Empirical Study On Network Flow Algorithms

TCSS553 ADVANCED ALGORITHMS

Resham Ahluwalia
University of Washington
resh@uw.edu

Sonal Goswami
University of Washington
sonal119@uw.edu

Karthik Kolathumani
University of Washington
kkarthik@uw.edu

I. INTRODUCTION

Problem Description: In this project, we are conducting an empirical study to see which of the network flow algorithms is better than the others.

Our goal in this project is to see if we can figure out for what kinds off graphs one algorithm des better than the others.
We will perform an empirical study and compare the following Network Flow Algorithms: Ford Fulkerson Algorithm, Scaling Ford Fulkerson Algorithm and Preflow Push Algorithm

A flow network is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow. A network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

II. METHODOLOGY

We were provided with graph generation code for four types of graph: Mesh Graph, Random Graph, Fixed Degree Graph and Bipartite Graph.

We created generic flow graph class which takes simple graph as an input and convert it into flow graph with flow and capacities on each edge. This generic class provides method for increasing flow on edges which takes care of adding/removing backward edges, adjusting residual capacity of forward edges and updating excess on involved nodes. It also exposes methods like incrementing height on vertex, getting residual capacity on edges, getting neighboring vertices as required by flow algorithms. Then, we proceeded with implementing different algorithms using this class.

III. ALGORITHMS

1. *METHOD: FORD FULKERSON*

The Ford Fulkerson method follows a simple implementation of three basic functions, a graph traversal algorithm to find the s-t path to set, a function to form the residual graphs with the residual capacities and a function to augment the flow. The graph traversal algorithm used in our code is DFS. We are using a Hashtable with an adjacency list representation to form a residual graph. Our algorithms initialize the residual graph with a flow value of 0. While there is a path from source to sink, the code tries to augment the flow with the bottleneck value found. It repeats the process until no augmenting paths are found.

2. ALGORITHM: SCALING FORD FULKERSON

The Scaling Max-Flow Algorithm, also known as the Scaling Ford-Fulkerson Algorithm is an extension of the Ford Fulkerson Algorithm. We improvise the Ford Fulkerson Algorithm's maximum-flow value and runtime by the Scaling Algorithm. We need to devise the algorithm in such a way that can find the largest value of a bottleneck in an s-t flow to maximize the flow of the graph. So, we introduce a scaling parameter Δ, which will find paths having bottleneck capacity of at least Δ. We generate the subset of the residual graph Gf (Δ), containing the set of edges in the residual graph having residual capacity of at least Δ. In the ScallingFordFulkerson.java code, we have used linked list to store the s-t flow.

3. ALGORITHM: PRE FLOW PUSH

For Preflow Push algorithm we started with maintaining a list of all vertices with positive excess. Our code takes a vertex out of this list and check all its neighbors' height label, if it finds any vertex with height less than the chosen vertex it performs the push operation else it performs the relabel operation. Having adjacency list representation makes easy to access the neighboring vertices. After performing the operation, in case of push it checks the two involved vertices for positive excess and them to list if needed. In case of relabel it adds the vertices back to the list. Algorithm stops when the list is empty i.e. there is no vertex left with positive excess.

IV. I NPUT GRAPHS

For each type of input graph, we varied the parameters of graph generation one by one. This includes varying different components in each type of graph. This approach was chosen to make sure the results that we get will tell us more about the dependencies of these factors on each of our individual

algorithm. Each of these input instances were then tested to observe the running times. The running time noted were an average of 5 such individual observations. All running times were captured in milliseconds.

