Empirical Study of Network Flow Algorithms

TCSS 543 : Advanced Algorithms Final Project Report

Anubha Agrawal
School of Engineering and Technology
University of Washington
Tacoma WA USA
anubha04@uw.edu

Gek Chuah Kenneth Goh
School of Engineering and Technology
University of Washington
Tacoma WA USA
ggoh@uw.edu

Adilbek Dostiyarov
School of Engineering and Technology
University of Washington
Tacoma WA USA
dostiyar@uw.edu

Sonia Xu
School of Engineering and Technology
University of Washington
Tacoma WA USA
sxu253@uw.edu

INTRODUCTION

Network flow problem is a well known type of graph problems where the input is a graph with capacities on its edges and the output varies with the specific application of a network flow. The goal of our study was to measure runtime when solving for maximum flow using three different network flow algorithms on different types of graph inputs. We measured runtimes used Ford-Fulkerson algorithm, scaling Ford-Fulkerson algorithm and the preflow-push algorithm. We created different types of graphs to test on: Fixed degree, random, mesh, and bipartite. Types of graphs, algorithms, and approaches will be discussed in the next sections of this paper. Moreover, this paper consists of a documentation of the study, the outcomes, and analysis of the test results.

METHODOLOGY

a. Classes

Before we go deeply into the description of each network flow algorithm we will introduce classes that were used for implementation:

1. *Residual Vertex* (name, used indicator, pointer to the next vertex, height, excess): This class was used to create and keep track of residual vertices in a residual graph.
2. *Residual Edge* (1st residual vertex, 2nd residual vertex, capacity): This class was used to create and keep track of edges in a residual graph.
3. *Residual Graph* (residual vertices and residual edges): This class was used to create a residual graph and contains a list of residual edges and residual vertices.

These classes are essential for the implementation of the three network flow algorithms we used in the study.

b. Ford Fulkerson:

Initially f(e) = 0 for all e in G
While there is an s-t path in the residual graph G_f
    Let P be a simple s-t path in G_f
    f'= augment (f, P)
    Update f to be f'
    Update the residual graph G_f to be $G_{f'}$
Endwhile
Return f

Fig 1: Pseudo Code : Ford-Fulkerson algorithm [1]

Given this pseudocode, we implemented this algorithm and tested it on different graphs. To implement Ford-Fulkerson, we first outlined how the algorithm works step by step.

1. Create empty residual graph
2. Find simple path using BFS
3. Update residual graph with minimum capacity

We then listed and created helper functions needed to create a residual graph. Listed here are the helper functions we created to help us find Max Flow using Ford Fulkerson:

1. *ListVertices and ListEdges* — lists vertices from input graph, lists edges from input graph
2. *sourceSinkPath* — returns path from source to sink
3. *visitedVertexUpdate* — updates which vertices have been visited
4. *getMinCapacity* — calculates min capacity for the given residual graph
5. *updateResidualGraph* — updates residual graph after each iteration given path and minimum capacity
6. *getMaxFlow* — creates a residual graph, initializes edges in graph, uses a while loop to continually find augmenting path until there is no more, returns max flow

The running time of this Ford-Fulkerson algorithm runs in O(m*C) time, where C is the sum of all the capacities of the flows and m refers to the number of edges in the graph. Comparison of runtime of all the algorithms will be discussed and compared in later section.

c. Scaling Ford-Fulkerson

Initially f(e) = 0 for all e in G
Initially set Δ to be the largest power of 2 that is no larger than the maximum capacity out of s: $\Delta \leq \max_{e \text{ out of } s} c_e$
While Δ ≥ 1
    While there is an s-t path in the graph $G_f(\Delta)$
        Let P be a simple s-t path in $G_f(\Delta)$
            f' = augment (f, P)
            Update f to be f' and update $G_f(\Delta)$
    Endwhile
    Δ = Δ / 2
Endwhile
Return f

Fig 2: Pseudo Code : Scaling Ford-Fulkerson algorithm

Given that Scaling Ford-Fulkerson is based on Ford Fulkerson algorithm we just need to make a few changes to increase the speed of finding max flow. To achieve this we created two new functions:

1. *getMaxSourceCapacity* - finds maximum capacity leaving source
2. *calculateDelta* - finds biggest power of 2 that does not exceed maximum source capacity

These are the additional functions we created to implement Scaling Ford-Fulkerson along with using all the helper functions we created to implement Ford-Fulkerson. There are also several changes in the way we insert edges to our residual graph. In comparison with Ford-Fulkerson's way of inserting edges, in Scaling approach we need to take care of edges that are not less than delta we found using *calculateDelta* function.

