Empirical Study Project of Network Flow

Zheng Zhou
Institute of Technology

Pai Zhang
Institute of Technology

Paranjit Singn
Institute of Technology

1. INTRODUCTION

Network flow problem is a common and important problem in graph theory. In network flow problem, network refers to a directed graph. Each edge in the graph has a capacity and can receive a flow. There are a source node and a sink node in the graph. The flow leaves the source node and flows into the sink node. The flow must satisfy the capacity conditions. The flow of each edge should be nonnegative and not greater than the capacity of the edge. The flow should also satisfy the conservation condition. It means that all flow into a node v must be equal to all flow leaving the node v. In this case, what we are interested is to find a maximum flow of a given graph. In this project, we implemented three algorithms to solve this problem. We already know the procedure of these algorithms in the textbook. However, implementation is a different thing. There are several goals of this project. One of the goals is to practice programming skills. Another goal is to understand these algorithms better. We also can learn how to compare the efficiency of different algorithms using different data sets. After comparing the performance using four kinds of data sets, we found that Scaling Ford Fulkerson Algorithm is more efficient than Ford Fulkerson Algorithm in most cases. Prepush-Push Algorithm is not stable especially in random data set. But it can be the most efficient algorithm in some specific graphs.

2. METHODOLOGY

2.1 Implementation of Algorithms

2.1.1 Ford Fulkerson

Description of the implementation of Ford-Fulkerson(data structure in your code, what algorithm you used in your code, difficulty during the development and how you tested your code) The basic implementation idea we took for coding the algorithm is same as defined in the algorithm. We take the input from the file and try to create an object of network flow, using linked list of nodes and edges. The object created contains list of vertices in the network and edges along with its capacities. We keep the source node as the first item in the linked list of vertices and sink node as the last item. We included three different options for selecting the source and sink node. Namely, First, user can specify the name of source and sink node; Second, user can specify the index of source and sink node; and last user can just provide the input file and we would take source node as s and sink node as t by default. After initial initialization of user input to network flow object, we loop through the network until there exist a S-T path in the flow. In each interaction, we try to find a path with minimum bottleneck, and augment that path in the graph network created during initialization. To reduce the amount of memory used during execution, we *didnt* use a separate residual graph object. Instead, we reduced the capacity of edges in each iteration by the amount of bottleneck, thus allowing us to know how much more flow we can send across S-T path. In Ford-Fulkerson algorithm, it is assumed that we always have an S-T path, however, it *doesnt* define the way to find the S-T path. I tried multiple approaches to find the S-T path in the implementation namely BFS and DFS, and picked BFS during the final stage as it provided better run time during testing. We tested with multiple types of graph to get the runtime with different implementation aspect. One special aspect we tried to optimize was the way we selected bottleneck. We tried to send maximum bottleneck instead of minimum bottleneck in each step. However, this approach consumed lot of memory during execution as we were required to keep a track of each path possible in every interaction. For smaller density graph, it was permissible to keep track of each path, however, we found out that with large network, it was just not possible to keep track of each path to find the maximum flow and it would give us memory exception. So, we changed the implementation to select the minimum bottleneck as defined in the algorithm. The runtime of Ford Fulkerson is O(mC).

2.1.2 Scaling Ford Fulkerson

Scaling Ford Fulkerson algorithm is implemented in Java 8 by creating a class ScalingFordFulkerson. This class has 6 private parameters and 7 private methods, 1 public method. The important mothods are as follows:

1. The public method is for calculating the maximum flow by input 3 formal parameters: sinknode, sourcenode and a network flow graph.

2. The getAugmentingPath(int Delta) method is to define a method to return the path required and if there's no s-t path, then return another null array.

3. The BFS method is to find a simple S-T path: use

WOODSTOCK '97 El Paso, Texas USA

a queue to record the unvisiting vertices. Each time when begin traversing the edges of a vertex, poll that vertex from that queue. Then when the edge value is greater than scaling parameter Δ, and its end vertex is not visited, add this end vertex to the queue and change its status to visited. Update the path each time when we find a new vertex. Finally if it visited the sink node, then return true, else false.

