Structure of Code

1.1 Library provided on Canvas

Graph Data Structures and Helpers

1. SimpleGraph.java
2. Vertex.java
3. Edge.java
4. GraphInput.java
5. InputLib.java
6. KeyboardReader.java

Graph Generators

1. BipartiteGraph.java
2. BuildGraph.java
3. MeshGenerator.java
4. RandomGraph.java

1.2 Main

Filename - `tcss543.java`

1. This class takes an input text file as argument which contains a graph. This file is generated using the library provided on Canvas for graph generation.
2. To execute the main method run the command:

```
$ java tcss543 inputfile.txt
```

1. This main method will construct a simple graph using the input file by calling the `loadSimpleGraph()` method provided in `GraphInput.java` (from Canvas)

1. After the graph is constructed, it invokes the following three algorithms to compute max flow and measure the time taken for each:

a. Ford-Fulkerson Algorithm
b. Ford-Fulkerson Algorithm with scaling
c. Preflow Push Algorithm
d. Largest label Preflow Push Algorithm

1. After execution, this will print the max flow of the input graph and the time taken for executing each algorithm.

Key methods:

`main()` – Starting point of code execution which takes input file as argument

`GraphInput.loadSimpleGraph()` – the input file is given as input to this method and a simple graph is returned

1.3 Ford-Fulkerson (F-F) Algorithm

Ford-Fulkerson algorithm is used to compute the maximum flow of a network.

Filename – `FordFulkersonAlgorithm.java`

1. This class implements the Ford-Fulkerson algorithm to compute max flow of a graph
2. The input to this algorithm is a network graph which contains a source vertex named "s" and a sink vertex named "t"
3. The implementation can be divided into three major parts:

a. Creating the residual graph
b. Finding an augmenting path from source to sink in the residual graph, if one exists
c. Adding additional flow through the augmenting path and recalculating the residual graph

2. The above step is repeated until there is no path from source to sink in the residual graph.
3. Total time taken to compute the max flow is measured in milliseconds and printed on the console

Key methods:

`calculateMaxFlow()` – This method is the entry point into the F-F algorithm. This method takes a SimpleGraph instance as input and returns the max flow of the graph. This method initializes the flow of edge in the graph to 0. An initial residual graph is computed. A while loop checks if a path from source to sink exists by calling `hasPathFromSourceToSink()` method. If a path exists, bottleneck of the path is computed using `computeBottleneck()` method and additional flow is augmented to the network using `addFlowToAugmentedPath()`. After adding the flow, residual graph is created again and the process is repeated. A running sum of flow is maintained that gets updated each time a flow is augmented. If no further progress can be made, this flow is the max flow through the network.

`createResidualGraph()` – This method takes a graph as input and returns a residual graph. The residual graph will have the same vertices as the input graph. Each edge of the input graph is scanned and if the edge has bandwidth available for more flow i.e. (capacity – flow) > 0, then a forward edge is added to the residual graph. If the edge has a flow greater than 0, then a backedge is added (edge in reverse direction) with capacity equivalent to flow of the forward edge.

`hasPathFromSourceToSink()` – This method takes a graph (residual) as input and returns true if a path from source to sink exists. If a path exists, this method will also store a mapping of vertices to trace the path from source to sink. This method uses **Breadth-First-Search graph traversal** algorithm to find the path from source to sink.

`computeBottleneck()` – If a path from source to sink exists in the residual graph as determined by the previous method, this method uses the mapping of vertices to trace the path and find the edge that has the least available bandwidth. This bottleneck is the additional flow that can be augmented to the existing network flow. This method returns a `double` value.

`addFlowToAugmentedPath()` – Additional flow equal to the value returned by computeBottleneck() method is added to the network flow. Each edge in the augmenting path will be updated with new edge flow value.

1.4 Scaling Ford-Fulkerson Algorithm

This algorithm is just as Ford Fulkerson Algorithm. In case of Ford-Fulkerson algorithm if the bottleneck is considerably small then the number of augmentations increases which in turn increases the work to find augmentation paths.

