Empirical Study on Network Flow*

Term Project: Advanced algorithms (TCSS543)

University of Washington

Misba Momin
misbam@uw.edu

Bharathi Manoharan
bmano89@uw.edu

Rituja Dange
ritujad@uw.edu

Spoorthy Balasubrahmanya
spb93@uw.edu

ABSTRACT

The goal of this project is to conduct an empirical study on network flow algorithms such as Ford Fulkerson, Scaling Ford Fulkerson and the preflow - push algorithm and tested with various inputs such as Random graph, Bipartite graph, Fixed Degree graph and Mesh graph keeping constant as number of nodes and edges, capacities and probability with respect to average running time for the above three mentioned algorithms. The results of this experimental help understand and analyze which algorithm performs better for the given input graph.

ALGORITHMS

• Ford Fulkerson • Scaling Ford Fulkerson • The preflow-push

KEYWORDS

Average Running Time, Augmenting path, Bottleneck, Residual Graph

1 INTRODUCTION

A detailed experimental study performed on the following algorithms they are Ford Fulkerson, Scaling Ford Fulkerson and the preflow-push algorithm. Implemented all the above three algorithms to find the running time for ten iterations and calculated the average running time. The obtained results are then plotted on graphs by taking number of nodes and edges or capacity in x axis with respect to average running time on the y axis to analyse which algorithm solely perform better for a specific input graphs.

2 DESIGN AND ANALYSIS OF ALGORITHMS

2.1 Ford Fulkerson

The Ford Fulkerson algorithm is an algorithm that compute maximum flow in the flow network. The idea behind this algorithm is very simple as there is a path from the source to the sink with available capacity on all edges in the path, we can send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path. The flow in a network flow is denoted by f(e) and the value f (e) represents the amount of flow carried by edge e in any flow network and C_e is the capacity of the edge in any flow network. A flow f must satisfy the following two properties. First is the (Capacity conditions) For each $e \in E$, we have $0 \leq f(e) \leq Ce$ and second is the (Conservation conditions) For each node v other than s and t, balance constraint is there. Balance constraint means that whatever is flowing in the node should come out of that node. node other than the source and the sink, the amount of flow entering must equal the amount of flow leaving. The source has no entering edges, but it can have flow going out; in other words, it can generate flow. Symmetrically, the sink can have flow coming in, even though it has no edges leaving it. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson. The idea of Edmonds-karp is to use BFS in Ford Fulkerson Algorithm as BFS always takes path with minimum number of edges.

2.2 Scaling Ford Fulkerson

In this algorithm, the maximum flow can be calculated without having to worry about selecting the path that has exactly the largest bottleneck capacity. Instead we will maintain a scaling parameter called delta and we will select the path that have the capacity of at least delta.

Step1: Initially set the flow to 0
Step2: set delta value to be less than the maximum capacity out of s
Step3: while delta>=1
While there is an s-t path in Gf (delta)
P be a simple s-t path
f' = augment (f, P)
Update f to f' and Gf(delta)
Endwhile
delta =delta/2
Endwhile

Return f
Step 1. Input graph preparation using 4 different categories by varying nodes edges and capacity.
Step 2. Obtain Vertices and Edges for the input graph using starter code for Simple Graph
Step 3. Design: Build original graph and residual graph for implementing network flow using adjacency List
Step 4. Calculate the starter value for delta (largest power of 2 less than max capacity flowing out of s).
Step 5 For a given delta, perform Breadth First Search to find, augmenting paths between s-t which have a bottleneck of at least delta.
Step 6. Calculate the bottleneck for these paths and increase the max flow by bottleneck.
Step 7 For each augmenting path, update the original graph and residual graphs follows. For forward edge of the augmenting path in original graph, increase the flow by bottleneck. For forward edge of the augmenting path in original graph, increase the flow by bottleneck.
For backward edge of the augmenting path in residual graph, decrease the flow by bottleneck.
Step 8. Update the delta and repeat the steps 3-8 until delta>=1
Step 9. Report the Maximum flow.

2.3 Preflow-push Algorithm

In push and relabel algorithm, for a given graph, each vertex has height and excess flow and each edge has flow and capacity. Excess flow will be pushed to next node if the height of next node is less than the height of current node and the edge between two nodes is not saturated. If the height of the current node is less than the height of the next node, then the height of the current node will be relabeled and again will try to perform push operations. Algorithm will run till no node except sink has excess flow. Finally, it will return max flow which is nothing but excess flow at the sink of graph.
The generic algorithm has a strongly polynomial O(V2E) time complexity, where V = number of vertices and E = number of edges of a graph G.