1. GRAPH: MESH GRAPH

The generation of mesh graph provided to us depend upon four parameters: the number of rows, the number of columns, the capacity (the capacity of each edge if constant capacity else maximum capacity) and the value of cc flag. To generate our input instances for mesh graph we varied the number of rows, number of columns and capacity one by one for both values of constant capacity flag. We kept the number of columns at 100, maximum capacity at 150, constant capacity flag as false and varied number of rows from 50 to 230. Then, we kept number of rows fixed at 100 and varied number of columns from 50 to 230. We also tried varying row experiment with maximum capacity set to 50 and 500. Then, we kept both number of rows and columns fixed at 50 and varied capacity from 50 to 140. All these experiments were done with constant capacity flag set to false. Then, we did experiments with constant capacity flag to true. For this, we set the number of columns as 200, capacity as 150 and varied number of rows from 100 to 280. Then, we fixed number of rows as 200 and varied number of columns from 100 to 280. Finally, we kept number of rows and number of columns as 200 and varied capacity from 150 to 600. All these experiments were done with constant capacity flag set to false. Then, we did experiments with constant capacity flag to true. For this, we set the number of columns as 200, capacity as 150 and varied number of rows from 100 to 280. Then, we fixed number of rows as 200 and varied number of columns from 100 to 280. Finally, we kept number of rows and number of columns as 200 and varied capacity from 150 to 600.

2. GRAPH: BIPARTITE GRAPH

The Bipartite Graph is a graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V. In our project study, we include the Source Node S and the Sink Node T for the Bipartite graph to show and calculate the flow of the network in the graph from the source to the sink. We have performed the experiment by varying the Number of Nodes on the Left-Hand side (U) or the Source side and the Number of Nodes on the Right-Hand side (V) or the Sink side, the Maximum Capacity values of the Graph and the Maximum Probability of the Graph. The Range that we varied for our experiment to observer the different results and patterns are as below: We kept the Number of Nodes on the Source Side: 200, Number of Nodes on the Sink side: 250, Minimum Capacity: 100, Maximum Capacity: 150, and the Probability: 0.25. In our first dataset, we varied the Probability in the range of 0.1 to 0.7 in the increment of 0.05 and produced the results. Then we varied the number of nodes on the source side from 155 to 500 trying out different combinations with the number of nodes on the Sink side in the range on 100 to 1000. We tried different combinations off keeping the number of nodes on the source side equal, less and more than the number of nodes on the right-hand side to observe different patterns. Then in the next dataset, we have altered the Maximum Capacity to see how much does the Maximum Capacity affect the 3 algorithms' performance on the Bipartite graph. We varied the range in increments of 100. We kept the range of the Maximum Capacity from 155 to 1000.

3. GRAPH: RANDOM GRAPH

The generation of the Random graphs was also based on three parameters: the number of vertices, the density percentage or the number of edges, and the capacity range. To generate our input instances for random graph, we varied the number of vertices in a range of 100 to 1000 with an increment of 100 vertices with each new observation. The density was in this case was set to a constant of 30%. This doesn't mean that we tried to keep the number of edges constant. We followed this approach to make sure the running time of the algorithms are less dependent on the density being less variable parameter. Next, we varied the density percentage from 0 to 100 with an increment of 10% in each successive observation. We kept the number of vertices and capacity range constant in this case. In our third kind of observation we varied the capacity range between 0 to 10000 with an increment of 1000 in each successive iteration.

4. GRAPH: FIXED EDGES GRAPH

The Fixed edges graphs also followed a similar three variable approach. The generation of the Random graphs was also based on three parameters: the number of vertices, the fixed number of edges, and the capacity range. The range for the vertices and degree of edges range between 1000 to 2000 and 50 to 100 respectively. The range of the capacities was taken between 1000 and 10000.

V. RESULTS

1. Graph: Mesh

Some interesting observations

In case of constant capacity preflow performs best and in case of maximum capacity mesh graph preflow performs worst. For small range of capacity, the preflow push algorithm performs worst but as we increase the value of maximum capacity the ford Fulkerson algorithm performs worst. In maximum capacity mesh graph the runtime of preflow doesn't strictly increase with increase in number of rows/columns.

Constant Capacity Mesh Graph

Varying capacity on constant capacity Mesh graph doesn't change running time of any algorithm. This is because all the flow push in all 3 algorithms are always equal to the constant capacity and all the backward edges made during algorithm are of this capacity. Hence the actual value of this capacity doesn't alter the execution and hence running time of these algorithms.