Filename – `ScalingFordFulkerson.java`

1. This class implements the Scaling Ford-Fulkerson algorithm to compute max flow of a graph.

2. The input to this algorithm is a network graph which contains a source vertex named "s" and a sink vertex named "t".

3. There is even a helper class called `GraphUtils.java` which helps to perform several graph operations like DFS.

4. The implementation can be divided into three major parts:

    a. Calculate the value of Scaling factor D and create a residual graph with edges that can accommodate the scaling factor.

    b. Finding an augmenting path from source to sink in the residual graph, if one exists.

    c. Adding the flow by value decrementing the scaling factor by 2, through the augmenting path and recalculating the residual graph.

Key Methods in Helper class GraphUtils.java:

`markAllVerticesAsUnvisited()`: Marks all the vertices in the given graph as unvisited.

`doDFS():` Given the source, sink and the overall graph, this method finds the s-t path through which the flow can be increased by the given value delta.

Key Methods in ScalingFordFulkerson.java:

`getDelta()`: The value maxCapacity is calculated from the capacities coming from source. Delta is calculated which is the maximum 2^x value which is <= maxCapacity. This is the Scaling factor D.

`getMaxFlow():` Initially, doDFS`()` method is called for all delta values greater than zero. If the resultant st-path size > 1, then the flow is increased through the found s-t path and the max flow is calculated using method `increaseFlowThroughPath().` Then the graph is reset by calling helper class method `markAllVerticesAsUnvisited()` and next path is found which can accommodate the flow by doDFS`().` Then delta value is halved and recursively flow is calculated till delta equals to 1. Double value of flow is returned.

1.5 Preflow Push (Push-Relabel) Algorithm

Filename – `PreflowPush.java`

1. This class implements the Preflow Push algorithm to compute max flow of a graph
2. The input to this algorithm is a network graph which contains a source vertex named “s” and a sink vertex named “t”
3. The implementation can be divided into three major parts:

a. Creation of a residual graph from the given graph.

b. Initialization of preflow in the residual graph. This initializes the heights and flow of all the vertices in the residual graph.

c. Push operation that makes a flow from a node with excess flow in the residual graph to a neighboring node which is at a lower height.

d. Relabel operation to increase the height of a node with excess node having no neighboring nodes at a lower height.

2. The above steps are repeated until no more push or relabel operation can be performed. That is, the program terminates when there are no more nodes with excess flow.
3. The excess flow of the sink when the program terminates gives the max flow of the network.
4. The total time taken to compute the max flow is measured in milliseconds and printed on the console.

Key methods:

`findMaxFlow()` - This method in the PreflowPush.java class takes the SimpleGraph instance and maps the vertex and edges in the SimpleGraph to nodes and Pedges (named Pedge because of naming conflicting in GraphCode)respectively. These are required to initialize the nodes and preflows required for the Preflow Push algorithm. The initialization involves setting all node heights to zero except for source. Source node has height equal to the number of nodes in the graph. The method to compute max flow is then invoked from the method *`findMaxFlow()`*

`computeMaxFlow()` –This method computes the max flow for the graph inputted. It maintains a queue to keep track of the nodes with excess flow. A while loop ensures that the iteration continues if there is any excess flow. If there is any, the height of that node is checked with the height of neighboring nodes. If the height of current node is less than the neighboring nodes, then the current node height is set to destination node height plus one. This is relabeling operation. After this, the excess flow is pushed to neighboring node. For this flowPossible is calculated which is the max flow that can be pushed at this instance. The flowPossible value is the minimum of the excess flow of the current node and the remaining capacity of the edge. The flow values in the residual graph is updated as (1)destination node excess flow= current destination node excess flow plus flowPossible.(2) forward edge flow= current flow + flowPossible (2)backedge flow= backedge flow-flowPossible. (3)currNode exess flow= crrNode exess flow- flowPossible.