3 IMPLEMENTATION OF THE ALGORITHMS

3.1 Ford Fulkerson Algorithm

The Ford Fulkerson algorithm is divided primarily in 3 Parts
1.Creating a residual graph from the simple graph.
2.To find the augmenting path using BFS in residual graph and add that in original graph
3.Report the maximum flow
Obtain vertices and edges for the input graph using starter code simplegraph. A residual graph is computed from the simplegraph.
Map<vertex, Edge> nodetoparent =New HashMap<> ()

3.2 Scaling Ford Fulkerson Algorithm

Implemented Scaling Ford Fulkerson using Adjacency list
Adjacency list representation:
HashMap<String, LinkedList<Edge>> original Graph
HashMap<String, LinkedList<Edge>> residual Graph
The Key is vertex label (String) and value LinkedList of edges which are incident on this vertex.

3.3 Preflow-Push Algorithm

Data structure: Queue (linked list), HashMap, ArrayList.
Queue (linked list): Queue is used to store nodes with excess flow for a given graph.
Structure of algorithm:
Push and relabel algorithm is divided into 3 parts.

1. Initialization
2. Push and relabel
3. Termination

Initialization
In push and relabel algorithm each node contains height and excess flow and each edge contains flow and capacity. The algorithm initializes height and excess flow as zero for all nodes except source node. Height of source node is equal to number of nodes.
Excess flow for each edge will be initialized as a zero except the excess flow of edges connected to source. Those edges will get saturated and flow of saturated edges will be equal capacity.
Push and Relabel:
The algorithm uses queue. Nodes with excess flow are added to the queue. Since queue is First in First out data structure, each node which is added first in the queue dequeues the first vertex which has excessFlow form the queue and executes push operations if following conditions are satisfied.

a. If height of the current node is greater than height of next node and
b. The edge is not saturated.

The node to which flow was pushed will be added to the queue.

Else If the vertex has excessFlow even after executing all possible push operations, it is relabeled and added back to the queue. Relabel operation increase height of node by one.

Terminating condition:

When there is no excess flow on any node except sink or if there is no vertex in queue then algorithm will terminate. Finally, it will return max flow which will be equal to excess flow at sink.

The loop dequeues the first vertex which has excessFlow form the relabelVertexQueue and executes push operations if possible

4 EMPERICAL STUDY AND RESULTS

4.1 Bipartite Graph

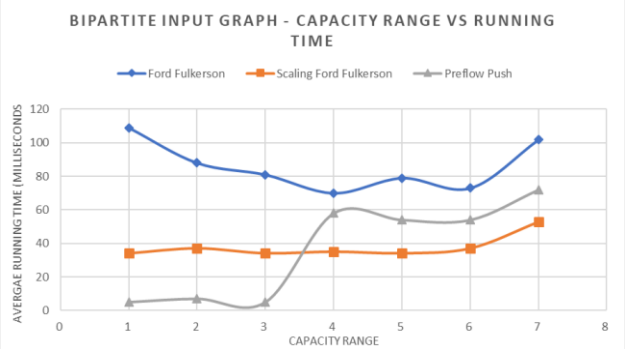

Figure 1: Bipartite graph –Capacity on X axis and Average Running Time on Y axis

Test 1: Figure 1 contains the data for the various tests that we have done by giving bipartite graph as input. In the first trial the capacity is being varied in the range of 0 to 250 ,50-250, 100-250, 200-250, 500-1000, 500-2000 and 500-5000. During experimenting with various capacity range, we noticed that the Preflow push algorithm performs better than Scaling Ford Fulkerson algorithm and Ford Fulkerson algorithm. In figure 1, it is noticeable that preflow push running time becomes larger when the capacity range was given between 200 and 250. That was the time preflow push algorithm started performing worse than Ford Fulkerson. As the capacity range increased from 500 to 1000 the scaling ford Fulkerson performance stabilized and performs better than Ford Fulkerson and preflow push. For the above input data given in table 1, the scaling max flow works best in terms of running time and it is more efficient. From this we noticed that, for the given bipartite graph, capacity is the constraint that matters the most and even though there are some other cases where preflow push algorithm performs better but when it comes to bipartite graph with varying capacities, the Scaling Ford Fulkerson performs the best.

Test 2: Figure 2 contains the data for the various tests that we have done by giving bipartite graph as input. In the first trial the number of nodes increased from 20 through 200. During experimenting with varying number of nodes in x axis, we did notice that the scaling ford Fulkerson and preflow push initially performed the same until the number of nodes increased till 100, however when the number of nodes are given between the range of 100 and 150, preflow-push performed better. And eventually when the number of nodes increased between the range of 150 and 200, the scaling ford Fulkerson performs better than preflow-push and Ford Fulkerson algorithm. All in all, we noticed that for the given inputs, in most of the cases, Scaling Ford Fulkerson algorithm preforms efficiently.

