Empirical Study Project on Network Flow

Yaqun Yu
yaqunyu@uw.edu

Fang Chen
fangchen@uw.edu

Xinhe Wang
xinhehe@uw.edu

Ching-Lun Lee
aloon@uw.edu

ABSTRACT
This report is contains four parts: introduction, methodology, result and future work. It is based on the three algorithms to find maximum flow for network problem. These three are the most popular algorithms in network flow field. Maximum flow problem is in a directed graph with source vertex s, sink vertex t, and non-negative arc capacities, to find a maximum flow from s to t. This problem first formulated in 1954 by T. E. Harris and F. S. Ross as a simplified model of Soviet railway traffic flow [1]. The well known, Ford-Fulkerson algorithm, is introduced to world by Lester R. Ford, Jr. and Delbert R. Fulkerson in 1955. In this report we implement three algorithms in java and test them in four types of graph. During the implement process, the Preflow-Push algorithm is new to our team. We will introduce our implement and empirical study procedure in details in the flowing part of this report.

1. INTRODUCTION
Maximum-Flow is based on directed graph concepts. In the Ford-Fulkerson algorithm we need to push flow through our graph, which is limited, by bottleneck in our residual graph. The same situation is applied to Scaling Ford-Fulkerson (SFF) algorithm, but in SFF instead of choose any random augment path, we introduce a parameter to be the measure bar when we choosing the augment path. In Preflow-Push algorithm we do not use augment path at all, which saves us time for choosing the augment path. Instead we push flow from source s to sink t as much as possible at first and are we do not have graph that can maintain amount of our flow, then we just return the excess back to source.

Start from here we are going to discuss our algorithm in details. First start with Ford-Fulkerson, as discussed in class, our graph has a residual graph, which contains all the nodes in our original graph, in our residual graph we have two types of edges, if we can find a s-t path in residual graph we will increase original graph's flow by bottleneck in this s-t path or decrease in order to push more from source to sink. The running time for Ford-Fulkerson algorithm is O (mC).

For SFF algorithm, this algorithm improve the choose method when come to augment path, this will reduce the running time of our algorithm by only choose the best augment path. The running time for SFF is O (m^2 log C), which is pretty useful in practice.

Preflow-Push (PFP) is a new algorithm to our team. The difference between PFP with Ford-Fulkerson instead of worry about the bottleneck in residual graph, we push as much as possible from source. As we all know in Ford-Fulkerson, each node except s and t, the flow into it is equal to the flow out of it. But sometimes we cannot maintain this in time, so here is Preflow notation. In a Preflow, the flow entering the node is allowed to be more than the flow leaving it. The algorithm introduced some indexes like label h (v), excess ef (v) for nodes and active nodes, in order to help the push process and find the max-flow quickly. Because push can only from higher label node to lower label node. Since we want to start from the source (s), the initial label for s is equal to n to make sure it can go through every node in graph if possible. Since the sink is our destination, we want it to be lower enough so flow can be sent through, so we initial it to h (t) = 0.

Consider any node v that has excess—that is, ef (v) > 0. If there is any edge e in the residual graph G_f that leaves v and goes to a node w at a lower height, so we can modify f by pushing some of the excess flow from v to w. We will call this a push operation. If we run into the situation that $e_f(v) >0$, (v, w) in G_f but h(w)>=h(v), then we cannot push further, now we need to execute relabeling operator to increase h (v) by one at a time to see if we can push again. If so do the push, otherwise repeat the situation until we can push again. When there is no excess for every node, we will find our desired flow—max-flow. This is because in this situation, our residual graphs have no s-t pass anymore, which is syncs with the definition of max-flow in Ford-Fulkerson and SFF. In the worst-case scenario, PFP algorithm running time is O (V^3).

2. METHODOLGY
The empirical study project consisted of several phases, begin with divide the work and design our implementation of three algorithms that are given and culminating with the empirical analysis of the running time and their sensitivity to each variables.