Preflow push algorithm runs fastest on constant capacity mesh graph greatly outperforming Ford Fulkerson and Scaling Ford Fulkerson. Increasing number of rows/columns increases running time of all three algorithms but increase in running time of Preflow push algorithm is much slower compared to increase in running time of other two algorithms.

Maximum Capacity Mesh Graph

Varying capacity on maximum capacity mesh graph gives some interesting results. With increase in capacity running time of Ford Fulkerson increases, as expected, since Ford Fulkerson's running time is bound by $O(C)$. Running time of Scaling Ford Fulkerson doesn't change much with increase in capacity, again as expected, since running time in this case is bound by $O(\log C)$. Running time of Preflow push algorithm changes erratically with change in capacity, as capacity doesn't play a role in bounding running time for preflow algorithm.

Scaling Ford Fulkerson performs best in maximum capacity mesh graph, followed by Ford Fulkerson and Preflow push algorithm performs worst. However, on increasing the value of maximum capacity Preflow push algorithm outperforms Ford Fulkerson. Scaling Ford Fulkerson still performed best as it's running time is much less sensitive to capacity as compared to Ford Fulkerson.

Increasing number of rows and columns increases the running time of all Ford Fulkerson and Scaling Ford Fulkerson algorithms. Running time of Preflow push algorithm also generally increases with increase in number of rows and columns however it is much less regular pattern with running time sometimes decreasing with more number of rows/columns.

2.Graph: Bipartite

Some interesting observations

By varying the Maximum Probability, we find that the Ford Fulkerson algorithm takes significantly longer execution time than Scaling and Pre-Flow as the probability increases.

By varying the number of vertices in the graph we observed that the Bipartite produces an interesting graph for number of nodes on the source side and the number of nodes on the sink-side. We observer that the runtime is at the peak when the number of nodes on the source side is equal to the number of nodes on the sink side or the difference between the node values is the least. It is faster for nodes having large differences in the values (# source nodes to # sink nodes). Ford Fulkerson algorithm takes significantly longer execution time than Scaling and Pre-Flow.

By varying the Maximum Capacity of the Edges in the Graph we observe that changing maximum capacity of the edges does not produce any significant change in the runtime of the Scaling and Pre-flow algorithms. However, Ford Fulkerson algorithm's runtime increases as the capacity increases up to a certain limit and then the range almost performs fine for larger capacities.

3.Graph: Random

Some interesting observations

In case of Random graphs, we saw that with increase in the density percentage, Ford Fulkerson and Scaling Ford Fulkerson performance was comparable. Scaling Ford Fulkerson performed better than Ford Fulkerson. We suppose that the reason is similar to what we learnt in the textbook CLRS[2] i.e. the running time of both the algorithms are dependent on the number of edges. The reason why the scaling algorithm performs better attributes to the fact that dependency on the capacity is reduced by a factor of $log(C)$ where C is the max capacity edge. Preflow algorithm performs the best as it depends on the number of vertices.

A similar observation was made by varying the number of vertices – Ford Fulkerson and Scaling Ford Fulkerson have similar performance. Preflow algorithm's runtime increased with increase in the number of vertices. The Scaling and the Ford Fulkerson algorithm performed almost equally.

By varying the capacity, we observed Ford Fulkerson performing the worst being dependent on input capacity for its running time. Scaling Ford Fulkerson and Preflow perform almost equally in this case. In case of scaling the runtime is reduced since it chooses the max residual capacity while finding the augmenting path. Preflow has no dependencies on the capacity and hence performs constantly with increase in the capacity range.

By varying the edges, number of vertices and capacity range, we calculated the average running time of each of the three algorithms and plotted their performance on a graph. It is important to note that by increasing the number of vertices will also result some increase in the number of edges. Even though this increase in edges is not as significant as we changed for our edge based running time analysis, but it becomes important to note that by increasing the number of vertices doesn't alone account to the change in the running time. Figure 7 to figure 9 in the appendix shows the graphs obtained by these observations.