The running time of this Scaling Ford-Fulkerson algorithm is at most $O(m^2 \log_2 C)$ time, where C is the sum of all the capacities of the flows and m is the number of edges in the graph.

d. Preflow Push

Algorithms based on augmenting paths maintain a flow f, and use the augment procedure to increase the value of the flow.

By contrast, the Preflow-Push algorithm will increase the flow on an edge-by-edge basis and does not depend on augmenting path.

A s-t preflow is a function f that maps each edge e to a nonnegative real number, **f: E -> R+**.

A preflow must satisfy the capacity conditions:

i. For each e in E, we have $0 \leq f(e) c_e$
ii. In place of the conservation conditions we require only inequalities: e into v f(e) e out of v f(e)
iii. Ef(v) = e into vf e - e out of vf (e)

A labelling is a function h: V -> $Z_{\geq 0}$ from the nodes to the nonnegative integers.

A labelling h and an s-t preflow f are compatible if

i. (Source and sink conditions) h(t) = 0 and h(s) = n
ii. (Steepness conditions) For all edges (v, w) E_f in the residual graph, we have $h(v) \leq h(w) + 1$

If s-t preflow f is compatible with a labelling h, than there is no s-t path in the residual graph.

If s-t flow is compatible with a labelling h, then f is a flow of maximum value.

```
Initially h(v) = 0 for all v ≠ s and h(s) = n and
f(e) = c_e for all e = (s, v) and f(e) = 0 for all other edges
While there is a node v ≠ t with excess e_f(v) > 0
Let v be a node with excess
        If there is w such that push (f, h, v, w) can be
applied then
            push (f, h, v, w)
        Else
            relabel (f, h, v)
Endwhile
Return (f)

Relabel (f, h, v)
Applicable if e_f (v) > 0, and
        For all edges (v, w) E_f we have h(w)  h(v)
    Increase h(v) by 1
    Return (f, h)

Push (f, h, v, w)
        Applicable if e_f (v) > 0, h(w) < h(v) and (v, w)
E_f
    If e = (v, w) is a forward edge then
        Let δ = min(e_f(v), c_e – f(e)) and
        increase f(e) by δ
    If (v, w) is a backward edge then
        Let e = (v, w), δ = min(e_f(v), f(e)) and
        Decrease f(e) by δ
    Return (f, h)
```

Fig 3: Pseudo Code : Preflow Push algorithm

This algorithm differs from previous ones and this approach utilizes different techniques that does not depend on the capacity of edges.

General Concept:

Choose the node with maximum height/label
Choose the edge with current label to push flow
If push is possible to lower node then perform push
If push is not possible relabel the current node and move to the next vertex with maximum height
On reaching sink node, calculate excess to obtain maximum flow

Since Preflow push does not rely on a technique described in previous algorithms the helper functions differ. However, there are still similarities that are presented due to the work with residual edges and vertices:

1. *getMaxFlow()* - iterates through the residual vertices and when it reached sink node it returns max flow by calculating excess flow for that node
2. *addVertices(SimpleGraph simpleGraph)* - initializes vertices of the input graph and sets the source node at maximum height
3. *addEdges(SimpleGraph g, HashMap <String, ResidualVertex> vertexList)* - initializes edges of the input graph and sets the edge capacity
4. *getActiveNode()* - returns active node i.e. node which neither the source or sink and some excess flow
5. *findMinHeight(ResidualVertex residualVertex)* - returns edge for current vertex which has other vertex with lower height
6. *push(ResidualVertex residualVertex, ResidualEdge forwardEdge)* - pushes some of the excess flow from current node to a lower node
7. *relabel(ResidualVertex residualVertex)* - updates current vertex when no more excess could be pushed from it and relabel the vertices with higher height

You can take a look at the example in Appendix [1]

The running time of this Preflow-Push Algorithm runs in $O(n^2m)$ where n refers to the number of nodes and m refers to the number of edges.