After divided three algorithms to our teammates, we first studied these algorithms from our textbook [2]. And then we reviewed the data code provided by Professor Kayee [5] on our course webpage. In this project we are using four types of data structure as our data source. We have Random graph, Fixed Degree graph, Mesh graph and Bipartite graph. In the given graph code there are several classes represent each characteristic in a graph for us to use. Such as vertex class represents a vertex in a graph and also data object that associated with a vertex in case we want to mark the vertex, which is been visited. Similarly, we have class edge along with a label object, which helps us a lot in our algorithm implement, for instance, to mimic the residual graph that goes along with flow. We can maintain our flow and capacity data values in each edge data object. We have graphinput class as well so that we can read graph file into our algorithm after generate the kind of graph we want.

For graph input and generation of our test graph, we were benefit from the graphGenerationCode that provided by Professor again. The code include java classes for generating bipartite, fixed degree, mesh and random graphs, each type we also have some text file as test graph that we can use to confirm our implement. These generators not only help us verify the correctness of our implantation but also efficiency our analyze work which will discuss later.

Our team implementation is using java as our universe language in convenience to combine together since each algorithm was coded by different group member/members. We both run into multi technique errors along the way of our implement. For example when we first combine our code together we find that when FF & SFF using the same graph, there is conflict between them because they both mark the graph and when one algorithm find the max flow first, the other algorithm can not use the graph anymore because current flow is feasible. We fixed this situation by input three times for each original test graph, one for each algorithm to avoid conflict. In order to make sure the correctness of our implementation, we not only use the sample graph that provided by professor, but also generate our own graph based on different level of varieties by using the graph generator. Because each algorithm is sensitive to different properties, by control the three main varieties, number of vertices, range of capacity and the density of edges, we can generate graphs for each type on every level to verify the validity of our implementation. In this way, we won't find our experimental data would be discrepancy with our expectation during analyze phrase.

During the graph generation we find that there is no direct edge variable we can manipulate. Instead we have density for random graph, edge out node for fixed graph and probability for bipartite graph, for mesh graph, the edges are fixed once we settle down the number of vertex. We did a little research on density, according to Wikipedia [6], for undirected simple graphs, the density is defined as D=2|E||V|(|V|-1), for directed simple graphs, the graph density is defined as D=|E||V|(|V|-1), our first approach is by control the vertex number and calculate different edges by this formula, turns out in our code, the density is not that simple, so we change our approach by taking density or probability or edges out of node as our yardstick for edges. Choosing property range of each variable is another challenging. By checking the graph that provided, we decide that this is suitable for exam the sensitivity to number nodes for our implement. After deep and numerous test runs we finally determine the best capacity range to test the capacity sensitivity for each algorithm and based on result of that, we choose a typical graph to test our sensitivity of edges.

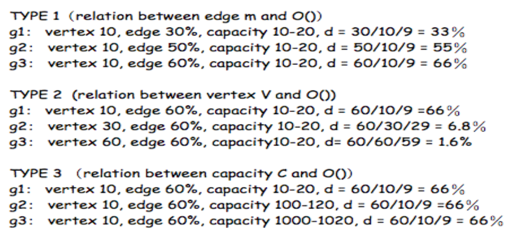
After we got all our graph text files for each type of graph, we combine three algorithms together and run three times for each type of variety control. All the test runs are achieved in the same computer under the same environment to eliminate all possible outside intrusions. By take average for each test, we use Microsoft excel to generate our result diagram. The result and analyze will show in next phrase.

3. RESULTS

We will introduce our study in details in this part. Start by graphed data, and we have analysis, observation during test and end by conclusion.

3.1 Graphed Data

As descript before, Figure 1 shows our original idea for our dataset, turns out the graph cannot be generated properly. So we change our approach by using provided sample graph. We first analyze each algorithm separately and then combine. After individual algorithm analyze, we put three implements together to generate their sensitivity for each variables.