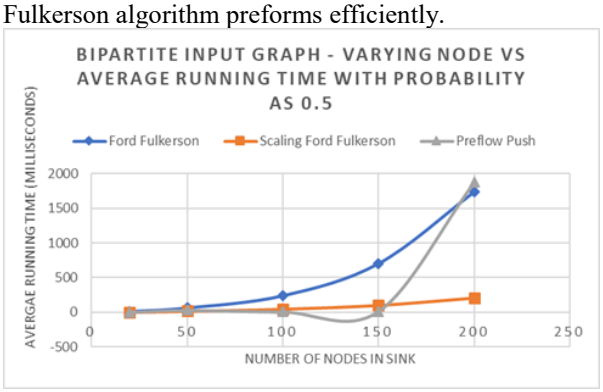

Figure 2: Bipartite graph – Number of nodes on X axis and Average Running Time on Y axis with Probability 0.5 with max_capacity = 50 and max_capacity = 1500

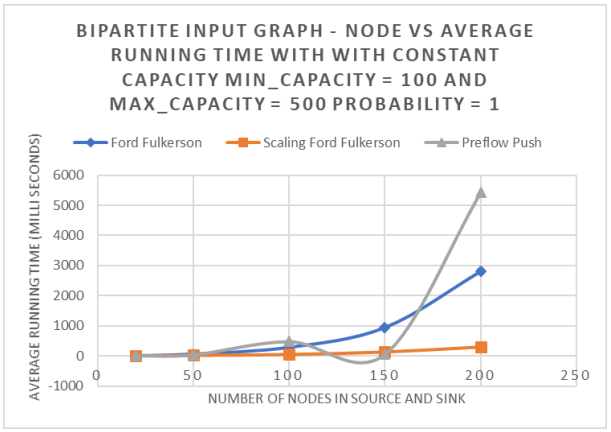

Figure 3: Bipartite graph – Number of nodes on X axis and Average Running Time on Y axis with Probability 1.0 with constant min capacity =100 and max capacity= 500

Test 3: Figure 3 contains the data for various inputs that we tested by varying the number of vertices. Referring Figure 3, we can notice that for the small number of vertices, we initially found out that all the three algorithms performed the same up to the number of vertices increased to 50. Once the number of vertices increased from 50 through 200 we noticed that the running time of the preflow push is maximum when we took 100 vertices. And again, preflow push performed better for the vertices ranges between 120 and 150, however the preflow push did not perform well

with the large scale of vertices. We could notice that the preflow push algorithm depends on the factor of number vertices taken in the input graph.

4.2 Summary of the test with bipartite graph as input

By giving bipartite graphs as input, we notice that by varying number of nodes, the scaling ford Fulkerson algorithm performs better. And as we vary the number of nodes from low range to high range the scaling performs better. The observations that I have made when experimenting with different inputs. The scaling ford Fulkerson is performing better for the bipartite input graphs and the reasons could be the way the algorithm is implemented. I have implemented the scaling ford Fulkerson algorithm by using adjacency list which help improve the running time of the algorithm. The second reason is that the tests that we have done for the bipartite graph was done on the following machine with specification as: MacBook Pro, Processor 2.7 GHz Intel Core i5. We have noticed that change in the results when we performed tests on different machines with different configuration. This could also be one of the reasons why scaling performed better for the bipartite graph. The third reason is that, for the preflow push to have better running time, it has a capacity constraint. So, when there is a varied capacity given as an input, the preflow tend to perform worse than ford Fulkerson and Scaling ford Fulkerson algorithm.

4.3 Random Graph

A random graph in which properties such as vertices, graph edges with maximum and minimum capacity and dense and connection between them is determined by random way. The random graph term refers to the probability distribution over the graph. Random graph can be described by either probability distribution or random process which generates them. In our graph code, random graph has four properties and that are vertices, dense, min capacity and max capacity. We varied one parameter at a time, keeping all other parameter as a constant generated the input graph for four varying factors of random graph. We used 10 iterations for calculating the average running time. We plot the four-output graph for varying factor versus average running time for all 3 algorithms.