4.Graph: Fixed Edges

Some interesting observations

In Fixed edges, we observed that the Ford Fulkerson and Scaling Ford Fulkerson both perform very similarly. This might be because we learnt in the class that both the Ford Fulkerson and Scaling Ford Fulkerson depend on edges. In case of Ford Fulkerson each time finding the augmenting path results in a runtime of $O(E)$. Which is also the case in the Scaling Ford Fulkerson with slightly different approach. So, they both perform equally. In case of Pre - Flow as it depends on vertices rather than edges we can see that it performs almost constantly with the increase in the edges.

With the increase in the number of vertices we can except a slight increase in the number of edges as well. The number of edges also increased with the increase in the number of vertices in this case. In case of Preflow, we saw a slight increase with

each observation. We observed for higher number of vertices the runtime of Preflow increased slowly.

By performing the analysis for the fixed degree graphs, we got similar results as that of random graphs. The only difference here is we can articulate the dependencies of the three algorithms based on increase in the number of edges by keeping the vertices and capacity range constant. Figure 10 to figure 12 in the appendix shows the graphs obtained by these observations.

VI. Future Work

For future work we would like to implement different strategies for picking vertices with positive excess in case of preflow algorithm, for e.g. Picking largest height vertex or using FIFO strategy. Also, we would like to parallelize preflow algorithm. For ford Fulkerson algorithm also, we would use different strategy for selecting augmenting path for e.g. Picking augmenting path with least number of edges. Also, we will compare the results we obtained from different type of machines configurations for e.g. machine with vector operation support. We would also like to study memory utilization of different algorithm.

We feel if there are unexpected lumps in the graphs, it can be due to the fact we used three different machines to perform the analysis. We will perform the analysis on a single machine to get more accurate results.

VII. References

[1] Algorithm Design- Jon Kleinberg, Eva Tardos
[2] Introduction to Algorithms – CLRS