4. The initialize method initialize the values of the simple graph.

5. The getincreasingFlow method is to find the minimum increasingFlow(>= Scaling parameter) of a determined s-t path. And augment the residul graph.

6. This augment method augments the given graph with value increasingFlow (>=Delta). It reduces the value of the forward edge by increasingFlow and adds/increases the value of reverse edge by increasingFlow.

The runtime of Scaling Ford Fulkerson is O(m^2logC).

2.1.3 Preflow Push

There are four main parts in the implementation of Preflow-Push Algorithm. The first part is initialization. We use HashMap data structure to store the height of the vertices, the flow of the edges and the excess values of the vertices. It is fast to get values using HashMap. The second part is to find active nodes. We iterate over the vertices in graph to find the active node. It will return the first found active vertex and return null value if it cannot find one. The third part is the push operation. If a given edge is applicable to be pushed, the flow of this edge will be pushed forward or backward. In this implementation, we do not need another data structure to store the residual graph. We can judge whether an edge is a forward path or a backward path by the direction of the edge and the flow on that edge. In the first version of the implementation, we update the excess values of all nodes. This is unnecessary and will reduce the performance of this algorithm. So we just update the excess values of the two vertices which related to this edge. The performance was improved dramatically. The forth part is the relabeling operation. This operation will increase the height of the given vertex by 1 if this vertex is applicable. A single relabeling operation will only take a constant time because we used HashMap.

2.2 Codes validations

Table 1 presents the results from inputs from the three samples given : g1 with Max flow 150 , g2 with mas flow 898, and 100v-5out-25min-200max with max flow 517. We found that all the results from our code matched the results as given in the sample. This means that our codes are correct in terms of validation.

2.3 Design of empirical study

In this report, we analyze the effects of four input of datasets on the runtime of the three algorithms. All our work were implemented on Eclipse by a machine of Intel (M) Core (TM) m3-6Y30 CPU 0.9Hz 1.51GHz, memory 4GB and X64 Win10 operating system.

For the input of Random datasets with parameters: number of vertices (n), density of edges(%), maximum capacity of edge and minimum capacity of edge. We analyze the effects of number of vertex on the runtime of the three algorithms, and we change the dense value from 11 to 100 (step 1) for the 3 vertex default numbers n: 40, 60, 80, as shown in Figs.1 (a) to (c). And also investigation on effects of maximum capacity on performance of runtime of the three algorithms were implemented. In each case, we change dense value from 11 to 100 (step 1), total 90 numbers, as shown in Figs.1 (d) to (f). All the runtimes results in sub-figures of Figure 1 are the average value of fifty implementations. The data of Figs.1(c) is shown in Table 2.

For the input of Fixed Range datasets with parameters: edges out of each node (out), number of vertices (n), maximum edge capacity, minnimum edge capacity. we analyse effects of the number of nodes, edges per node and the maximum edge capacity on the performance of runtime of the three algorithms, as presented in Figs.5-7.

For the input of Bipartite datasets with parameters: edges on source node, edges on sink node, probability P, max capacity Cmax, min capacity Cmin. We analyze the effects of P, number of source nodes, and max capacity on the performance of the three algorithms. The results are shown in Figs. 5 to 7 respectively. We choose ten values of P and five number of edges from source and 5 max capacity to investigate the effects.

For the input of Mesh datasets with parameters: row numbers m, column numbers n. We analyze the effects of m and n on performance of the three algorithms. And the corresponding results are shown in Figs 8-9. In each figure, we change parameters from 10 to 50 total five parameters to investigate the runtime.

