Empirical Study on Network Flow

Swati Garg
University of Washington
Tacoma
swatig1@uw.edu

Surabhi Agrawal
University of Washington
Tacoma
agraws@uw.edu

Jessica Goel
University of Washington
Tacoma
goelj@uw.edu

ABSTRACT

In this paper, we conducted an empirical study to see which of the network flow algorithms perform better for different types of input graphs. We analyzed the Ford-Fulkerson algorithm, Ford-Fulkerson with scaling algorithm and Preflow-Push algorithm on four different types of graphs: Bipartite, Mesh, Random and Fixed degree, and calculated the max flow of the network.

KEYWORDS USED

Graphs; Network Flow; Time complexity; Algorithms; breadth first search; Runtime; FordFulkerson; Scaling FordFulkerson; Preflow-Push

INTRODUCTION

The network flow problem consists of a directed graph which has a capacity associated with every edge. The amount of flow on every edge has the restriction that it can never exceed the capacity of the edge and is always positive i.e. 0 <= f <= C. There are source and sink nodes in the network flow graph. These nodes do not have the restrictions as the source node does not have any incoming edges whereas the sink does not have any outgoing edges. A network flow problem is required to calculate a maximum flow that can be pushed through the network. This flow has the following constraints:

1. Capacity constraints
2. Flow Conservations

The most popular algorithms that are used to calculate the max-flow of any network are the Ford-Fulkerson, Ford-Fulkerson with scaling and Preflow-Push algorithms.

The network flow algorithms are intensively used in day to day applications such as Airline scheduling, Image segmentation, Electrical circuits and more.

We performed an empirical study to analyze the performance of the three algorithms. This study was aimed at understanding the behavior of these algorithms under different network topologies/structure. From this study, we concluded that the performance of any algorithm is affected by the number of vertices in the network, number of edges, density, maximum capacity and the data structures used.

METHODOLOGY

1. Algorithms:

Ford-Fulkerson Algorithm:

This algorithm computes the maximum flow of the network. As long as there is a path between source to sink with available capacity, the flow can be sent through the network. This flow is calculated by summing the bottlenecks on all the augmenting paths.

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

The runtime of the algorithm is O(mC), where m is number of edges and C is the total capacity coming out of the source.

Ford-Fulkerson with scaling:

The biggest challenge for FordFulkerson is to find a good augmenting path. The number of iterations can be increased if the path chosen has the minimum capacity. This can be overcome by choosing the path with maximized bottleneck capacity.

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

```
            Update f to be f' and update $G_f(\Delta)$
        EndWhile
        $\Delta = \Delta/2$
    EndWhile
    Return f
```

The runtime of the algorithm is O($m^2 log_2 C$), where m is the number of edges and C is the total capacity coming out of the source.

Preflow-Push:

The FordFulkerson and scaling Ford-Fulkerson algorithms are based on augmenting paths in the graph to get the max flow. The preflow Push algorithm does not depend on path augmentation. It maintains the pre-flow on the edges and gradually converts them into maxflow by moving flow to neighboring vertices using push operation.

```
Preflow-Push
  Initially h(v) = 0 for all v≠ s and h(s) = n and
    f(e) = $c_e$ for all e = (s, v) and f(e) = 0 for all other edges
  While there is a node v≠ t with excess $e_f(v) > 0$
        Let v be the node with excess
        If there is w such that push(f, h, v, w) can be applied then
              Push(f, h, u, w)
        Else
              Relabel(f, h, v)
        Endif
  EndWhile
  Return (f)
```

The runtime of the algorithm is O($n^2 m$), where n is the number of nodes and m is the number of edges.

2. The Experiment:

To perform the empirical study, we wrote the program tcss543.java. This is a main program that reads the input.txt (graph file created from the graph generation codes provided) file from the console. It runs the network flow algorithms. The main executes each algorithm thrice and calculates the average of the time taken by each run. We are doing this to ensure accuracy of the time calculation.

We have created different java classes for each algorithm implementation (Ford-Fulkerson, Ford-Fulkerson with scaling, Preflow-Push). These programs take the input file, load the graph and return the max-flow of the graph represented in the input file.