TYPE 1 (relation between edge m and O())
g1: vertex 10, edge 30%, capacity 10-20, d = 30/10/9 = 33%
g2: vertex 10, edge 50%, capacity 10-20, d = 50/10/9 = 55%
g3: vertex 10, edge 60%, capacity 10-20, d = 60/10/9 = 66%

TYPE 2 (relation between vertex V and O())
g1: vertex 10, edge 60%, capacity 10-20, d = 60/10/9 =66%
g2: vertex 30, edge 60%, capacity 10-20, d = 60/30/29 = 6.8%
g3: vertex 60, edge 60%, capacity10-20, d= 60/60/59 = 1.6%

TYPE 3 (relation between capacity C and O())
g1: vertex 10, edge 60%, capacity 10-20, d = 60/10/9 = 66%
g2: vertex 10, edge 60%, capacity 100-120, d = 60/10/9 =66%
g3: vertex 10, edge 60%, capacity 1000-1020, d = 60/10/9 = 66%

Figure 1 Original Dataset

3.2 Result Analysis

3.2.1 Individual analyze

We run three different experiments to test the performance of the Ford-Fulkerson Algorithm under different types of graphs. Firstly, to discover the relationship between the number of edges and the running time of FF algorithm, we fixed the number of vertices and the number of capacity. From figure 1 we can see that as the number of edges going up the running time of FF increases as well. In addition, we fixed the number of edges and number of capacity, we can see from figure 2, as the number of vertices going up the running time of FF is not strictly increasing or decreasing, therefore, the number of vertices do not influent the running time of FF algorithm. Furthermore, we fixed the number of vertices and edges, as the number of capacity going up, the running time going up as well. The results of our experiments are highly consistent with the theory of the Ford-Fulkerson Algorithm runtime analysis, for that the running time of FF algorithm is only associated with the number of edges and the number of capacity of the graph.

As show in Figure 3 that the running time for individual test with SFF, we can see that the running time increases when the number of edges increases. The running time also increases when the capacity increases, but it is not obvious. If we increase the number of nodes, the running time may increase or decrease. Thus, there is no relationship between running time and number of nodes.

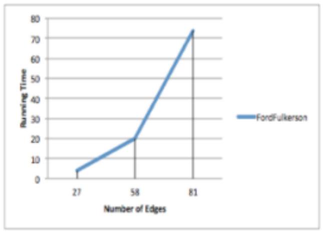
For Preflow push algorithm, we can clearly see from Figure 4 that the vertex number is the main reason that causes the increase of running time. Capacity's affection is almost negligible.

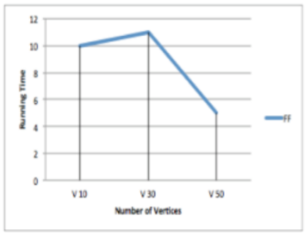
1. Fix the Number of Vertices and Capacity

	E27	**E58**	**E81**
Running Time (ms)	4	20	74
MaxFlow	39	70	115

2. Fix the Number of Edges and Capacity

	V10	**V30**	**V50**
Running Time	10	11	5
MaxFlow	112	30	25





3. Fix the Number of Vertex and Edges

	C10-20	**C100-120**	**C1000-1020**
Running Time	7	11	17
MaxFlow	112	873	8068

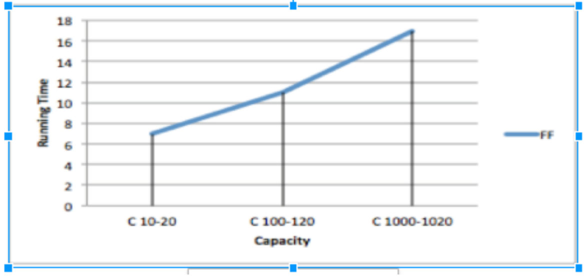


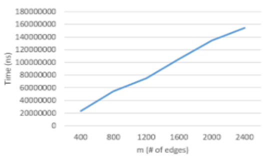
Figure 2 Ford Fulkerson on Random Generated Graph

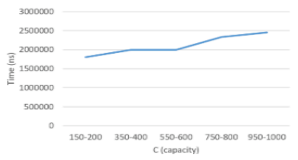
1. Fix the Number of Vertices and Capacity

	E400	E800	E1200	E1600	E2000	E2400
Runtime(ns)	23246598	54695235	74521259	105423598	134568254	154359875

2. Fix the Number of Vertex and Edges

	C150-200	C350-400	C550-600	C750-800	C950-1000
Runtime(ns)	1803908	1997997	1998918	2334725	2455652