I. APPENDIX A: INPUT GRAPHS

1. Mesh Graph

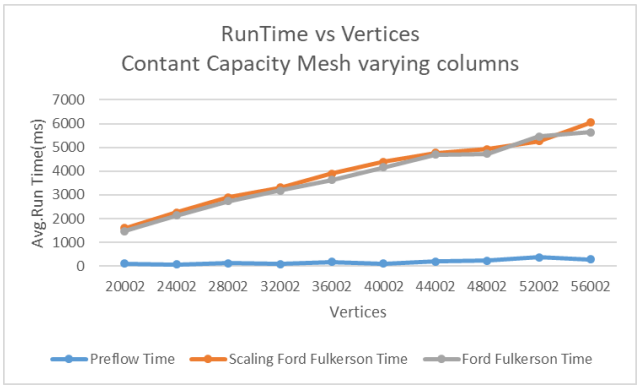

Figure 1

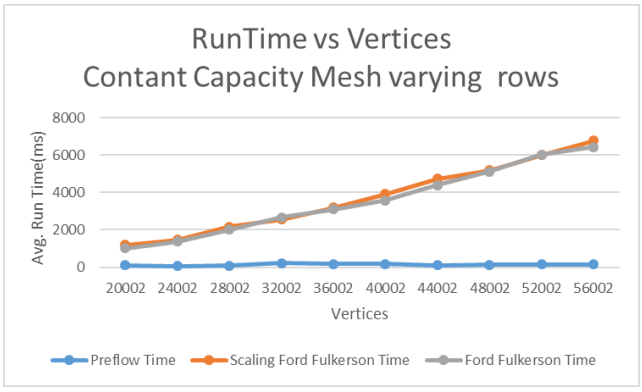

Figure 2

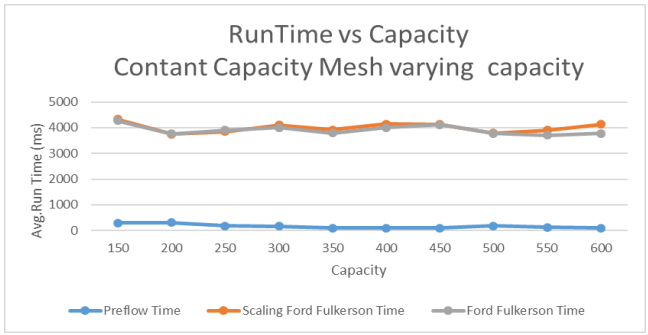

Figure 3

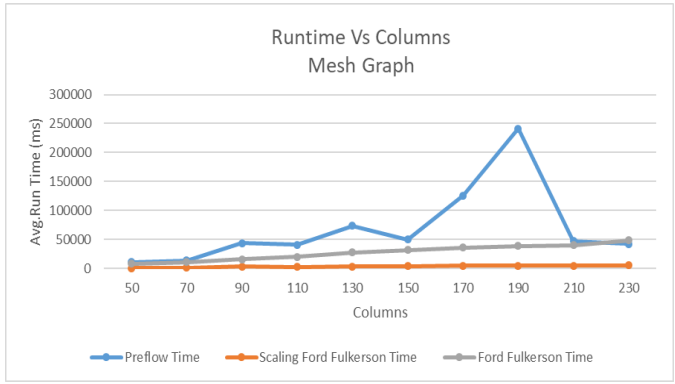

Figure 4

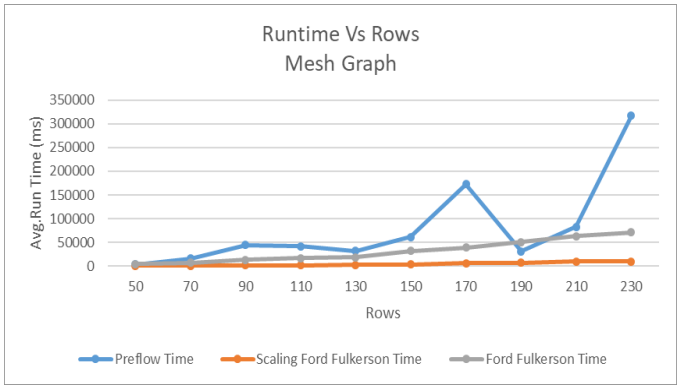