Once the residual graph is updated with the new flow values, check if the destination node or the current node has excess flow. If yes, add the respective node to the queue. While loop terminates when there are no more nodes in the queue that tracks the nodes with excess flow. The eflow of the sink when the while loop terminates gives the max flow of the network.

1.5 Largest Label Implementation of Push-Relabel Algorithm

Filename – `LargestLabelPushRelabel.java`

1. This class uses largest label implementation on Preflow Push algorithm to compute max flow of a graph.

2. The input to this algorithm is a network graph which contains a source vertex named "s" and a sink vertex named "t".
3. The implementation is divided into four major parts, and the steps are repeated until no active node is left in queue. i.e. the algorithm terminates when all the internal nodes have excess 0.

a. Initialization of preflow in the residual graph. This initializes the distance, count and flow of all the vertices in the residual graph.
b. Discharge operation chooses the adjacent node to push the excess flow, if there are no adjacent nodes then it finds the next largest label node to enqueue into the active node list. If not, it relabels the nodes using relabel operation.
c. Push operation that pushes the flow from a node with excess flow in the residual graph to the neighboring unsaturated node.
d. Relabel operation increases the distance of the active node that has no neighboring nodes with a lower distance.

4. The excess flow of the sink when the program terminates gives the max flow of the network.
5. The total time taken to compute the max flow is measured in milliseconds and printed on the console.

Key methods:

getMaxFlow() –This method takes the source and sink nodes as parameters and computes and returns the max flow for the given graph. All the required fields such as distance, excess, count, active are initialized. The adjacent vertices of all vertices are listed using *addEdge()* method. This method first gets the adjacent vertices of source s and computes the excess for each of the adjacent vertices and adds them in a queue using *Enqueue()* method. It checks if the queue is not empty, then it calls the Discharge() method to discharge the excess flow to the neighboring nodes. This iteration continues until there are no active vertices in the queue. Once the loop ends, it returns the maximum flow value (i.e. the value of excess at the sink).

Discharge() –This method takes the active vertex as a parameter. This method checks and pushes (using Push() method) the flow if the active vertex has any adjacent nodes with a positive excess value. If there are is one incident edge(to t) from the active node then it re-computes the largest distance of nodes from the sink using *Gap() method*. If there are no adjacent nodes then this method relabels the vertex using Relabel() method.

Push() – This method takes the edge to be pushed as a parameter. It computes the minimum value, between the excess and the residual capacity, that can be added to an edge. If the distance of the node is at least 1 greater than the node to be pushed and if the excess has a positive flow, then the flow is added along the edge.

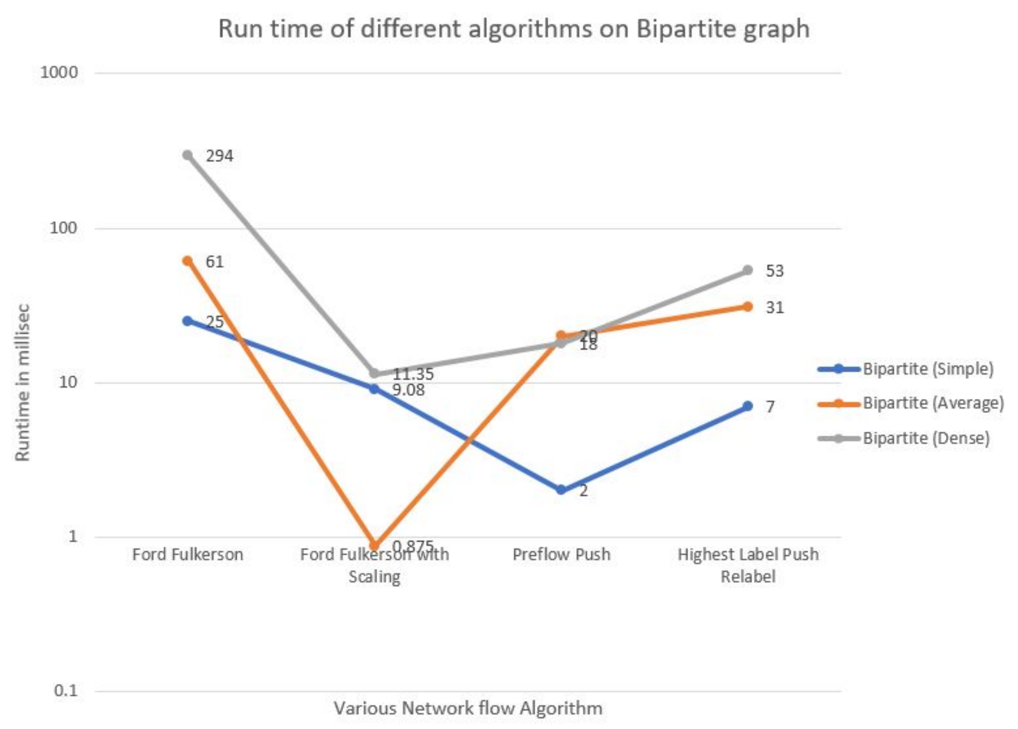
ReLabel() – This method takes the vertex, to be relabelled, as a parameter. It checks if the vertex has a positive excess and then it increases the value of distance of that node by 1.