3. Fix the Number of Edges and Capacity

	V30	V60	V90	V120
Runtime(ns)	18730824	13047640	6700737	10748312

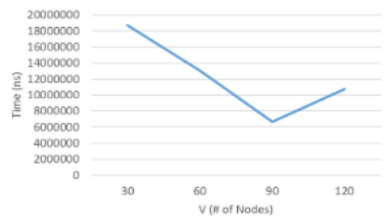
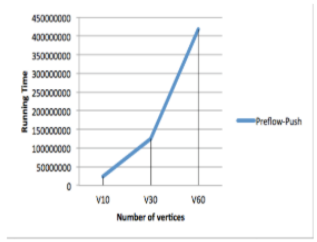


Figure 3 SFF on Random Generated Graph

1. Fix the Number of Capacity
Capacity is min = 10, max = 20

2. Fix the Number of Vertex
Vertex = 10

	V10	V30	V60
Running Time	23472833	125815418	417444374



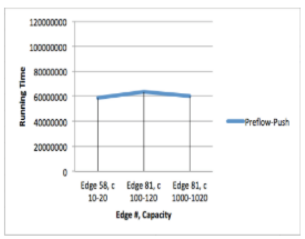


Figure 4 Preflow Push on Mesh and Random Generated Graph

3.2.2 Three Algorithms Combine Analyze

After individual algorithm analyze, we put three implements together to generate their sensitivity for each variables. As show in Figure 5. In our test set, the graph type is Random, Bipartite, Mesh, and Fixed Degree. The graph size is small and medium. For each test, we run three times, and then take the average running time as our test data.

Our analysis is four types of graph both only change on the size of graph, which is mainly on vertex. We can see from above, Preflow-Push has a dramatic increase in running time as the vertex number goes up for both type of graphs. The FF is only response to random type when number of vertex increase, while the difference in running time of SFF is almost negligible. We can say that Preflow-Push is the most sensitive to vertex number among all three algorithms.

3.2.3 Other Two Variables Control

In previous work we already involve the vertex number control method, and we analyze the affection for it to running time.

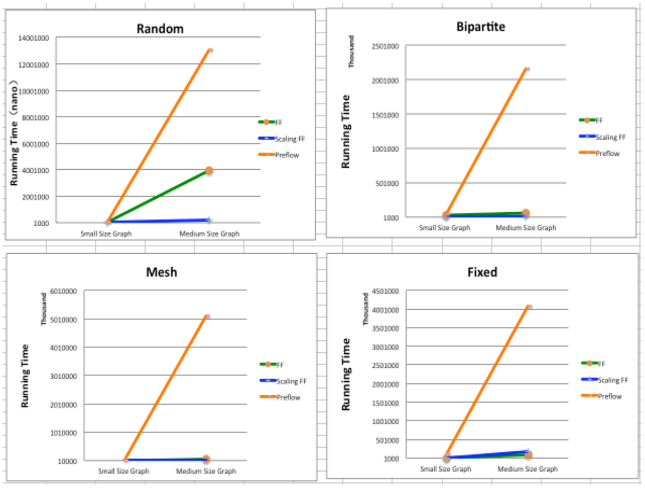


Figure 5 Three Algorithms in Four Graph Types

During this analyze procedure we first have no clue about how to define the range, after many random test we figure out the inner rules and here we select five ranges that we think is most typical for our result. For each type of graph, we are using the range for capacity in 2,000-5,000, 2,000-10,000, 10,000-50,000, 100-10,000 and 100-100,000. We choose these five because they represent different range level, 2,000-10,000, and 10,000-50,000 are both in four times relation while one is in lower range and the other is in higher range. 2,000-5,000, 2,000-10,000 is both started from 2,000, one is reach up to 5,000, and the other one is much higher. 100-10,000 and 100-100,000 are both in large range level, but the other one is covered more than the first one. We believe these five sets covers all kinds of change possibilities in capacity range level.

And for the vertex number and density, since we know the running time of PFP is highly depends on vertex number, but we want other two algorithms running in the same normal condition which means capacity has a range and not too little vertices. Thus we pick 50 as our vertex number as well as the density 0.99/99%.