Analysis of the Preflow-push algorithm:
Throughout the Preflow-push algorithm,

i. the labels are nonnegative integers;
ii. f is a preflow, and if the capacities are integral, then the preflow f is integral; and
iii. the preflow f and labelling h are compatible.

If the algorithm returns a preflow f, then f is a flow of maximum value.

e. Testing

We generated different types of graph inputs to experiment if we could identify a pattern with how the algorithms behave with different types of graph inputs. The following are the types of graphs we have generated:

1. *Bipartite Graphs*: These graphs are varied by capacity ranges, probabilities and the number of vertices..

2. *Fixed Degree Graphs:* These graphs are varied by capacity ranges, edge count leaving each node and the number of vertices.
3. *Mesh Generator Graphs:* These graphs are varied by capacity ranges and vertices count by varying number of rows and columns.
4. Random Graphs: These graphs are varied by capacity ranges, density and the number of vertices to get three different variations.

In total, for the study, four types of graphs with 11 different variations and five graphs for each variation were used as input. This amounted to 55 graph files being used as input.

In short, we want to evaluate the running times of the three algorithms when we vary the range of capacities, number of vertices and densities for the 4 graph types.

For undirected simple graphs, the graph density is defined as:

$$D = 2|E| / |V|(|V|-1)$$

While for directed simple graphs, the graph density is defined as:

$$D = |E| / |V|(|V|-1)$$

where |E| is the number of edges and |V| is the number of vertices in the graph.

In this study for the varying of capacities on edges for the graphs, we find that the running time of the Preflow-Push algorithm is the highest of the three algorithms. Secondly, for the varying of densities for the graphs, we have that the running times of the Ford-Fulkerson algorithm being the highest of the three, and then followed by Scaling Ford-Fulkerson and the Preflow-Push algorithm.

Lastly, for the varying of the number of vertices in the graph, we have that the running times of the Preflow-Push algorithm being the highest of the three, and then followed by the Ford-Fulkerson and finally, the Scaling Ford-Fulkerson algorithm.

BirpartiteGraph.java is used to generate a bipartite graph as users are required to specify the number of nodes on the source side, sink side, max probability, minimum capacity of flow and maximum capacity of flow. An output text file with a specified name will be generated.

RandomGraph.java is given and used to generate graphs of fixed degree.

MeshGenerator.java is given and used to generate mesh graphs.

BuildGraph.java is given and used to generate random graphs.

When no input file is given upon running the “tcss543.java” file, the program automatically generates an input text file of vertices, edges and flows. Then it proceeds to run the various algorithms on the input text file containing vertices, edges, and flows.

An object of the SimpleGraph class will be instantiated along with an instantiation of the GraphGenerate class as well.

The instantiation of the GraphGenerate class will then involve calling to the classes generating graphs with different number of vertices or nodes, different probability/density/degree and different maximum capacity.

Then the graphs are assigned to the SimpleGraph class that represents a graph.

The text file will be filled with information of the Random, Bipartite, FixedDegree and Mesh graphs accordingly.

Instructions we followed to run all three algorithms 5 times are in Appendix [2]

Explanation of the generation of graphs process:

For the Random graphs, we will vary the minimum capacities on the edges, densities of the graphs as it is determined by the number of vertices and edges, and also the number of vertices.

For the Mesh graphs, we will vary the minimum capacities on the edges, and the number of vertices.

Graph generation is done sequentially.