We presented the performance plots of four max flow algorithms on each of the three instances of four input graphs by varying parameters and analyzed them.

Tests and performance evaluation

Plot of Runtime Analysis on logarithmic scale for various network flow algorithms taking Bipartite Graph as Input

Input Graphs	Maximum flow	Ford-Fulkerson	Scaling Ford-Fulkerson	Preflow Push Relabel	Largest Label Preflow Push
Bipartite (Simple)	373	25	9.08	2	7
Bipartite (Dense)	100	294	11.35	18	53
Bipartite (Average)	315	61	0.875	20	31



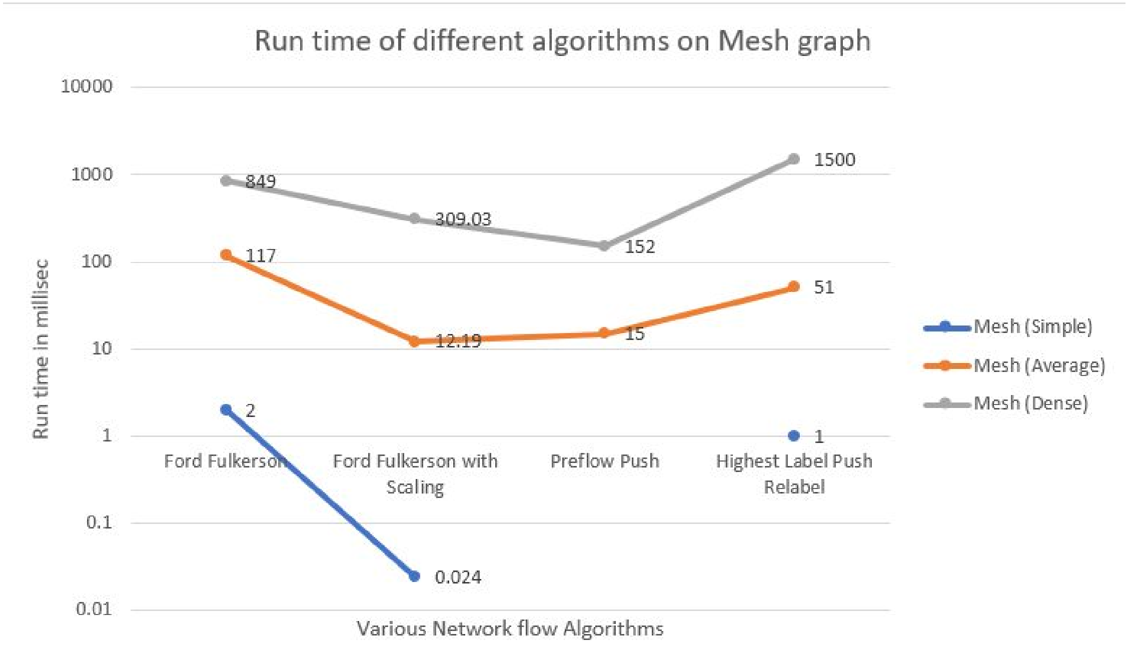
Above Graph shows the performance of four max flow algorithms on three instances of Bipartite Graph as specified below