After implementing the main network flow algorithms, and testing them on small inputs, we planned to test them on larger graphs. For this, we first decided that we were going to test the performance of these algorithms by varying one parameter of the graph and keeping the others constant. Eg: First, varying the number of nodes and keeping the number of edges and maximum capacity constant. Then varying the number of edges and capacities in turn. This way we would know which parameter affected which graph the most. We also planned to analyze all the data obtained by varying different parameters together and see which algorithm performed best for a particular type of graph. We generated the input files using the graph generation code provided, according to the decided variations in graph parameters (i.e number of nodes, edges and maximum capacity).

We also created an automation code for testing the code with the multiple input files and to capture the time taken by each algorithm. This program picks up all the files from a folder, executes each algorithm thrice, captures the average time taken by each algorithm and then outputs the result in a .cls file.

After running the experiment on all test files created for all four types of graphs, we generated the graphs displaying the behavior of each algorithm on various input data. We analyzed these graphs together and reached a few conclusions.

3. Data for Test Files:

For the analysis of the empirical formula we are given four graphs. We generated test files for each type by varying number of nodes, edges and capacities and then generated the files using the graph generation code provided. Below are the range of graph sizes used:

1: **Bipartite Graph**:

This is the graph whose vertices can be divided into two sets. One set on the source side and the other on the sink side. Every edge from one set joins the other set.

We categorized the files based on the following:

1. Keeping number of nodes and range of capacities constant and varying the number of edges:

   Number of nodes at source side: 100

   Number of nodes at sink side: 100

   Min-Max capacity: 1-50

   Probability: 0.1 – 1
2. Keeping probability of forming edges and capacity ranges constant and varying the number of nodes:

   Number of nodes at source: 10-150

   Number of nodes at sink: 10-150

   Min-Max capacity: 1-50

   Probability: 0.5

3. Keeping the number of nodes and edges constant and varying the capacity ranges:

   Number of nodes at source: 100

   Number of nodes at sink: 100

   Max-capacity: 50 -20000

   Probability: 0.5

2: **Mesh Graph:**

Mesh graphs have edges from s to each node in the first column, from each internal node to the node on its right, both ways between every internal node and the nodes above and below it, and from the last column nodes to the sink.

We categorized the files based on the following:

1. Keeping the max capacity constant and varying the number of nodes (as we vary the number of nodes, number of edges also change):

   Number of rows: 5-80

   Number of columns: 4-80

   Max capacity: 50

2. Keeping the number of nodes and edges constant and varying the max capacity:

   Number of nodes at rows: 20

   Number of nodes at columns: 10

   Max capacity: 50-10000 (we keep the capacities constant in alternate files by setting the –cc flag)

3: **Random Graph:**

This graph is generated randomly using the number of vertices, max capacity and the density of the graph.

We categorized the files based on the following:

1. Keeping number of nodes and capacity range constant and varying the number of edges:

   Number of nodes: 100

   Min-Max capacity: 1-50

   Density: 10-100 %

2. Keeping density of graph (edges) and capacity range constant and varying the number of nodes:

   Number of nodes: 20-200

   Min-Max capacity: 1-50

   Density: 30%

3. Keeping the number of nodes and density (edges) constant and varying the capacity range:

   Number of nodes: 100

   Min capacity: 1-5000

   Max capacity: 50- 20000

   Density: 30%

4: **Fixed Degree**:

This graph has a fixed degree of edges coming out of each node.

We categorized the files based on the following:

1. Keeping number of nodes and capacity range constant and varying the number of edges:

   Number of nodes: 100

   Min-Max capacity: 1-50

   No of edges out of each node: 10-96

2. Keeping number of edges and capacity range constant and varying the number of nodes:

   Number of nodes: 40-160

   Min-Max capacity: 1-50

   No of edges out of each node: 30

3. Keeping the number of nodes and edges constant and varying the capacity range:

   Number of nodes: 100

   Min capacity: 1-5000

   Max capacity: 50-20000

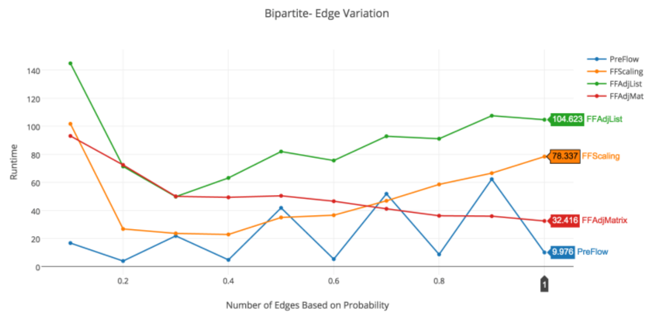
   No of edges out of each node: 30

4. Algorithm Analysis:

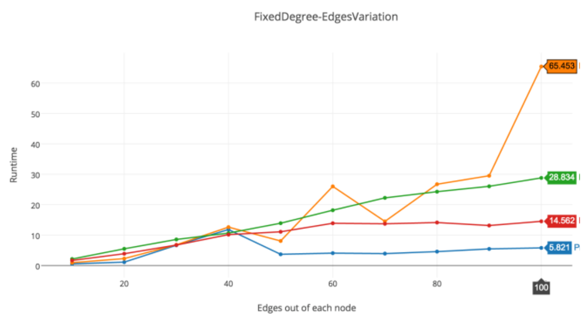
After generating all the input files as per the data given above, we generated the graphs, with runtime on the Y-axis and number of nodes, number of edges or max capacity on the X-Axis, and analyzed them based on common parameters.

1. Varying number of edges:

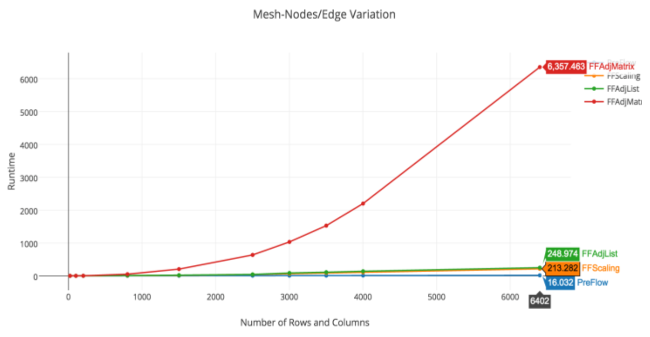
   Ford-Fulkerson with scaling generally performs the worst because it is dependent on the number of edges. The runtime of the FordFulkerson may be higher if it chooses bad augmenting path.