For the fixed degree graphs, we vary the range of capacities on the edges, change the number of edges going out of each vertex, and also change the number of vertices in the graph. For example:

- 20v-3out-4min-355max

20 vertices
3 edges out of each node
capacities range from 4 to 355
max flow: 368

- 100v-5out-25min-200max
  100 vertices
  5 edges out of each node
  capacities range from 25 to 200
  max flow: 517

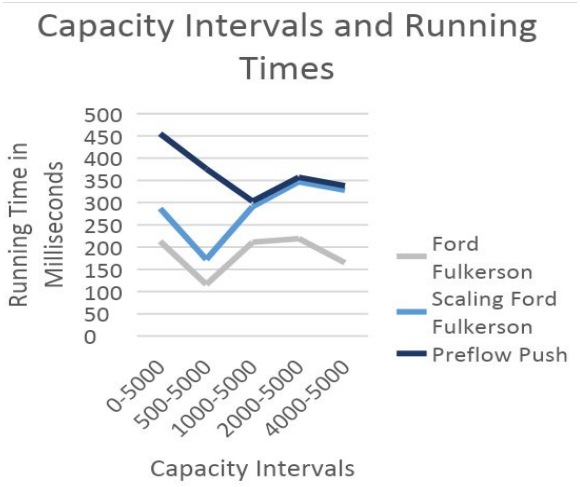
For bipartite graphs, we vary the range of capacities on the edges, change the probability of each vertex, and also change the number of vertices in the graph. For example, we have that:

We have a main java code, “tcss543.java”. In this main section Java codes, we have a throw exception case where if no input text files are input, the default cases of 55 cases belonging to the 4 different types of graphs are being tested. As the default case of 55 cases are being run in the IDE which is Eclipse in our case, the output will be generated for which we derive our results.

The results derived will be the maximum flow and running times of the various algorithms, Ford-Fulkerson, Scaling Ford-Fulkerson and Preflow-push algorithms.

Testing Capacity Range							
# of vertices	min capacity	max capacity	total capacity	max flow	Ford Fulkerson	Scaling Ford Fulkerson	Preflow Push
50	0	5000	0-5000	54024	213	74	168
50	500	5000	500-5000	45624	116	56	204
50	1000	5000	1000-5000	63286	211	80	12
50	2000	5000	2000-5000	78702	219	128	10
50	4000	5000	4000-5000	99703	165	163	10

Table 1: Results of running the three algorithms by varying the capacity ranges on edges



Graph 1: Graphs of running times of algorithms by varying capacities on edges for the graphs

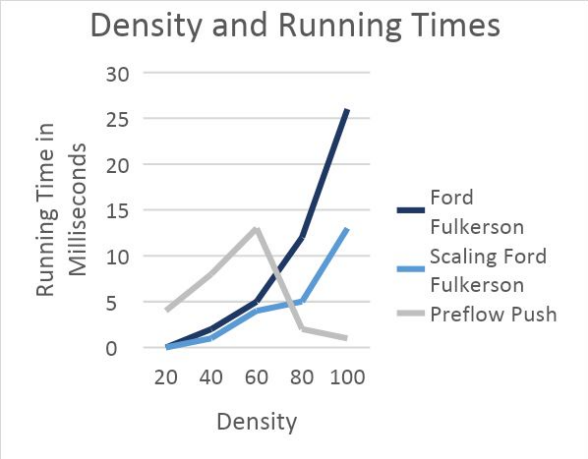
For the varying of capacities on edges for the graphs, we have that the running times of the Preflow-Push algorithm being the highest of the three.

The running time of the Ford-Fulkerson algorithm runs in O(mC) time.
The running time of this Scaling Ford-Fulkerson algorithm is at most $O(m^2 \log_2 C)$ time.
The running time of this Preflow-Push Algorithm runs in $O(n^2 m)$.

From the table above, we can see that there is a downward trend for the running time of the Preflow-Push algorithm while there is an upward trend for the other two algorithms, Ford-Fulkerson and Scaling Ford-Fulkerson.

Testing Density							
density	number of vertices	min capacity	max capacity	max flow	Ford Fulkerson	Scaling Ford Fulkerson	Preflow Push
20	25	25	125	0	0	0	4
40	25	25	125	682	2	1	8
60	25	25	125	946	5	4	13
80	25	25	125	1386	12	5	2
100	25	25	125	1765	26	13	1

Table 2: Results of running the three algorithms by varying the density of graphs



Graph 2: Graphs of running times of algorithms by varying densities of the graphs

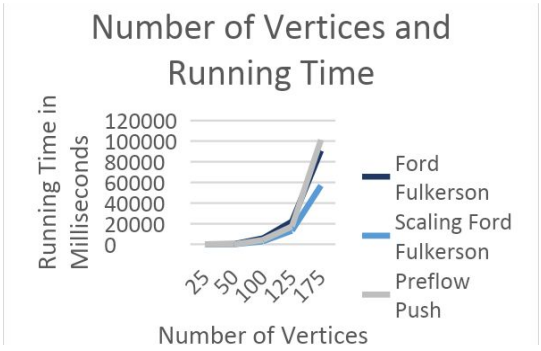
For the varying of densities for the graphs, we have that the running times of the Ford-Fulkerson algorithm being the highest of the three, and then followed by Scaling Ford-Fulkerson. We increase the densities by increasing the number of edges.

Ford-Fulkerson's runtime increases the most and we believe this could be due to the fact that this algorithm's runtime is dependent on the number of edges and sum of capacities. Therefore, an increase in density means an increase in edges, which would also increase the sum of capacities.

Testing Vertices							
density	# of vertices	min capacity	max capacity	max flow	Ford Fulkerson	Scaling FF	Preflow Push
50	25	25	125	791	3	2	9
50	50	25	125	1926	218	116	10
50	100	25	125	2540	5784	2857	4202
50	125	25	125	5497	21964	13276	17273
50	175	25	125	6742	90915	56957	101287

Table 3: Results of running the three algorithms by varying the vertices count.

From the table above, we can observe that there is an upward trend for both the running times of the Ford-Fulkerson and Scaling Ford-Fulkerson algorithms. Note that we are increasing the capacities as we increase the number of edges and therefore the density. However, the running time of the Preflow-Push algorithm does not rely on the capacities and instead, there is a downward trend for its running time.



Graph 3: Graphs of running times of algorithms by increasing the number of vertices of the graphs

For the varying of the number of vertices in the different types of graphs, we have that the running times of the Preflow-Push algorithm being the highest of the three, and then followed by the Ford-Fulkerson and finally, the Scaling Ford-Fulkerson algorithm.

From the tables above, we can observe that there is an upward trend for all the running times of the three algorithms. Note that the increasing of the running time is the most evident for the Preflow-Push algorithm since its running time is directly affected by the increasing or decreasing of the number of vertices in the graphs.

FUTURE WORK

If we have a chance to run this experiment again, we would like to automate the process of data collection and the creation of graphs so that it is no longer a manual process. Moreover, we also wanted to try running the algorithms on a larger dataset of varied graphs to see the trends in a more objective manner.

We would also want to try modifying the scaled ford-fulkerson algorithm, for example, implementing a scaling factor of 3, 5 etc. to see how much the running time changes.

The beauty of algorithms come with real world applications. Indeed, it would be nice to see how these algorithms work

for a real datasets. For example: gas transfer, people evacuation from war spots, etc. We would also be interested in applying our implementation to logistic problems.

Division of Labor:

Since the project was nicely divided into several parts it was easy to divide them:

Anubha took Fixed degree graph generation and together with Kenneth went deeply into a preflow push algorithm implementation.

Kenneth besides preflow, took responsibility for generation of mesh graphs and running the tests.

Sonia took Ford-Fulkerson implementation together with random graph generation and did a demo for running tests.

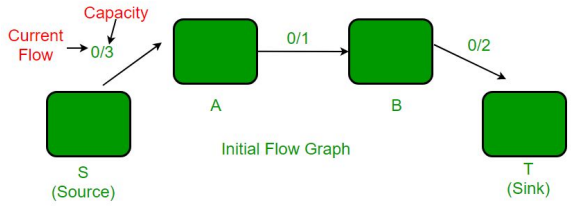
Adilbek was responsible for Scaling version of Ford-Fulkerson and Bipartite graph generation.