3. RESULTS

3.1 Input of Random datasets

Figure 1 is the result of performance for the generated random data set. Each value is the mean value of 50 test results in order to improve the validity of the data and eliminate the error. As we can see in all six figures, the runtime of Preflow-Push is not stable at all. It can be much faster than other two algorithms and can be slower than other two algorithms. This is because the ping pong effect in Preflow-push algorithm. A small change in the number of edges can enhance the ping pong effect. In each figure above, we can see that the runtime of Ford Folkerson Algorithm and the Scaling Ford Folkerson Algorithm increase with the density of the data. The reason is that the complexity of all three algorithms is related to the number of edge in the graph. In fig.s 1(a) to 1(c), we increase the number of vertex from 40 to 80 in a fixed range of capacity between 10 and 100. We can see the run time of all algorithms increase with the number of vertex. In the figs.1 (d) to (f), we increase the capacity of each edge in the graphs by increasing the max value of capacity. We found that the run time of Ford-Folkerson Algorithm and the Scaling Ford-Rolkerson increase dramatically. The runtime range of the Preflow-Push Algorithm did not increase so much.

3.2 Input of Fixed Range datasets

The effects of number of edges on performance of Fixed Range is shown in Fig.2, we could find that both runtime of Ford Fulkerson and Scaling method increases when edges increases. It could be seen that Ford Fulkerson is slower than Scaling Ford Fulkerson. As the same as Randomdatasets, the Preflow push is completely not consistent with the theoretical running time.

Figure 3 shows the effects of number of edges per node

Table 1: Comparison of tested results with the given samples

	g1(Bipartite)	Ford Fulkerson	Scaling Ford Fulkerson	Preflow Push
Maxflow	150	150	150	150
	g2(Bipartite)	Ford Fulkerson	Scaling Ford Fulkerson	Preflow Push
Maxflow	898	898	898	898
	100v-5out-25min-200max(Fixed degree)	Ford Fulkerson	Scaling Ford Fulkerson	Preflow Push
Maxflow	517	517	517	517

Table 2: Average Runtime (50 times) of three proposed algorithms when dense change from 11 to 100 of a Random dataset: n=40, Cmax=100, Cmin =10.

dense(%)	FF	Scaling FF	Pre-Push	dense(%)	FF	Scaling FF	Pre-Push	dense(%)	FF	Scaling FF	Pre-Push
11	1	2	26	41	2	2	23	71	14	10	30
12	0	0	13	42	2	2	1	72	14	12	31
13	0	0	0	43	5	4	21	73	14	11	2
14	0	0	1	44	5	4	21	74	14	11	3
15	0	0	16	45	6	5	1	75	13	12	33
16	0	0	15	46	5	3	1	76	14	10	2
17	0	0	0	47	5	4	28	77	14	11	2
18	0	0	18	48	4	4	1	78	17	15	35
19	0	0	16	49	5	4	2	79	11	10	32
20	0	0	15	50	3	3	26	80	18	14	34
21	1	0	0	51	6	5	26	81	18	15	34
22	0	0	15	52	9	7	2	82	18	14	3
23	0	0	0	53	8	6	24	83	17	15	35
24	0	0	0	54	7	6	2	84	18	13	3
25	1	0	0	55	9	7	2	85	18	15	36
26	1	1	19	56	10	8	2	86	20	18	32
27	1	1	16	57	7	6	26	87	19	15	36
28	2	2	19	58	7	5	2	88	18	14	34
29	2	2	17	59	11	10	28	89	22	17	34
30	2	2	19	60	11	8	2	90	21	16	35
31	2	1	19	61	9	7	33	91	20	19	35
32	2	1	22	62	10	8	2	92	22	17	4
33	1	1	22	63	9	6	27	93	21	19	3
34	4	3	1	64	14	12	3	94	23	19	4
35	1	1	26	65	11	10	2	95	26	22	3
36	2	1	1	66	9	8	2	96	24	21	4
37	2	2	25	67	9	7	2	97	22	19	3
38	2	2	21	68	9	8	2	98	30	25	39
39	3	2	1	69	14	12	31	99	24	22	4
40	3	2	1	70	11	9	2	100	27	22	4

on runtime. It could be found that Preflow Push is still not consitent with the theoretical running time, Scaling Ford Fulkerson has the best runtime performance, Ford Fulkerson second, Prepush flow the last.