Test 1: The figure 4 represent the Graph for varying vertices range vs average running time. Figure 4 contains the data for the various tests that we have done by giving random graph as input. In this testing, we consider other parameter as constant throughout the process and keep varying the vertices from 50 to 250 at a step size of 50. The constant parameters are (dense=100, min capacity=250 and max capacity=500) While doing experiments with the number for vertices in the increasing order, we noticed that the Scaling Ford Fulkerson algorithm performs better than Preflow-push algorithm and Ford Fulkerson algorithm. Also, with varying number of vertices in x axis, we did notice that the scaling ford Fulkerson and Preflow push initially performed the same until the number of nodes increased till 100, however when the number of vertices increased to 150 from 100, scaling ford Fulkerson performed better**.** This is clearly noticeable from graph that as we are increasing number of vertices, the running time is also increasing. The running time is more for all vertices when the number of vertices are maximum (i.e. highest for this graph). There are some other cases where Preflow push algorithm performs better but when it comes to random graph with varying vertices, the Scaling Ford Fulkerson performs the best as compared to other algorithms

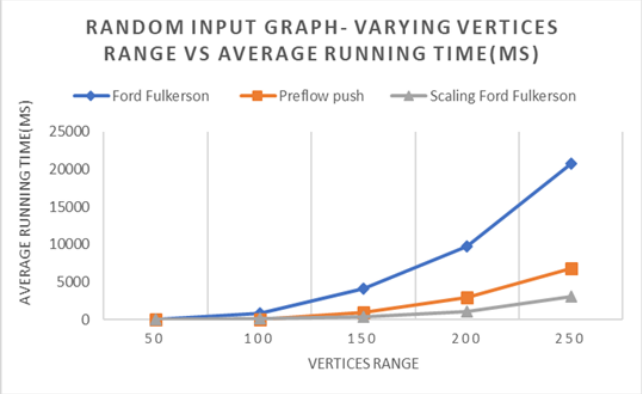

Figure 4: Random Input Graph- Varying Vertices range on X axis and average running time on Y axis

Test 2: The figure 5 represent the Graph for varying dense range vs average running time. Graph 4 contains the data for the various tests that we have done by giving random graph as input. In this testing, we consider other parameter as constant throughout the process and keep varying the dense from 50 to 250 at a step size of 50. The constant parameters are (dense=100, min capacity=250 and max capacity=500). While performing experiments, noticed that scaling ford Fulkerson perform better as compared to Preflow push and ford Fulkerson algorithm. For scaling ford Fulkerson average running time is almost linearly increasing for dense after 50. For Preflow push, the running time is more for dense=50 then it reduced for other 2 inputs with different dense value and finally it again increased. The ford Fulkerson algorithm is performing worst as compared to other two algorithms. There are some other cases where Preflow push algorithm performs better but when it comes to random graph with varying dense, the Scaling Ford Fulkerson performs the best as compared to other algorithms.

Test 3: The figure 6 represent the Graph for varying minimum capacity range vs average running time. Graph 6

contains the data for the various tests that we have done by giving random graph as input. In this testing, we consider other parameter as constant throughout the process and keep varying the min capacity from 50 to 250 at a step size of 50. The constant parameters are (dense=30, vertices=30

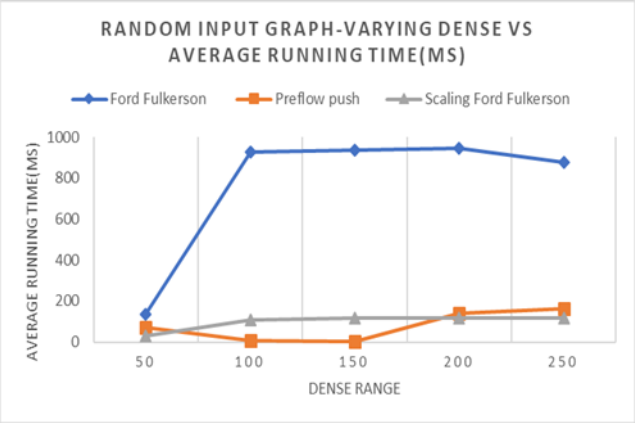

Figure 5: Random Input Graph- Varying Dense range on X axis and average running time on Y axis

and max capacity=250). For random input graph with varying min capacity, scaling ford Fulkerson perform better as compared to Preflow push and ford Fulkerson algorithm. Initially when capacity was 50, scaling ford Fulkerson and Preflow push algorithm took same running time. When capacity increased to 100, Preflow push took less time than scaling ford Fulkerson but when capacity went beyond 100 clearly Preflow push took more time than scaling ford Fulkerson. The average running time for Preflow push algorithm suddenly shoot up when capacity is 150.It's hard to determine why this shoot up happened. The ford Fulkerson algorithm is worst performer.