Figure 5

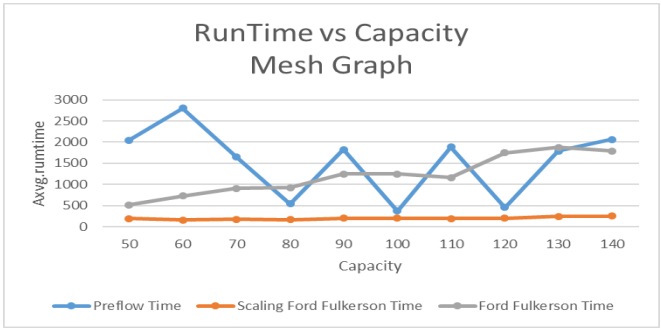

Figure 6

2. Random Graphs

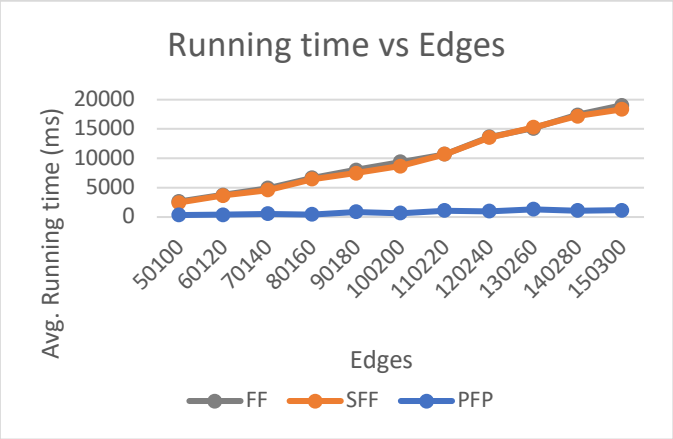

Figure 7

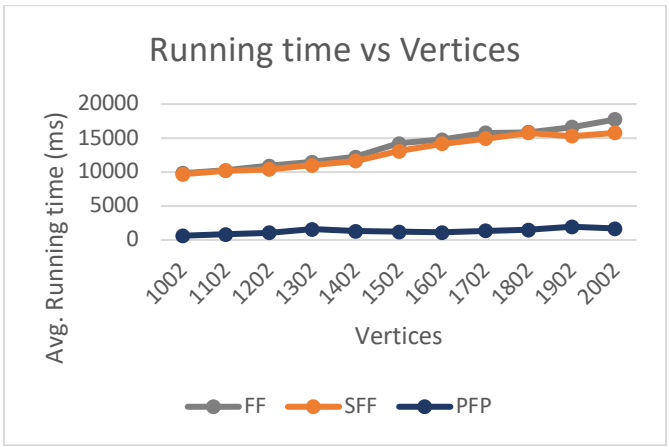

Figure 8

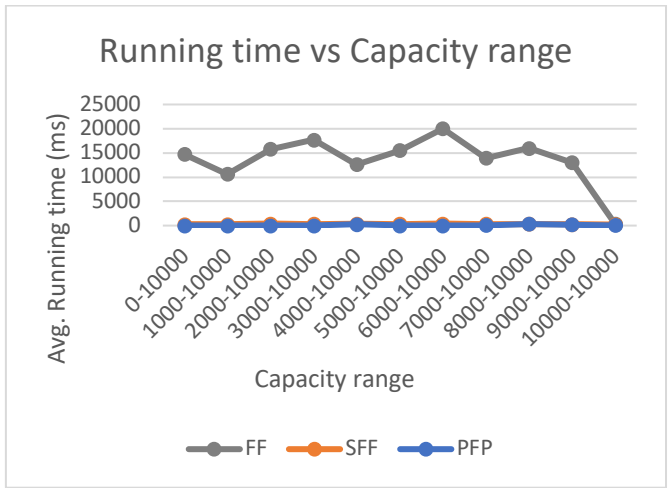

Figure 9

4. Fixed Edge Graphs

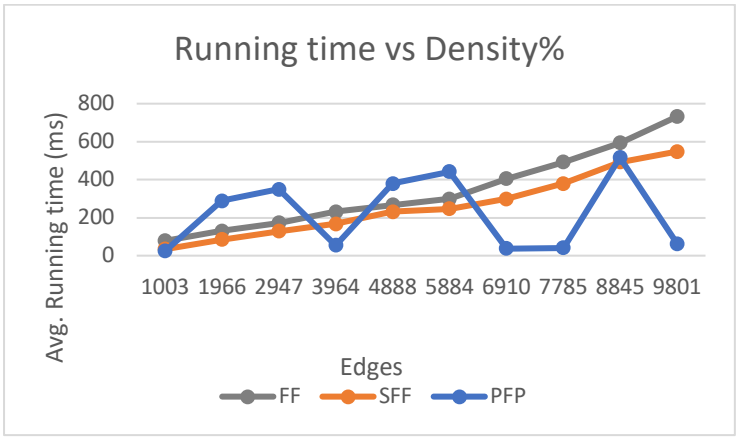

