm:**

The Preflow-Push algorithm, on the other hand, selects a vertex with excess flow and performs push and relabel operations until all excess flow is appropriately handled. The algorithm prioritizes vertices with the highest height for further processing. This strategic choice of the next vertex prevents the flow from circulating in cycles. Additionally, by consistently reducing the excess of a vertex to zero before moving to the next one and always selecting the node with the maximum height as the next vertex, the algorithm effectively avoids the creation of circular paths.

In summary, both algorithms can accommodate cyclic flow networks with the proper implementation of key strategies. Ensuring the correct tracking of vertices, handling excess flow systematically, and making informed choices in vertex selection contribute to the algorithms' resilience in the presence of cycles. It is imperative to adhere to these implementation considerations for the algorithms to operate effectively and yield accurate results.