We all contributed to the final project slides and the report. All team members took part in editing and proofreading both the report and slides too.

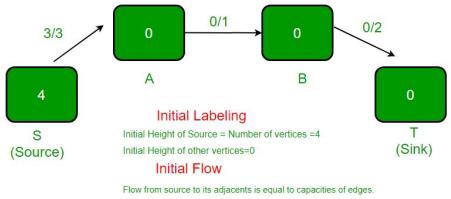
Appendix:

[1]

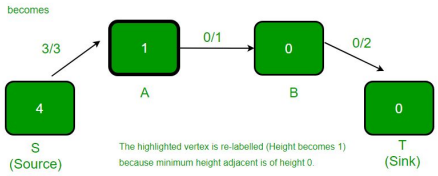
1. Initial graph:



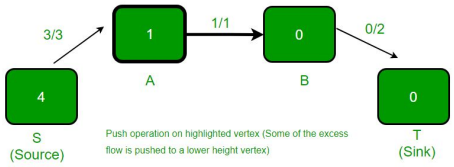
2. After Pre-Flow operation. In residual graph, there is an edge from A to S with capacity 3 and no edge from S to A.



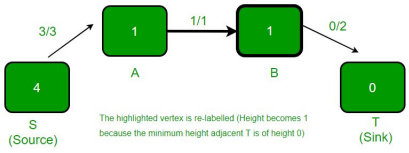
3. The highlighted vertex is relabeled (height becomes 1) as it has excess flow and there is no adjacent with smaller height. The new height is equal to minimum of heights of adjacent plus 1. In residual graph, there are two adjacent of vertex A, one is S and other is B. Height of S is 4 and height of B is 0. Minimum of these two heights is 0. We take the minimum and add 1 to it.



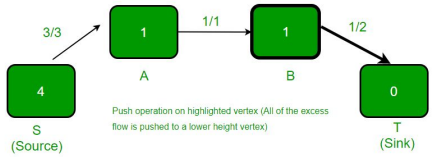
4. The highlighted vertex has excess flow and there is an adjacent with lower height, so push() happens. Excess flow of vertex A is 2 and capacity of edge (A, B) is 1. Therefore, the amount of pushed flow is 1 (minimum of two values).



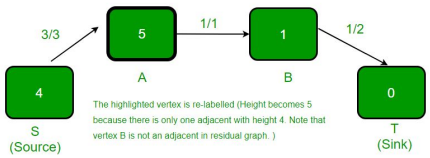
5. The highlighted vertex is relabeled (height becomes 1) as it has excess flow and there is no adjacent with smaller height.



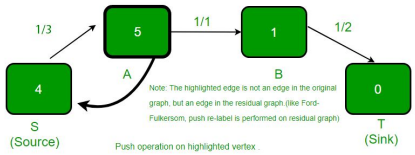
6. The highlighted vertex has excess flow and there is an adjacent with lower height, so flow() is pushed from B to T.



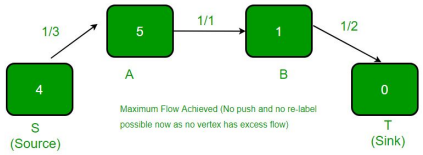
7. The highlighted vertex is relabeled (height becomes 5) as it has excess flow and there is no adjacent with smaller height.



8. The highlighted vertex has excess flow and there is an adjacent with lower height, so push() happens.



9. The highlighted vertex is relabeled (height is increased by 1) as it has excess flow and there is no adjacent with smaller height.



[2]**Instructions:**

1. We start by instantiating an instance of the “tcss543” class.
2. We also have to instance an instance of the “SimpleGraph” class.
3. We instantiate an instance of the algorithm, “FordFulkerson” class.
4. We instantiate an instance of the algorithm, “ScalingFordFulkerson” class.
5. We instantiate an instance of the algorithm, “PreflowPush” class.
6. “Run” is a class used to input the graph file to test the three algorithms.
7. The “runAlgorithm” class is used to run each of the different algorithm for 5 times since we are generating 5 different test cases for each different variable of the different types of graph we are testing.