Figure 4 the effects of maximum capacity of edges on runtime. Obviously Scaling Ford Fulkerson has the best performace. As shown in the figure, the SFF (red line)even not change when C increases. This characterristic is very different from the Ford Fulkerson and Preflow Push. The reason is mainly because its runtime is O(n^2logC), which means the capacity has less effects compared with edges to Scaling Ford Fulkerson O(mC) and Preflow Push O(mn^2).

3.3 Input of Bipartite datasets

Figure 5 presents the effects of number of edges from source on runtime. we could find that Scaling Ford Fulkerson still has the fastest runtime, preflow push the last.

Figure 6 presents the maximum of capacity on performance. Still, capacity has less effects on Scaling Ford Fulkerson as the same as Fixed Range. The Preflow (blue line) presents not to be consistent with the theoretical runtime. Ford Fulkerson is faster than Preflow push, but less than Scaling Ford Fulkerson.

Probability's effects on runtime are discussed in Figure 7, we could find that Probablity has little effects on the runtime of Ford Fulkerson and Scaling Ford Fulkerson. But the Preflow push shows strong variations just as the other cases shown above.

3.4 Input of Mesh datasets

Figure 8 shows the effects of number of columns on performance of Mesh datasets.The rows are constant 10. The number of Columns increases from 10 to 50. We could find that the Scaling Ford fulkerson runs the fastest, Ford Fulkerson second, and Preflow Push the last. However, it could be seen that the Preflow Push does not have any variations as the above input.

In Figure 9, the effects of number of rows are discussed. The columns are set to be 10. We could find that when the number of columns increased from 10 to 50, the runtime of Scaling Ford Fulkerson has the fastest runtime, Ford Fulkerson Second, and the Preflow Push the last!

3.5 Effects of Input datasets on runtime

For the Augmenting algorithms (Ford Fulkerson and Scaling Ford Fulkerson), we found no exception between this two