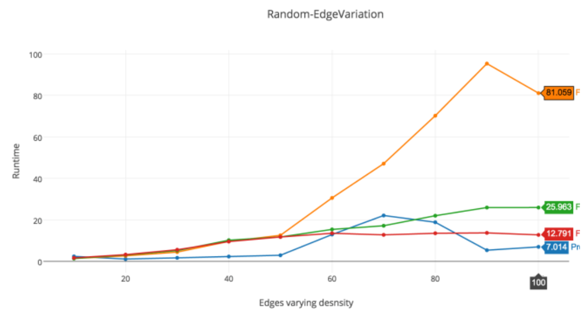
Bipartite graph-Edge Variation



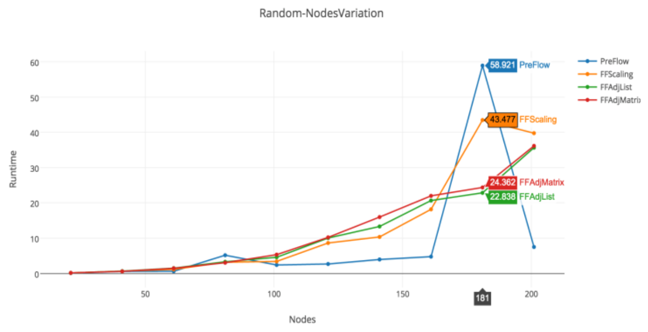
Fixed degree-Edge Variation



Mesh Graph-Nodes and Edge Variation



Random Graph-Edge Variation



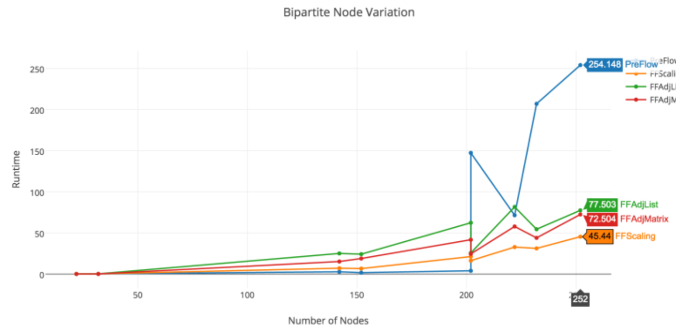
Random Graph-Nodes Variation

2. Varying nodes:

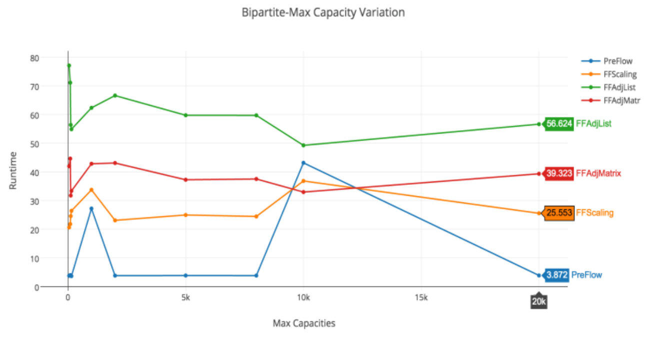
   We observe that after the certain number of nodes, the performance of preflow-push algorithm decreases considerably.

3. Varying Capacities

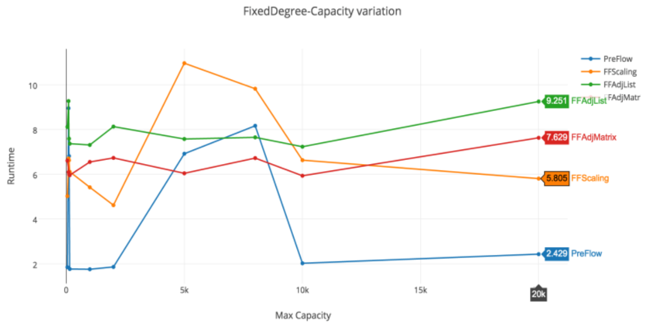
   As the max capacity increases, FordFulkerson is likely to be inefficient if it chooses a non-optimal augmenting path.



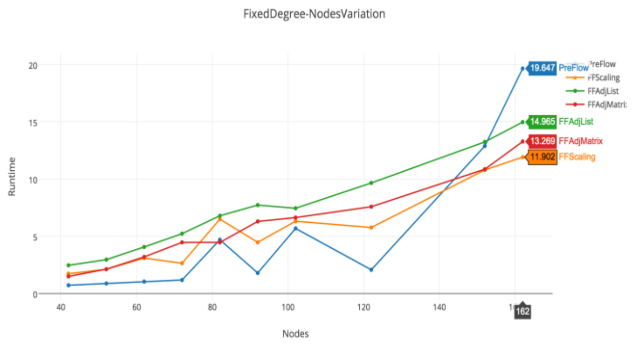
Bipartite Graph-Nodes Variation



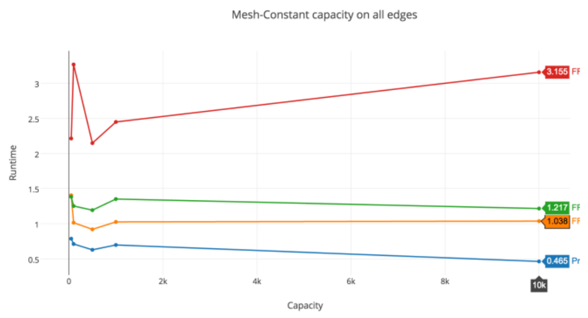
Bipartite Graph-capacity variation



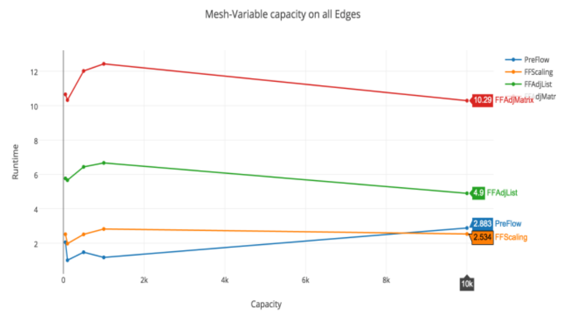
Fixed Degree Graph-capacity variation



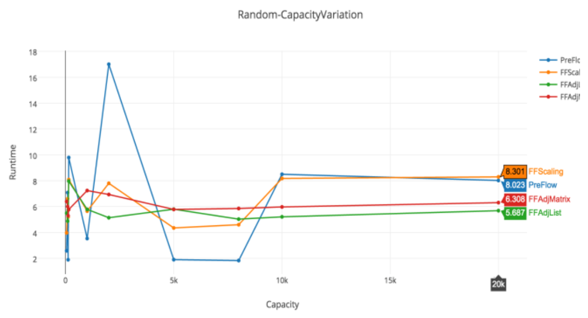
Fixed Degree-Nodes variation



Mesh Graph (with constant capacity)