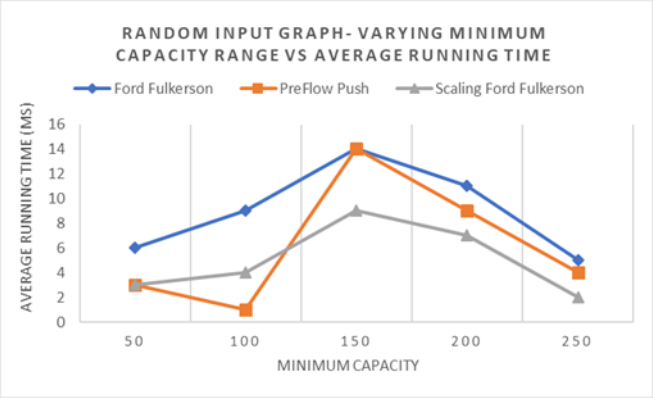

Figure 6: Random Input Graph- Varying Min Capacity range on X axis and average running time on Y axis

4.4 Summary of the test with Random Graph as input

By giving random graphs as input, we notice that by varying number of vertices, the scaling ford Fulkerson algorithm performs better. But, with varying number of vertices in x axis, we did notice that the scaling ford Fulkerson and Preflow push initially performed the same until the number of nodes increased till 100, however when the number of vertices increased to 150 from 100, scaling ford Fulkerson performed better**.** For all experiments, scaling ford Fulkerson was performing better**.** The scaling ford Fulkerson is performing better for the random input graphs and the reasons could be the way the algorithm is implemented. Another reason might be the machine used for testing of random graph. The machine was having specification as: Intel® Core™ i7-4510U CPU @2.00GHz 2.60GHz. The third reason is that, for the Preflow push to have better running time, it has a capacity constraint. So, when there is a varied capacity given as an input, the Preflow tend to perform worse than ford Fulkerson and Scaling ford Fulkerson algorithm.

4.5 Fixed Degree Graph

In Fixed degree graph, the parameters that we can vary are: the range of capacities on the edges, number of vertices and number of edges. We varied the capacity range to see the difference in running time between Ford-Fulkerson and Scaling Ford-Fulkerson. This is because we know theoretically that the running time of Ford-Fulkerson is O(m C) and running time of scaling Ford-Fulkerson is O(m^2 log C) where C is total capacity on all edges coming out of source. Based on this understanding we expect the running time of Ford-Fulkerson algorithm to be more when compared to Scaling Ford-Fulkerson.

Test1: We started with varying the capacity range starting from 0-50 and then gradually increased the capacity range as follows: 0-100, 0-250, 0-500, 0-1000, 0-1500 and 0-2000. So, the capacities on edges will have any value ranging from 0 till 50 in first dataset. Likewise, capacity value on edges in the respective datasets will range from its min value and goes up to its maximum value. Number of vertices and edges were kept constant and the values are 30 and 25 respectively. For all the input files generated, we did 10 iterations to find the running time and max flow returned by all the three algorithms were same. The basic idea behind doing 10

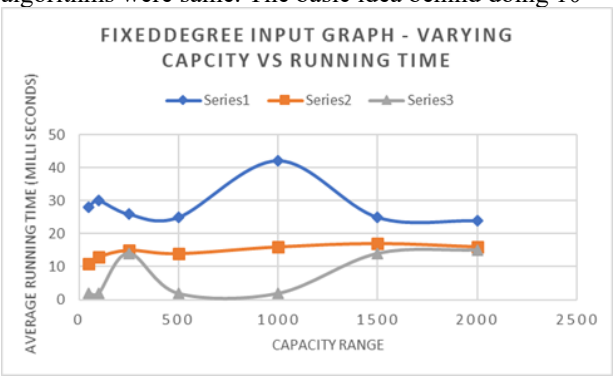

Figure 7: Fixed Degree Graph: Varying capacities Vs Average Running time

iterations were to make sure that the results are consistent. We plotted the results in graph with capacity range as X-axis and Average running time as Y-axis (Figure 7).

We observed that the running time of Scaling Ford-Fulkerson is less in comparison to Ford-Fulkerson as expected (Figure 7). Also, the running time of Pre-Flow push algorithm does not depend on capacity and it is reflected in results in Figure 7. We could see increase in running time for pre-flow push for the first few data and in the end which could be because of more relabelling operations for those datasets because of the randomness on the distribution of capacity values.

Test 2: For Fixed Degree graph, then we varied the number of vertices and edges keeping the capacity on edges as constant (minimum capacity – 10 and maximum capacity = 100). In Fixed degree, number of edges cannot be equal to number of the vertices. So, we took number of edges as equal to (number of vertices – 1) to keep the number of vertices almost same to edges so that we can plot in a graph using the same scale. The number of vertices: edges taken for this are: 10:9, 50:49, 75:74, 100:99, 125:124, 150:149, 200:199, 250:249. The results for this set of experiment (10 iterations) are plotted in Figure 8 with number of vertices/edges in X-axis and Average running time in Y-axis.