- Simple graph: 10 source side vertex, 20 sink side vertex
- Dense: 100 source side vertex, 200 sink side vertex
- Average: 50 source side vertex, 10 sink side vertex

Plot of Runtime Analysis on logarithmic scale for various network flow algorithms taking Mesh Graph as Input

Input Graphs	Maximum flow	Ford-Fulkerson	Scaling Ford-Fulkerson	Preflow Push Relabel	Largest Label Preflow Push

Mesh(Simple)	**3**	**2**	**0.024**	**0**	**1**
Mesh (Average)	**65**	**117**	**12.19**	**15**	**51**
Mesh (Dense)	**362**	**849**	**309.03**	**152**	**1500**



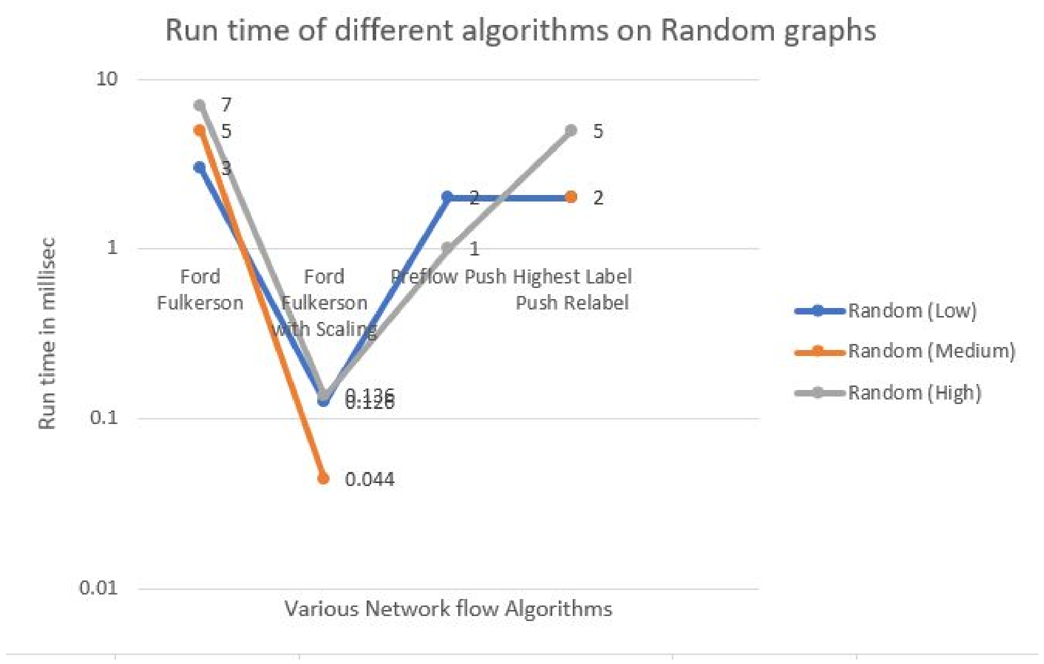
Above Graph shows the performance of four max flow algorithms on three instances of Mesh Graph as specified below

- Simple graph: 2 vertices, 2 edges, each of capacity 5
- Average: 10 vertices, 20 edges, each of capacity 20

- Dense: 50 vertices, 50 edges, each of capacity 20

Plot of Runtime Analysis on logarithmic scale for various network flow algorithms taking Random Graph as Input

Input Graphs	Maximum flow	Ford-Fulkerson	Scaling Ford-Fulkerson	Preflow Push Relabel	Largest Label Preflow Push
Random (Low)	2	3	0.126	2	2
Random (Medium)	21	5	0.044	0	2
Random (High)	49	7	0.136	1	5