Random

Vertex 50, Density 99

	2000-5000	2000-10000	10000-50000	100-10000	100-100000
FF	364432615.3	349036673.7	352377013.7	265159916	199301370
	332765036	348521611	417948401	287462703	193416757
	448544386	386696481	313605937	261556580	186401240
	311988424	311891929	325576703	246460466	218086114
Scaling FF	153428375.7	38765108.33	38075995	32581792	34920877.7
	144340293	38115513	38842415	33148069	34005508
	176088835	40382459	39115994	32867466	33962021
	139855999	37797353	36269576	31729842	36795104
Preflow	993232366.7	46811690.33	41550229.67	626790031	642080264
	1041614823	46922833	34246370	696828558	660579164
	984363270	52557401	44233927	607801857	649103726
	953719007	40954837	46170392	575739677	616557901

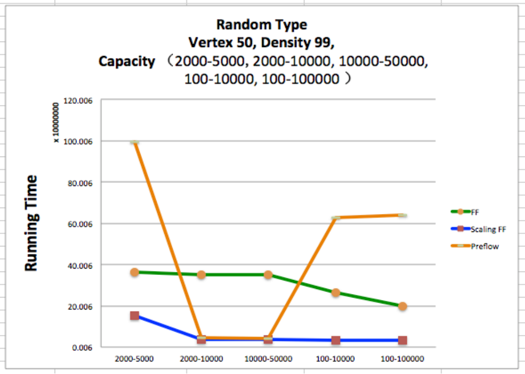
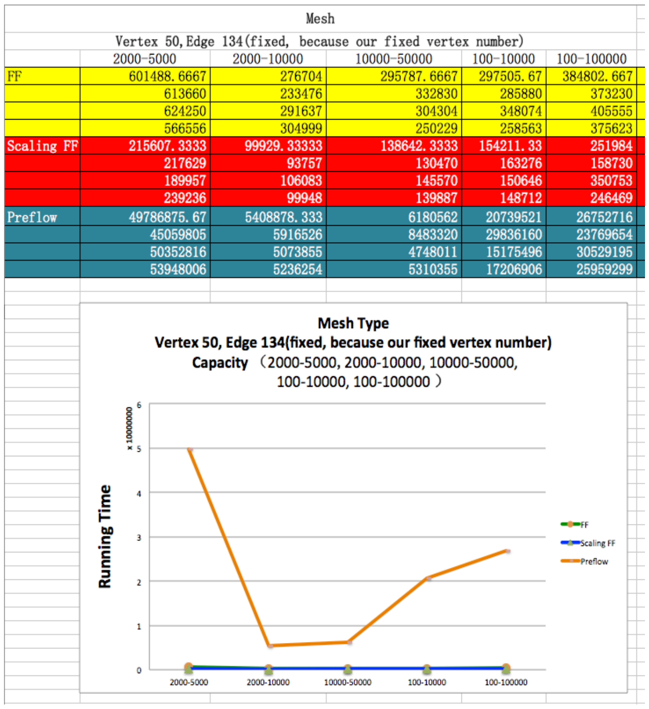


Figure 6 Three Algorithms in Random Type Graph

As we can see in Figure 6, this test is based on random graph type. The running time performance for PFP is more vibrate than the other two, and when the capacity range has the four times relationship, the running time is minimized. Otherwise the running time is higher than the other two. The SFF is much stable and stay at the lowest running time all the time. And FF is pretty stable as well, it always stay higher than SFF.

Figure 7 is the result we got for mesh type graph, as we mentioned before, whenever our vertex number is fixed, the edge number is automatically fixed. As usual we choose the average of three times tests as our data, and we can see from the graph that PFP is similar shape with the one in random, but this time, both FF and SFF performed rather well than PFP, so we have no junction point between them, which means that in mesh, the performance of FF and SFF is better in mesh graph than random.

As for bipartite, the shape for each algorithm is pretty different with them in previous graph. This time, FF is not as good as SFF, SFF's running time is always the lowest as usual, but along with the capacity range changing, the FF and PFP's running time is dropping, for PFP there is two turning points. At the end FF is almost parallel with SFF but still higher and PFP is drop to the same as SFF. From this figure we can say that in bipartite, SFF is more sensitive of capacity than the other two.