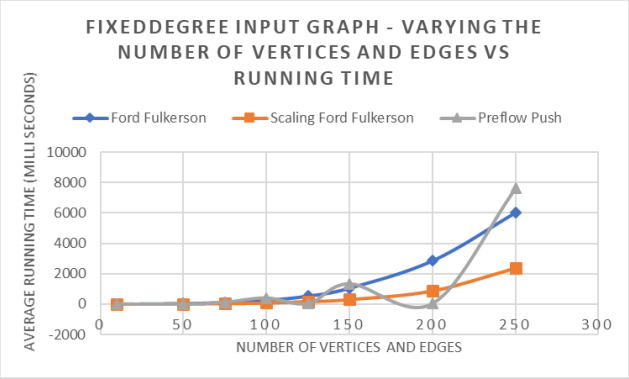

Figure 8: Fixed Degree Graph: Varying the number of vertices/Edges Vs Average running time

Our hypothesis for varying this parameter is that the running time for all the three algorithms depends on vertices/edges. So, we wanted to see with increase in number of vertices/edges, how the running time of the three algorithms is getting increased as in line with the theory.

Observation from the figure 8: When the number of vertices/edges are small, the running of all the three algorithms are almost similar and with increase in the number of vertices/edges, the running of three algorithms is getting increased. The running time of pre-flow push is V^2E and hence after certain higher value of number of vertices/edges we can see the running time of pre-flow push is greater than the other two algorithms which we were expecting to see in results as well.

Test 3: Test is repeated with varying only the number of vertices this time whereas number of edges and capacity are kept constant (minimum capacity = 0 and maximum capacity = 250, number of edges = 50). Hypothesis behind varying this parameter is that running time of pre-flow depends on number of vertices whereas the Ford-Fulkerson, scaling Ford-Fulkerson does not depend. So, the expectation from this test is that the running time of pre-flow push to increase with the increase in number of vertices and the running time of other two algorithms to be consistent with their running time. The number of vertices varied as follows: 60, 100, 150, 200, 250. The test results for this test is captured and plotted in figure 9 with number of vertices in X-axis and Average running time in Y-axis.

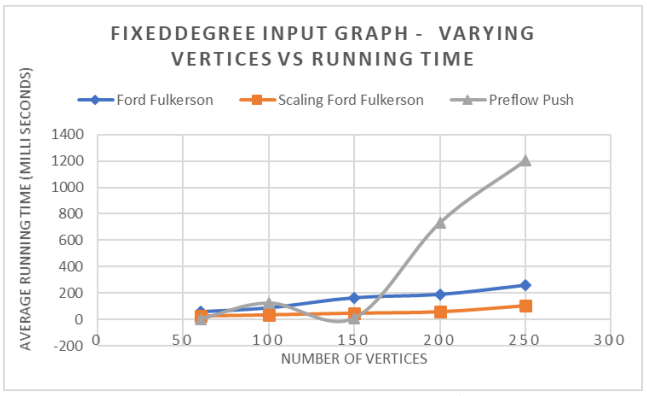

Figure 9: Fixed Degree Graph: Varying the number of Vertices Vs Average running time

From the figure 9, we observed the results to be consistent with our hypothesis. The running time of Ford-Fulkerson and Scaling Ford-Fulkerson algorithms is consistent and the running time of pre-flow push increases with the increase in the number of vertices.

Test 4: The last test was performed for Fixed Degree graph by varying the number of edges and keeping the number of vertices and the capacity as constant (minimum capacity = 0 and maximum capacity = 250, number of vertices = 250). The motivation for doing this test is that the running time of all the three algorithms depends on number of edges and hence we would expect with increase in number of edges, the running time of all three algorithm to be increasing. The number of edges varied as follows: 50, 90, 140, 190, 240. The test results for this test is captured and plotted in figure 10 with number of edges in X-axis and Average running time in Y-axis.

From Figure 10, it is observed that, with increase in number of edges, the running time of all the three algorithms are increasing as expected. For the last data, the running time of pre-flow push drops. We guess this is because of the number of relabel operations for this specific input file is less which is due to the distribution of capacity value on edges (capacity value randomness).

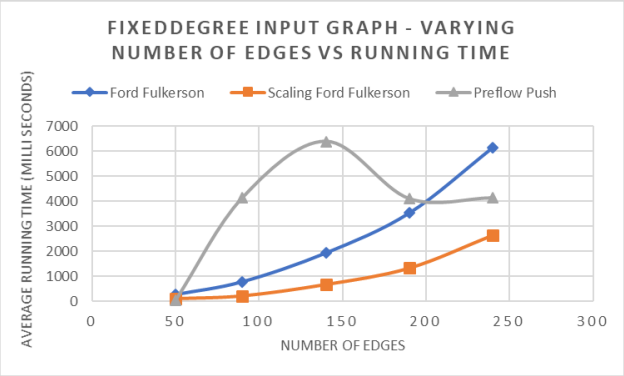

Figure 10: Fixed Degree Graph: Varying the number of Edges Vs Average running time