Figure 10

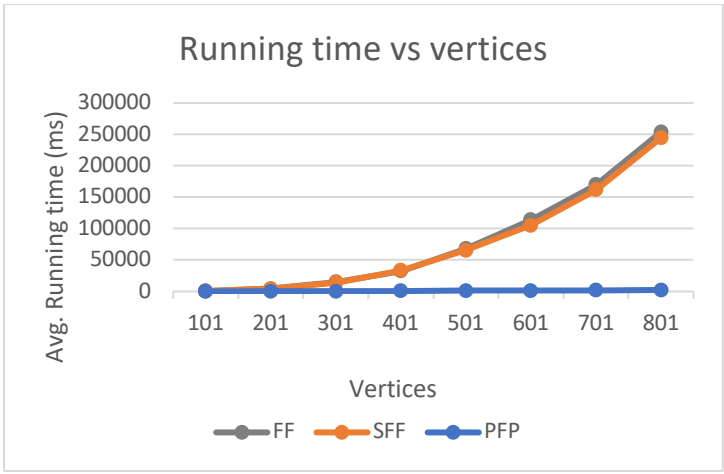

Figure 11

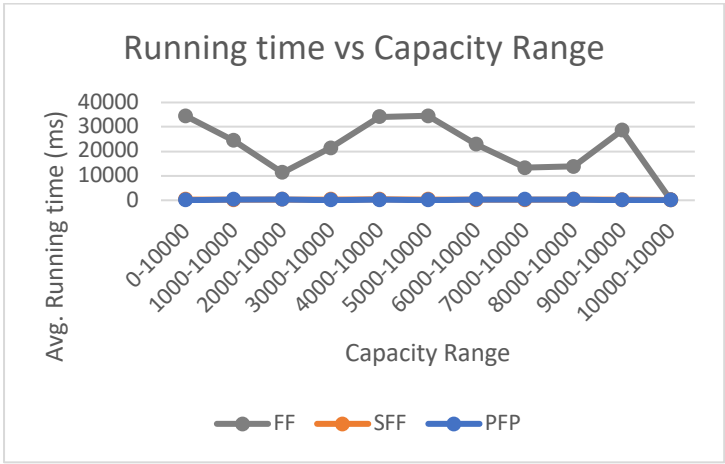

Figure 12

4. Bipartite Graph

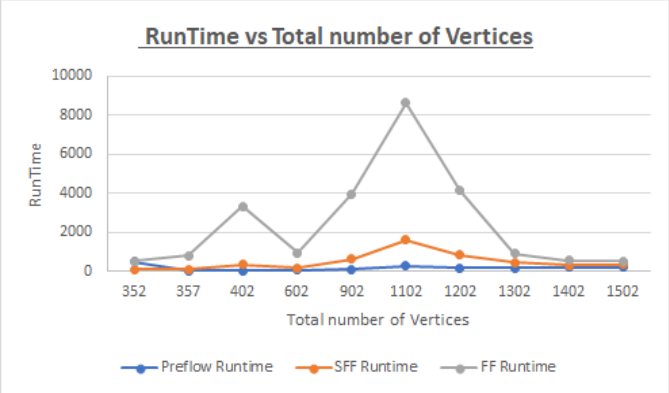

Figure 13

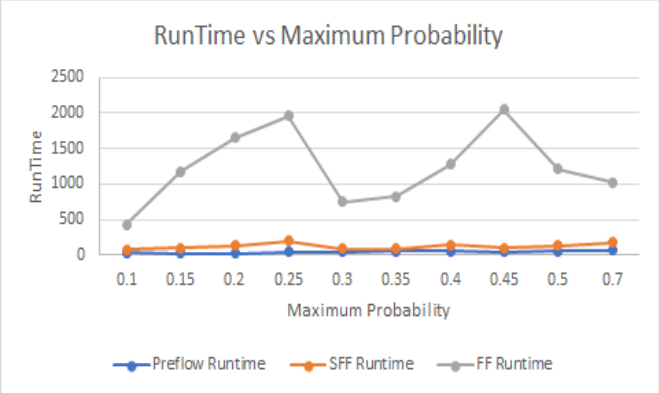

Figure 14

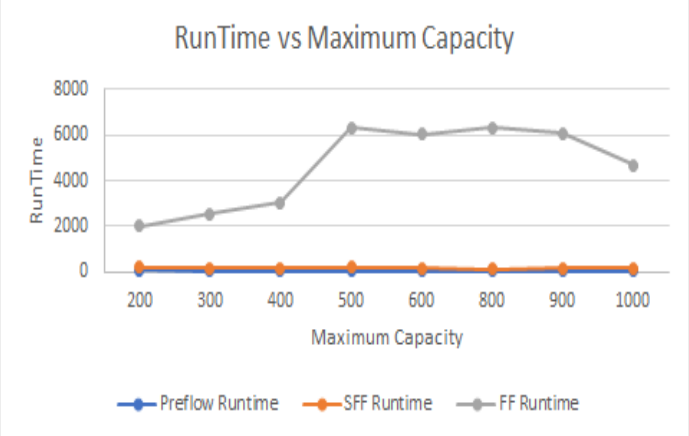

Figure 15