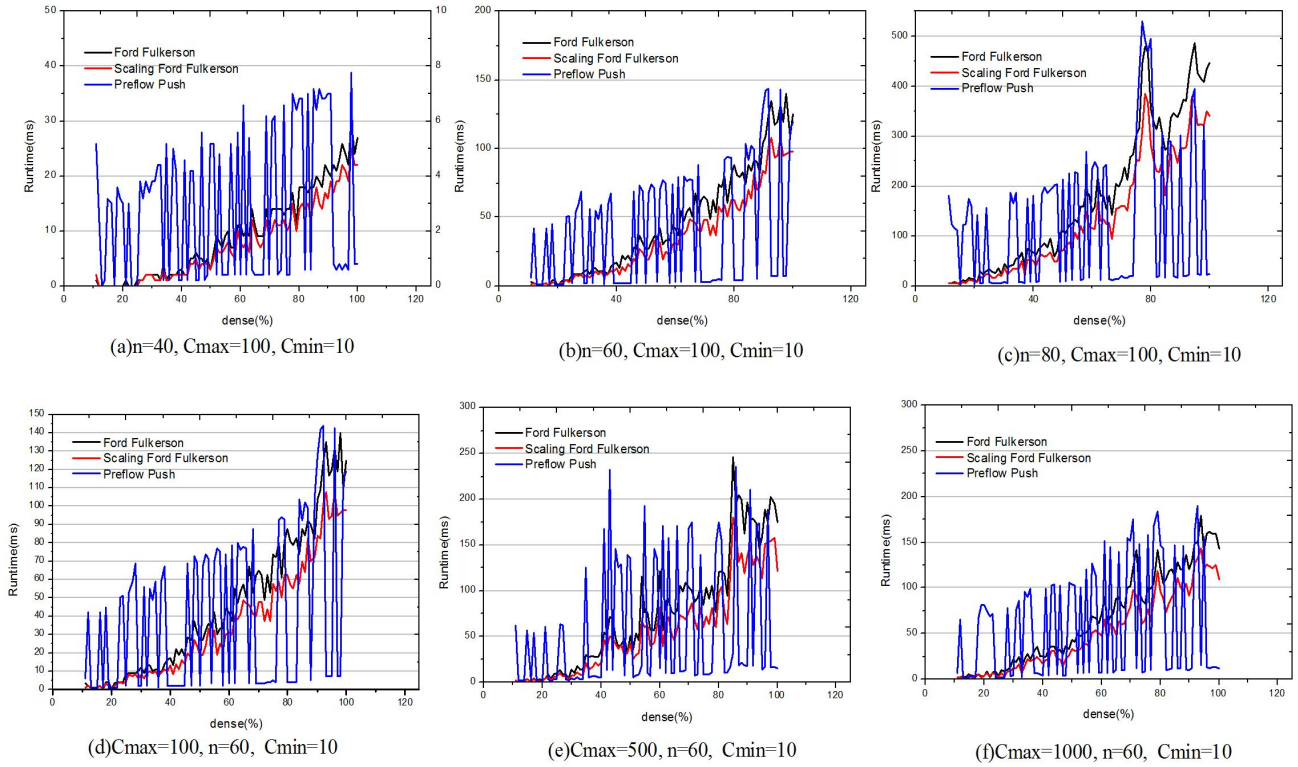

Figure 1: Runtime as a function of density of edges(%) when inputs are random datasets with different parameters.

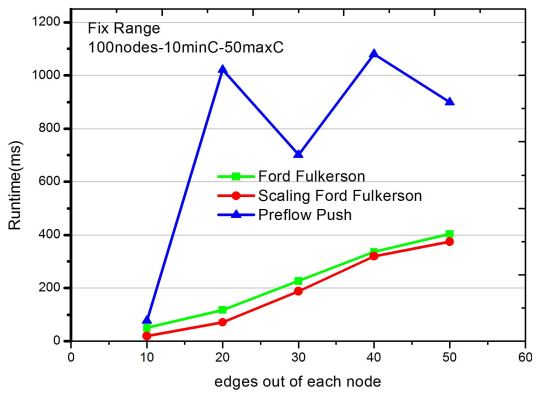

Figure 2: Effects of number of edges on performance of Fixed Range datasets.

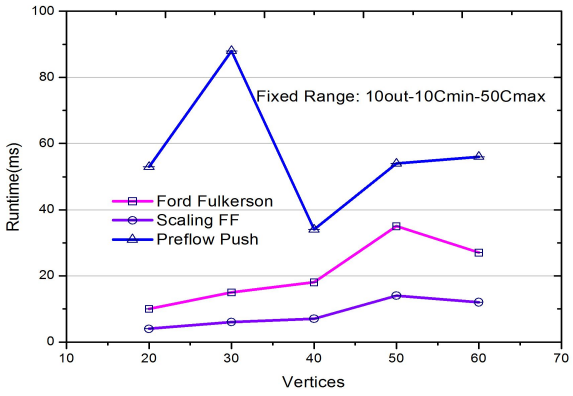

Figure 3: Effects of number of nodes on performance of Fixed Range datasets.

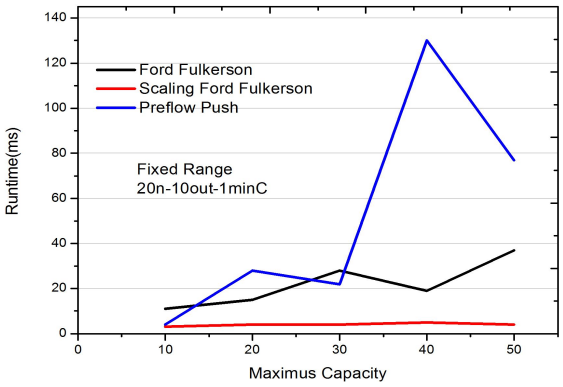

Figure 4: Effects of Maximum Capacity on performance of Fixed Range datasets.