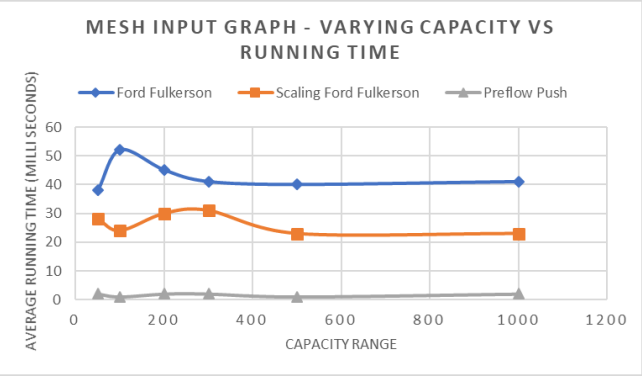

Figure 11: Mesh Input Graph: Varying the capacity Vs Average Running time

4.6 Summary of the test with fixed degree graph as input

From the tests conducted with Fixed degree graph as input, we observed that the pre-flow push algorithm performed better when the capacity range is varied. In case of variation in the input size (number of vertices/number of edges), the scaling Ford-Fulkerson performed better than the other two algorithms. It was observed that the theoretical running time of the algorithms were in line with the results that we got from our tests. In two places (for specific dataset), there is fluctuation in the running time of pre-flow push algorithm from the expected behavior, which we guess is because of the increased/decreased number of relabeling operations for those particular datasets which in turn increases/decreases the running time of the pre-flow push algorithm respectively. The tests that we have done for the Fixed Degree graph was done on the following machine with specification as: Acer Aspire E15 Inter Core i3, 2.4 GHz 4GB RAM.

4.7 Mesh Graph

The parameters that we can vary in Mesh graph are: number of rows, number of columns, capacity value on edges. There is a flag -cc in Mesh Graph Generator program which will set the capacity as constant in all edges for input file getting generated for that run when -cc is set. We generated all our input files with -cc flag set.

Test 1: For testing with mesh graph as input, we started with varying the capacity value on edges by keeping the number of rows (m) and columns (n) as constant (m=30 and n=35) with -cc flag set. Since the -cc flag is set, the capacity on all edges was same. For example, for input file generated with capacity as 50 will have capacity as 50 on all edges. The capacity values were varied as follows: 50, 100, 200, 300, 500, 1000. Test was conducted for 10 iterations and graph was plotted for the results with capacity in X-axis and Average run time in Y-axis (Figure 11).

Observations from figure 11 are as follows: We know that pre-flow push does not depend on capacity and the same had been reflected in results as well. Pre-flow push performed better than the other two algorithms. Among the remaining two algorithms, scaling performed better than the Ford-Fulkerson as expected.

Test 2: Second test was performed by varying the number of rows and columns by keeping the capacity as same for all input files (capacity = 100). The number of rows/columns were varied as follows:10, 50, 75, 100, 125, 150, 200. Test was conducted with these input files for 10 iterations and the results were captured. The graph was plotted with number of rows/columns in X-axis and Average running time in Y-axis (Figure 12).

We observed that varying the number of rows/columns did not affect the running time of the pre-flow push algorithm. Initially all the three algorithms were performing similar. With the increase in the number of rows/columns, the running time of Ford-Fulkerson and Scaling algorithm increases whereas the running time of the preflow-push algorithm was consistent.

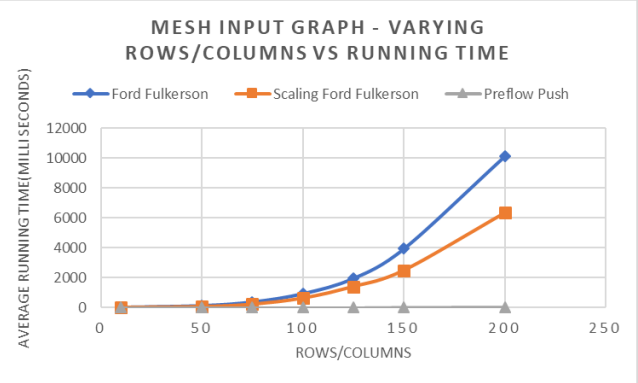

Figure 12: Mesh Input Graph: Varying the number of rows/columns Vs Average running time

Test 3: Then next test was performed by varying only the number of columns by keeping the capacity value and number of rows as constant (capacity =50 and rows = 50). The value of columns was varied as follows: 10, 50, 100, 150, 200. The results were plotted in a graph with number of columns in X-axis and Average time in Y-axis (Figure 13).