Mesh

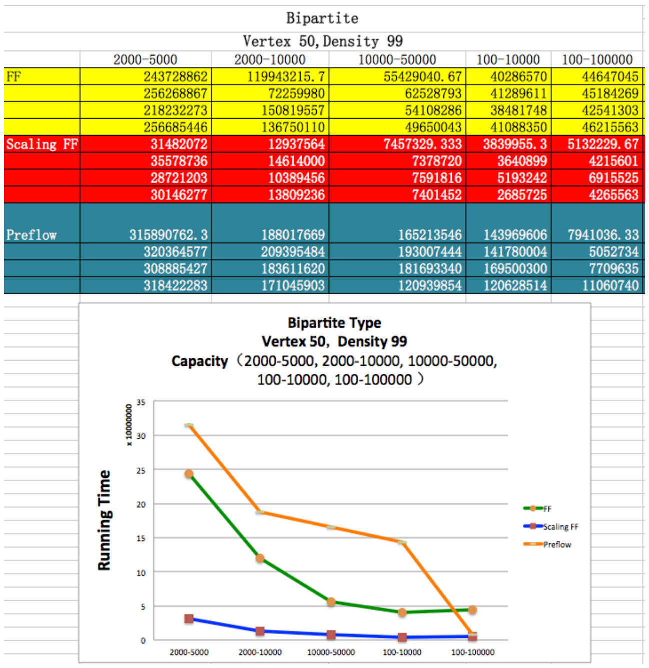
Vertex 50, Edge 134(fixed, because our fixed vertex number)

	2000–5000	2000–10000	10000–50000	100–10000	100–100000
FF	601488. 6667	276704	295787. 6667	297505. 67	384802. 667
	613660	233476	332830	285880	373230
	624250	291637	304304	348074	405555
	566556	304999	250229	258563	375623
Scaling FF	215607. 3333	99929. 33333	138642. 3333	154211. 33	251984
	217629	93757	130470	163276	158730
	189957	106083	145570	150646	350753
	239236	99948	139887	148712	246469
Preflow	49786875. 67	5408878. 333	6180562	20739521	26752716
	45059805	5916526	8483320	29836160	23769654
	50352816	5073855	4748011	15175496	30529195
	53948006	5236254	5310355	17206906	25959299

Mesh Type
Vertex 50, Edge 134(fixed, because our fixed vertex number)
Capacity（2000-5000, 2000-10000, 10000-50000, 100-10000, 100-100000 ）
Running Time
x 10000000
0
1
2
3
4
5
6
2000-5000
2000-10000
10000-50000
100-10000
100-100000
FF
Scaling FF
Preflow

Figure 7 Three Algorithms in Mesh Type Graph

For fixed type graph, it is different as well. At the first two range the three algorithms almost at the same level of running time. But start from 10,000-50,000, the running time of PFP start to increase, and reach to the highest at 100-10,000 and then drop down to the same as the other two at range 100-100,000. We think for this type of graph, the PFP's running time in high level range is slower than the low lever range in general, but there is a limit, once the range reach to that limit, it becomes fast again. The other two algorithms stay at low running time and don't have much change along the procedure. Again, PFP is more sensitive to the range of capacity than the other two.