LESSONS LEARNED FROM OTHER GROUPS

Team 1: Richard/ Harnidh/ Fares/Jingru

We learned that from team 1 that confirms our implementation of the Scaling Ford-Fulkerson algorithm, we can generally reuse most of the code from the Ford-Fulkerson algorithm, except that we need to add a function to compute delta.

Team 3: Michael/ Danielle/ Grant/ Emma

We learned from team 3 that further confirms our understanding that the Scaling Ford-Fulkerson algorithm introduces an additional scaling parameter. If there are no s-t path, we typically decrease the scaling parameter delta.

For the Preflow-Push algorithm, we also learned from team 3 that confirms our understanding that flow flows ‘downhill’.

Team 5: Jyotil/ Tejashri/ Bharti

We learned that for Ford-Fulkerson algorithm, we can sum up as it uses augmented path to calculate maximum flows and it obeys the conservation of flows.

We learned that they also used BFS to find the shortest augmenting path from source to sink in residual graph.

For Scaling Ford-Fulkerson algorithm, we learned that they also used BFS to find the shortest augmenting path from source to sink in residual graph.

For Preflow-Push algorithm, we learned that we can sum up as it does not use the augmenting path procedure. It does not obey the conservation conditions for flow. And most importantly, vertex can have more incoming flow than outgoing flow. In short, when there is no node with excess flow, or the queue is empty, the algorithm terminates.

We learned that we could also present our varying parameters for each type of graphs more clearly.

Team 8: Sitong Liu/ Jingyuan Sun/ Yankun Sun/ Samar Abu Arafah

We learned from Team 8 that for Scaling Ford-Fulkerson algorithm, we can also illustrate the key idea that it prefers heavier flows.

For Preflow-Push algorithm, we also learn that we can implement this algorithm using the data structure, “Vertex Queue”.

Team 6: Guangzheng Wu/ Zeng Fu/ Lu Han/ Ranying Cai

We observed from this team that Preflow-Push algorithm works the best for each type of graph.

For both Ford-Fulkerson and Scaling Ford-Fulkerson algorithms, we are reminded that we can also use DFS to find the shortest augmenting path from source to sink in residual graph.

For the Preflow-Push algorithm, we learned from them that we can sum it up as that the algorithm will end when we cannot update the preflow. Also, for the “update process”, get every node’s excess value. If it is greater than 0, we need to add a number to the flow of the edge which is connected to this node. Here, the increase is min ($e_f(v)$, f(e)), according to the textbook.

Team: Deepthi/ Thuan Lam/ Asmita Singla

We learned from this team that in the starting of the Ford-Fulkerson algorithm, we first initialize the flow to 0. While there is an augmenting path P, we keep augmenting flow along P.

In Scaling Ford-Fulkerson algorithm, we learned from them again that DFS is possible to be used here.

Most importantly, their results are that Scaling Ford-Fulkerson algorithm took the shortest time for the Bipartite graphs. For Mesh graphs, Scaling Ford-Fulkerson algorithm also took the shortest time. However, for random graphs, we learned from them that both Preflow-Push and Scaling Ford-Fulkerson algorithms took the shortest time.

Lastly, for fixed degree graphs, Scaling Ford-Fulkerson algorithm took the shortest time.

Team: Bowei, Hanfei, Feifei, Tong Wu

We learned from Tong Wu that the goal of the Preflow-Push algorithm is to initialize a preflow and convert it into a flow in order to find the maximum flow.

We also learned from her that their implementation of the Preflow-Push algorithm consists of Relabel operations in $O(n^2)$ and Push operations in $O(n^3)$, thus their overall time efficiency is $O(n^3)$.

ACKNOWLEDGMENTS

We will like to thank Professor Ka Yee Yeung for her valuable feedback and guidance in this project.

REFERENCES

1. Jon Kleinberg, Eva Tardos. Algorithms Design. 2006 by Pearson Education, Inc.
2. https://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/