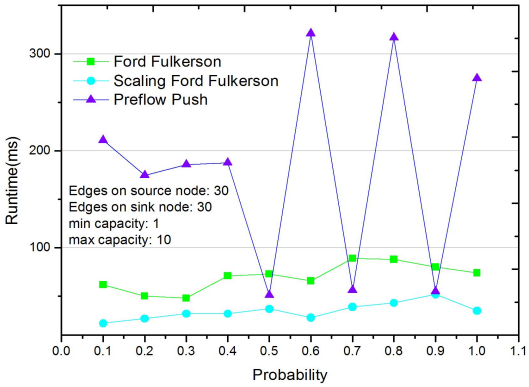

Figure 7: Effects of probability of edges on perfomance of Bipartite datasets.

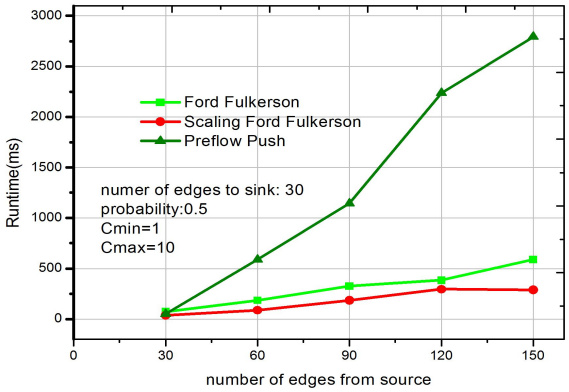

Figure 5: Effects of number of edges on source on performance of Bipartite datasets.

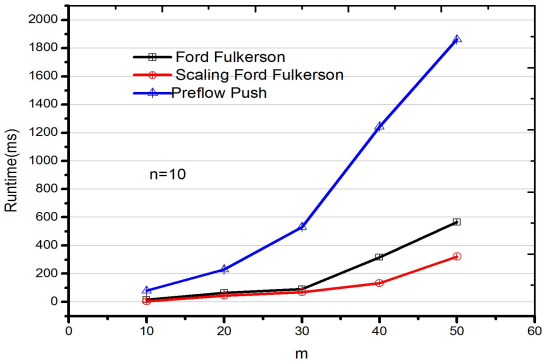

Figure 8: Effects of number of columns on performance of Mesh dastasets

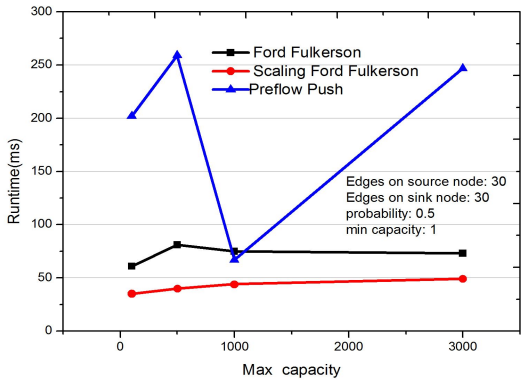

Figure 6: Effects of maximum capacity on performance of Bipartite datasets.

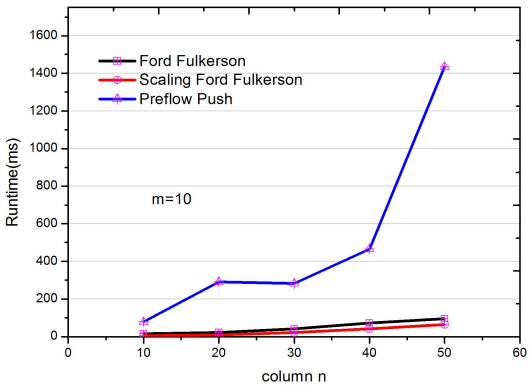

Figure 9: Effects of number of rows on performance of Mesh dastasets

algorithms: Scaling Ford Fulkerson was always faster than Ford Fulkerson no matter what dataset we chose. Because the augmention of Ford fulkerson depends on the bottleneck. Bit Scaling method reduce this problem by lower the upper bound to $O(m^2C)$.