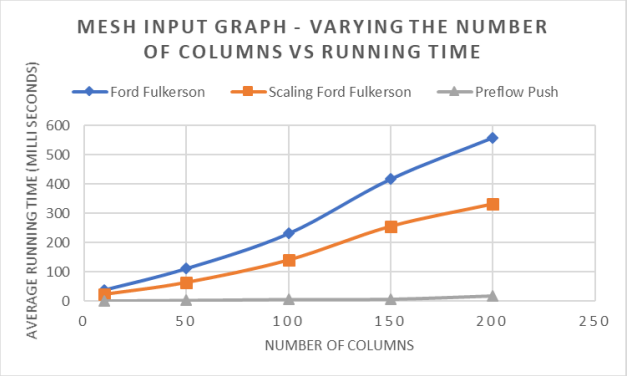

Figure 13: Mesh Input Graph: Varying the number of columns Vs Average Running time

It was observed that results were like the results obtained from the previous test (Figure 13 has results like figure 12 results).

4.8 Summary of the test with Mesh graph as input

With the tests results obtained from the Mesh graph input testing, push-relabel algorithm performed better than the other two algorithms. The running time of other two algorithms increases with increase in number of rows/columns. From the remaining two algorithms, scaling Ford-Fulkerson performs better than the Ford-Fulkerson algorithm. The tests that we have done for the Mesh graph was done on the following machine with specification as: Acer Aspire E15 Inter Core i3, 2.4 GHz 4GB RAM.

5 Conclusion

After a detailed study of implementing network flow algorithms we did come to know several interesting facts about the Ford Fulkerson, Scaling Ford Fulkerson and Pre-flow Push Algorithms.
Some of the interesting observations while experimenting by providing different input graphs such as bipartite, fixed Degree, Random and mesh graphs with respect to above three mentioned algorithms are:
First, for the bipartite graph input, all the three algorithms initially give almost the same running time, however as the number of nodes increased in the input graph we did notice the change in performance of preflow push and Ford Fulkerson algorithms.
Second, for the random graph, again the scaling for Fulkerson algorithm performs better in all the test cases we conducted. Initially when we try to increase the number of nodes, for the first two values such as 50 and 100 both scaling ford Fulkerson and preflow push algorithm performs almost the same, when the number of vertices increased with the step size of 50. As the number of nodes increased to very large number, the scaling ford Fulkerson algorithm performs the best among three algorithms.
Third, for the fixed degree graph input, with the increase in capacity, we did notice that the preflow push algorithm performs better. Whereas in increasing the no of nodes shows that the scaling for Fulkerson algorithm is working efficiently. With this, we did observe a pattern for scaling ford Fulkerson and Pre-flow push algorithm that there is dependency with respect to capacity in preflow push algorithm. The higher the capacity the better is the preflow push algorithm efficiency. Also, another pattern we observed with respect to scaling ford Fulkerson algorithm is that there is a dependency with respect to number of nodes. Higher the number of nodes, the better is the performance of scaling ford Fulkerson algorithm.
Lessons Learned:
Hence by involved in empirical study of network flow, we have understood plethora of concepts about what happens in a network flow and all the team members thoroughly understood the concept of ford Fulkerson, Scaling Ford Fulkerson and Preflow Push algorithms. Also, we have tried out different data structures during implementation such as LinkedList, arrays, HashMap and queues. To have the better running for the algorithms we have tried out different data structures and understood clearly which one must be included to get the efficient results. Stepping through the code during debugging helped us improve the debugging skills.

6 Division of Labor:

Team Members

Spoorthy Balasubrahmanya: Designed, developed and implemented Scaling Ford Fulkerson Algorithm and generated Bipartite input graphs and tested it. I have generated 15 input graphs and test over it. Also, the output is plotted on graphs to have a better understanding on the trends for each algorithm. Contributed in creating a comprehensive project report, code documentation.

Bharathi Manoharan:
Designed, developed and implemented a part of Pre-flow push Algorithm. Generated Mesh input graphs and tested it. I have generated around 25 input graphs and tested our code using the generated input files. Also, the results were plotted

in graphs to have a better understanding on the running time for each algorithm. Contributed in creating a comprehensive project report, code documentation.

Misba Momin:

Designed, developed and implemented Pre-flow push Algorithm. Generated Fixed Degree input graphs and tested it. I have generated around 25 input graphs and tested our code using the generated input files. Also, the results were plotted in graphs to have a better understanding between the running time of three algorithms. Contributed in creating a comprehensive project report, code documentation.

Rituja Dange:
Designed, developed and implemented Ford Fulkerson Algorithm. Generated Random input graphs and tested it. I have generated 30 input graphs and tested it with our code. Also, the results are plotted on graphs to have a better understanding on the trends for each algorithm. Contributed in creating a comprehensive project report, code documentation.

7 References

[1] Algorithm Design Textbook by Jon Kleinberg. Eva Tardos