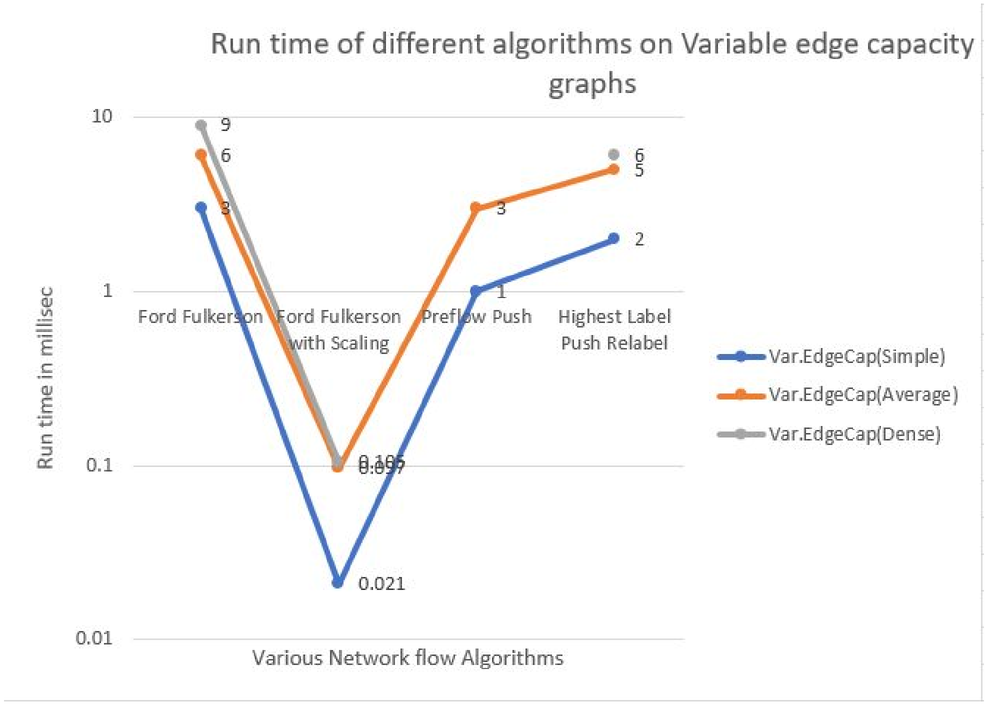
Above Graph shows the performance of four max flow algorithms on three instances of Random Graph as specified below

- Simple graph: 10 vertices, 25 edges

- Average: 10 vertices, 50 edges
- Dense: 10 vertices, 100 edges

Plot of Runtime Analysis on logarithmic scale for various network flow algorithms taking Variable Edge Capacity Graph as Input

Input Graphs	Maximum flow	Ford-Fulkerson	Scaling Ford-Fulkerson	Preflow Push Relabel	Largest Label Preflow Push
Var.EdgeCap(Simple)	1	3	0.021	1	2
Var.EdgeCap(Average)	4	6	0.097	3	5
Var.EdgeCap(Dense)	13	9	0.105	0	6



Above Graph shows the performance of four max flow algorithms on three instances of Variable Edge Capacity Graph as specified below

- Simple:10 vertex, upper bound capacity 5,lower bound capacity 1,
- Average:20 vertex, upper bound capacity 10,lower bound capacity
- Dense: 50 vertex, upper bound capacity 10,lower bound capacity 3

Division of labor

1. Chaitra SG - Analysis and implementation of Ford-Fulkerson algorithm and integration of code.
2. Tejaswi Ginuga-Analysis and implementation of Scaling Max Flow algorithm
3. Saranya Devi- Analysis and Implementation of Largest Label Implementation of Push Relabel algorithm
4. Sayana Saithu-Decided the test input graphs,analysis and implementation of Preflow Push algorithm