For the Preflow Push algorithms, it could have "Ping Pong Effects". Therefore, it is extremely not consistent with its upper bound $O(n^2m)$. So it is hard to conclude that which dataset it is good for.

4. FUTURE WORK

We would like to try to search a way to change the way bottleneck is found in the algorithm. As of now we find the minimum bottleneck that we can send across S-T path. This approach takes considerable amount of time, as sometime bottleneck chosen by the algorithm can be small, it may take many iterations to find the maximum flow. Instead of this, we would want to find the maximum bottleneck that can be send across the S-T path, so that the algorithm can complete in minimum step possible, which will improve the efficiency considerably. The main challenge in changing this aspect is running memory consumption. We would have to come up with an intelligent idea to use minimum memory possible and yet find the maximum bottleneck.

In the preflow-push algorithm, the method to find active nodes is executed in many times. So it is meaningful to improve the efficiency of this method. In the current implementation of the algorithm, we iterate over the all vertices in the graph to find an active node. This will take O(n) time in the worst case. We can use a queue to improve the efficiency of this method. We store the active nodes in the graph in a queue. And we will choose the first vertex in the queue and update the queue. The algorithm will terminate when the queue is empty. Another part we want to improve is to use TreeMap instead of HashMap. The HashMap is not stable in iteration because the elements inside are not ordered. And TreeMap is also fast for get and put operation. We are also interesting to improve the stability of the algorithm and avoid the ping pong effect.

5. DIVISION OF LABOR

Codes: Ford Fulkerson algorithm is implemented and javadoc made by Paranjit Singh. Scaling Ford-Fulkerson is implemented by Zheng Zhou. Preflow Push algorithm is implemented by Pai Zhang.

Slides: The slides shown in class: the part of Introduction and Ford Fulkerson are developed by Paranjit Singh. The Scaling Ford Fulkerson, Results and Figures in slides are made and plotted by Zheng Zhou. Pai Zhang did the Preflow Push and the conclusions part. The slides is presented by Paranjit Signh.

Report: Paranjit Singh wrote the part of 2.11 Ford Fulkerson and 4 first paragraph of Future work, and 6.1 Lessons Learned.

Zheng Zhou wrote the 2.12 Scaling Ford-Fulkerson, 2.2 Codes Validations, and Results 3.2 to 3.5. Data, Tables and Figures in this report are drawn by Zheng Zhou.

Pai Zhang wrote the Introducion part and 2.13 Preflow Push, 3.1 Random Datasets. 4.2 Future work, 6.2 lessons learned and Reference.

6. LESSONS LEARNED

6.1 Augmenting algorithms

We learned the reason why Scaling Ford Fulkerson, algorithm was better than Ford Fulkerson, as it specifically targeted the way bottleneck is selected in each iteration. However, we learned that even Scaling Ford Fulkerson is not completely optimized when finding the bottleneck. As it wastes too many iteration, finding a bottleneck in S-T flow, which *doesnt* exist. We learned the different aspect of usability of BFS and DFS, how each approach is suitable in some aspect, and no one is a clear winner. Similarly, we need to see the network we are trying to solve before selecting an algorithm, as density, edge capacity etc., play a significant role in running time for each algorithm.

6.2 Pushflow push algorithms

In this implementation of Preflow-Push Algorithm, we found that the performance of this algorithm is not as same as we expected. It is not stable especially in random data set. In random data sets, a small change in edges can cause the dramatic change in performance because of the ping pong effect. So Preflow Push Algorithm is not recommended in random graphs. If the graph is stable and does not have ping pong effect, Preflow Push algorithm will be a very competitive algorithm. We also learned that the data structure and the details of implementation such as eliminating unnecessary iterations are also critical for the performance of the algorithm.

References

Genon, D., Velghe, V., DEVILLE, Y., AUBRY, F., DELVENNE, J. C. Study and analysis of network flows.