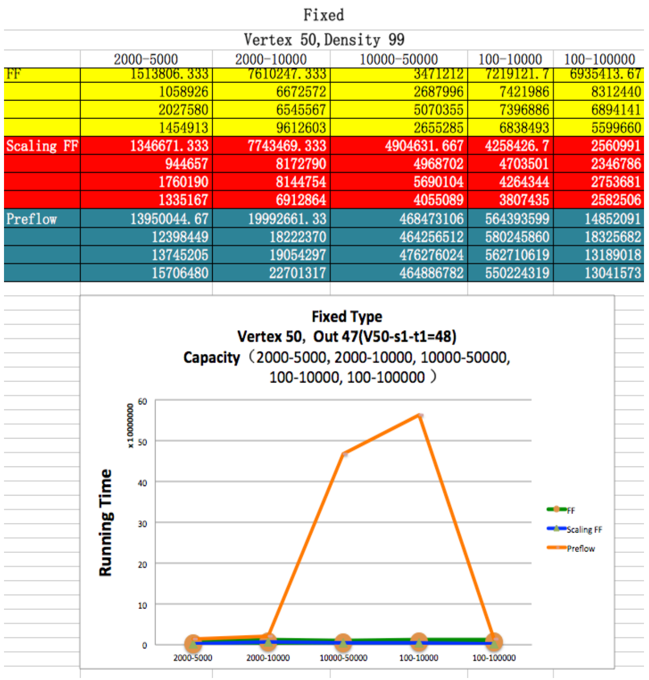


Bipartite

Vertex 50, Density 99

	2000–5000	2000–10000	10000–50000	100–10000	100–100000
FF	243728862	119943215. 7	55429040. 67	40286570	44647045
	256268867	72259980	62528793	41289611	45184269
	218232273	150819557	54108286	38481748	42541303
	256685446	136750110	49650043	41088350	46215563
Scaling FF	31482072	12937564	7457329. 333	3839955. 3	5132229. 67
	35578736	14614000	7378720	3640899	4215601
	28721203	10389456	7591816	5193242	6915525
	30146277	13809236	7401452	2685725	4265563
Preflow	315890762. 3	188017669	165213546	143969606	7941036. 33
	320364577	209395484	193007444	141780004	5052734
	308885427	183611620	181693340	169500300	7709635
	318422283	171045903	120939854	120628514	11060740

Figure 8 Three Algorithms in Bipartite Type Graph



Fixed

Vertex 50, Density 99

	2000–5000	2000–10000	10000–50000	100–10000	100–100000
FF	1513806. 333	7610247. 333	3471212	7219121. 7	6935413. 67
	1058926	6672572	2687996	7421986	8312440
	2027580	6545567	5070355	7396886	6894141
	1454913	9612603	2655285	6838493	5599660
Scaling FF	1346671. 333	7743469. 333	4904631. 667	4258426. 7	2560991
	944657	8172790	4968702	4703501	2346786
	1760190	8144754	5690104	4264344	2753681
	1335167	6912864	4055089	3807435	2582506
Preflow	13950044. 67	19992661. 33	468473106	564393599	14852091
	12398449	18222370	464256512	580245860	18325682
	13745205	19054297	476276024	562710619	13189018
	15706480	22701317	464886782	550224319	13041573

Figure 9 Three Algorithms in Fixed Degree Type Graph

Since the performance in fixed is more interesting than the other types we decided to use this graph type as our test type for the sensitivity of edges test. As we can see from Figure 10, we picked the capacity range from 2,000-10,000, in order that the running time for three algorithms are at the same level. As for the chose of edges, based on previous high amount of tests, we find out that the level of vertex can make a huge different. Since last time we use 47 as our out edges in respect of fixed degree graph. We are choosing 47 as well this time, and the other two levels are 10+ and 20+. From Figure 10 we can see clearly that when it is in edge 25, the running time reach to the highest. And when it is in 10, the running time is higher than the other two, but not as high as the one at 25. At edge 47, it is drop down to the same level with the other two. Again, we are using the average of multiple tests as our data. Obviously, PFP is sensitive to edges than the other two as well.

Vertex 50, Cap 2000-10000

	10	25	47
FF	980825	2192649	3041579
Scaling FF	749130	1873600	1783107
Preflow	120168566	293921063	11534810

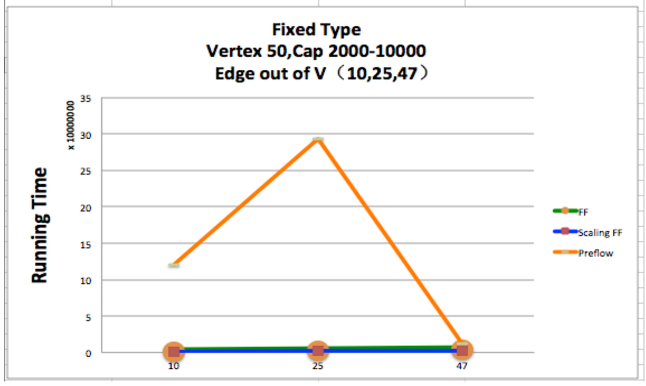


Figure 10 Three Algorithms in Fixed Degree Type Graph

3.2.4 Observation

We firstly tested the performance of our three algorithms on the eight original graphs provided with the project description. We noticed that the Ford-Fulkerson Algorithm and the Scaling Ford-Fulkerson Algorithm performed much better than the Preflow-Push algorithm. In addition, the Scaling Ford-Fulkerson is generally the fastest. However, in our later tests, we found out that under some conditions Preflow-Push algorithm performs much better than the Ford-Fulkerson algorithm for finding the max flow and mostly as efficient as the Scaling Ford-Fulkerson Algorithm. And according to the theories of the runtime of three algorithms, the Preflow-Push algorithm should works better than other two algorithm when the number of the vertices is fixed to a value and the capacity is big enough to overweight the influence brought by the number of vertices on our performance testing. To confirm our observation, we tested on five different graphs of each type of graph. The five graphs have the same number of vertices (which is 50) and the same other parameters (like, density, number of edges per vertex and probability) used for graph generation except the range of the capacity (which are 2000-5000, 2000-10000, 10000-50000, 100-10000, and 100-100000). We can see from the Figure 6, Preflow-Push performed very slowly than other two algorithms while under some conditions it performed better than Ford-Fulkerson algorithm and even Scaling Ford-Fulkerson algorithm.

3.3 Conclusion

Apparently, we can give a conclusion that it is better for us to scale the Ford-Fulkerson algorithm. The performance of scaling Ford-Fulkerson is better than original Ford-Fulkerson in most of the time. In some types of graph such as random and bipartite, the scaling one performs even much better. However, this doesn't mean that there do not exist a graph type that would lead scaling to be a worse choice. More set of graphs need to be experimented to raise the competence in this conclusion.

As for Preflow-Push algorithm, it is very sensitive to the change of variables. It performs not as well as scaling Ford-Fulkerson algorithm in most of the graphs. Based on the experiment we have done, scaling Ford-Fulkerson is a feasible way for us to solve the maximum flow problem since the running time it takes is fast and steady, but the future work show that Preflow-Push can be much better if we improve our implement, which we will discuss in details.

4. FUTURE WORK

We have already implemented SFF, which has better performance than FF in most of the time. However, there exists another algorithm, which is called *strongly polynomial algorithm* [2]. This algorithm is polynomial in |V| and |E| only, and works with numbers having a polynomial number of bits. We can implement this by applying BFS to find the augmenting path. It had been proved that this algorithm would terminate in at most O (mn) iterations. We can implement this algorithm and compare with the three algorithms we have already implemented, and find out which one has better performance under which situation.

If we were to do the experiments again, we'd have to define the class method before implement it so we can keep consistency between the coding works of different people.

Since what we have analyzed is from the changing of number of vertex and capacity, we want to test the three algorithms' performance on different edge values. We compared the performance in a FixedDegree Graph, we will want to see their performance on other types of graphs by changing the number of edges, and then we can do more conclusions on that circumstance.

Another future work is to improve the performance of Preflow-Push algorithm, currently our code picks up the vertex v which has highest excess value, the time efficiency is O (n^3). We can improve it to O (mn) by a pointer current (v) for each node v o the last edge on the list that has been considered for a push operation.

From the coding work and performance result analyzing, we found that choosing a kind of algorithm is not the only one factor that affects the performance. For example, the Preflow-Push algorithm we implemented has $O(n^3)$, but the number of vertices is not the only one which impact its performance, we observed given a fixed number of vertices, Preflow-Push has different performance time under different number of edges test graphs. By reviewing our code, we found that there is optimization work to it, such as improve the method of looking up edges, getting edge capacity. Using Hash Map rather than scanning the whole list of edges can achieve a lot of querying work.

5. REFERENCES

[1] Maximum Flow Problem https://en.wikipedia.org/wiki/Maximum_flow_problem.

[2] Jon Kleinberg, Éva Tardos, Algorithm Design, Published by Addison Wesley, 2006. ISBN 10: 0321295358 / ISBN 13: 9780321295354.

[3] COS 423 Lecture 21 Maximum Flows，Robert E. Tarjan 2011

[4] Advanced Algorithms ppt by Piyush Kumar

[5] Code provided by professor on canvas: https://canvas.uw.edu/courses/1020090/files/folder/project

[6] Density Graph https://en.wikipedia.org/wiki/Dense_graph