Mesh Graph with Varying Capacities



Random Graph-capacity variation

This is the analysis when we compared different algorithms based on the variation of nodes, edges and capacities. We also compared these algorithms, based on how they performed for particular input graphs. We found that it could not be accurately predicted, which algorithm was faster for a particular type of input graph. The runtime depends mostly on the network topology, i.e the number of nodes, edges and the edge capacities in the network.

5. Results:

After performing the study, we came to the following conclusions:

1. The Ford-Fulkerson with scaling algorithm depends very heavily on the number of edges, as its running time is $O(m^2 \log C)$, where m is the number of edges. This algorithm takes more time if the max capacity is small and the number of edges is large, which makes the logC term almost the same as C. But, number of edges causes the runtime to spike.
2. The Ford-Fulkerson depends on the total capacity of the graph out of source, as its runtime is O(mC). Its runtime increases as the maximum capacity increases.
3. The Preflow-push depends very heavily on the number of vertices and the time taken by the algorithm increases after certain number of vertices, because of running time $O(mn^2)$, where n is number of vertices and m is the number of edges.
4. The performance of the algorithms depends not only on the implementation but also the type of data structure used. Eg.: Adjacency matrix and Adjacency list representations.
5. The graphs for the capacity variation have a zigzag pattern, because the input files take the range of capacity and the graph generated can have capacity equal to or less than the max capacity. For example: If we consider capacity ranges for two different input files, one with max capacity 60 and the other with max capacity 40. The first graph can have capacities ranging from 0-60, i.e the first and second graphs can have the same capacities by chance.

6. Challenges:

1. We found out that the use of different data structures will impact the time complexity of the algorithm. To further analyze the impact we created two implementations of Ford-Fulkerson algorithm. We implemented the algorithm using the adjacency list and adjacency matrix, and analyzed the results using both the implementations.
2. The execution of Mesh Graph for over 100x100 nodes was giving out of heap space. So could not test it on large number of nodes
3. The graph created using the same input parameters, generates random graph every time. This can vary the result. To prove this we created 3 test files for bipartite graph with the same parameters like nodes at source =150; nodes at sink = 150; probability = 0.5 and max capacity 1-50. After running the implementation on each file we found that maxflow and runtime was different, due to different topologies created everytime.

Conclusion

From the empirical study of the network flow algorithms, we came to the conclusion that the Preflow-push algorithm is the most efficient algorithm. This is subject to the case where the number of nodes is not very high.

If number of nodes for any graph is large, the Ford-Fulkerson with scaling algorithm will be preferred. If the number of edges is very high, and the max capacity of the network is low, this algorithm will not be efficient due to the m^2 component (which does not exist in the Ford-Fulkerson algorithm).

The performance of the FordFulkerson is affected by the augmenting path chosen. It can be very inefficient in case the path chosen is bad. This increases the number of iterations for calculating the max flow.

We also found that, the performance of the algorithms not only depends on the implementation but also on the type of data structure used.

Future Work

To extend the empirical study analysis for future we would like to test all the three algorithms on the real time data and analyze the performance of each algorithm. We can use the data from the real-time applications such as Airline scheduling, Image segmentations etc.

We can also extend the project to generate the multiple input files using same input parameters. Run the implementation of algorithms for each file and take the average of each time execution. This will give us a better idea of how each algorithm behaves and how the graph code generates different files on each run of the same input parameters.

References

1. Algorithm Design : Jon Kleinberg and Eva Tardos
2. Introduction to algorithms: Thomas H Cormen, Charles E. Leiserson, Ronald L Rivest and Clifford Stein
3. https://en.wikipedia.org/wiki/Random_graph
4. https://en.wikipedia.org/wiki/Bipartite_graph
5. https://en.wikipedia.org/wiki/Network_flow
6. https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm