Report - Empirical study project on network flow

Alekhya Palle, Ashwin Meena Meiyappan, Sri Vibhu Paruchuri
University of Washington, Tacoma

Introduction:

Description:

This document is a report on the project Empirical Study on Network Flow. Here we implemented three algorithms Ford Fulkerson, Scaling Ford Fulkerson and Preflow-Push algorithm to different types of graphs representing network flows.

Goals:

The goals of this project are to analyse the results of the three algorithms using different which represent the network flows from various scenarios. The comparison of execution times of all three graphs would give us the best-suited algorithm for a situation. We also find out how the flow changes per different algorithms when we fluctuate the capacity of edges.

Results:

The pre-experiment expectation was that Preflow-Push would perform better, but contrary to expectations, the Ford Fulkerson algorithm was better suited for some inputs. Also, the type of traversal in Ford Fulkerson and Scaling Ford Fulkerson plays a major role in the execution time.

Methodology:

Algorithms:

1. Ford-Fulkerson: The idea behind the algorithm is: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths.

2. Scaling Ford-Fulkerson: This algorithm is a variant of Ford-Fulkerson which considers choosing good augmenting paths in the maximum flow problem to minimize the number of iterations.
3. Preflow-Push: Previously mentioned algorithms based on augmenting paths maintain a flow f and use the augment procedure to increase the value of the flow. By way of contrast, this algorithm will increase the flow on an edge-by-edge basis.

**Please refer to appendix section 1 for more information on the above-mentioned algorithms.*

Implementations difficulties:

1. We had difficulties with deciding a common data structure for all three algorithms. A preflow-push algorithm stores two new parameters(excess and height) of every vertex.
2. Deciding on how to vary parameters for input graphs.
3. Also deciding on using Breadth-First Search (BFS) or Depth-First Search (DFS) for Ford-Fulkerson and Scaling Ford-Fulkerson. So we ran the input graphs against algorithms using both BFS and DFS traversals.

Testing:

We ran all three algorithms against the input graphs only once. We captured the first execution time instance because python's way of allocation memory will impact the runtime of the algorithm.

1. For input Bipartite graphs we tested the following parameters:
    - Execution time vs varying nodes and keeping a range of capacity constant.
    - Execution time vs varying range of capacity while keeping nodes constant.

2. For input Mesh graphs we tested the following parameters:
    - Execution time vs varying nodes and keeping a range of capacity constant.
    - Execution time vs varying range of capacity while keeping nodes constant.
3. For input Random graphs we tested the following parameters:
    - Execution time vs varying nodes and keeping edge & range of capacity constant.
    - Execution time vs varying range of capacity while keeping edges & nodes constant.
    - Execution time vs varying edges while keeping nodes & range of capacities constant.

Max vertices : 600
Min vertices : 100

Max edges/node: 300
Min edges/node: 100

Max range of capacity/edge: 600
Min range of capacity/edge: 100

Input instances:

1. Used the "BipartiteGraph.java" file to generate bipartite graphs of varying parameters.
2. Used the "MeshGenerator.java" file to generate mesh graphs.
3. Used the "RandomGraph.java" file to generate random graphs.

**Refer to Appendix Section 2 for the description of input graphs.*

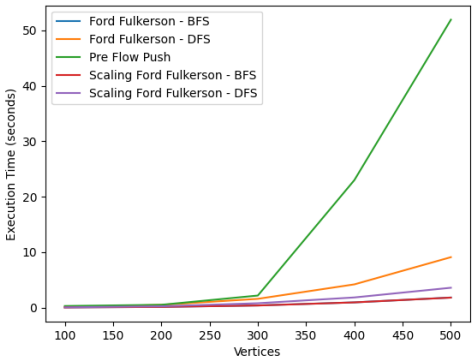
Data structures used:

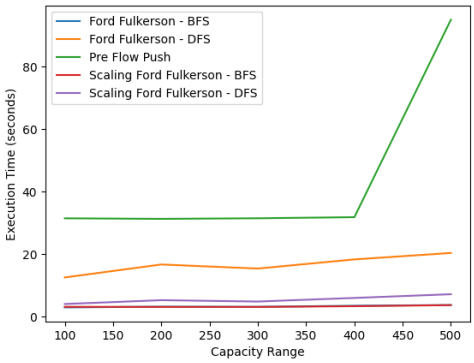
Adjacency matrix: Ford-Fulkerson and Scaling Ford-Fulkerson. Refer to "graphConv()"
Adjacency lists: for preflow-push algorithm

Choosing source(s) and sink(t) for our input graphs:
The first vertex in the input .txt files as source s and the last unique vertex as sink t.

Results:

1. Bipartite Graphs:

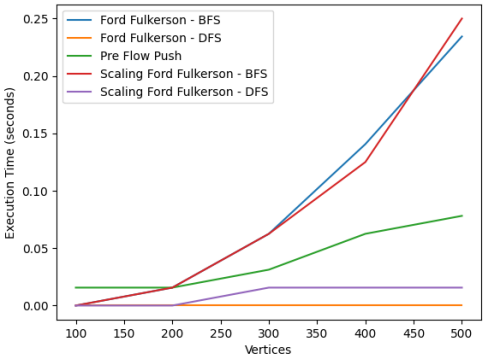


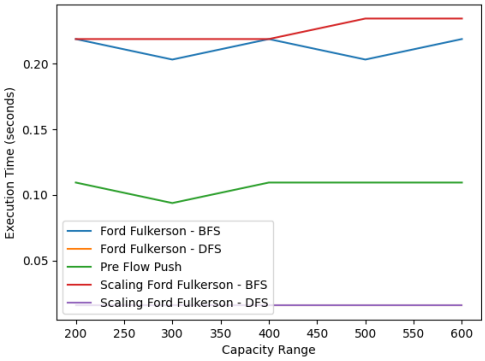


   a. The number of vertices:
      i. As the number of vertices increases, the execution time of all the algorithms increases as well. The execution time of the preflow-push is worse than the other algorithms.
      ii. The BFS versions of Ford Fulkerson and Scaling Ford Fulkerson perform better than their DFS counterparts.

   b. Capacity Range:
      i. As the range of capacity increases, the execution time doesn't vary much for all the algorithms.
      ii. There is a sudden spike noticed in the Preflow Push algorithm as the capacity range crosses a certain value.

2. Mesh Graphs:

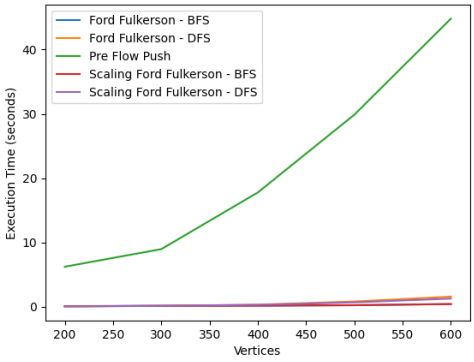


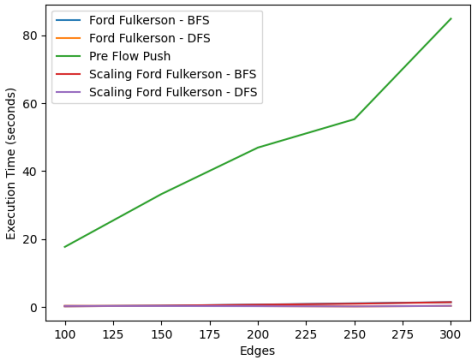


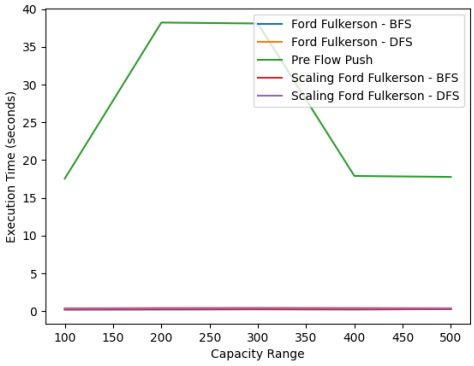
a. The number of vertices:
   i. As the number of vertices increases, so does the execution time for most of the algorithms except for DFS Ford Fulkerson which stays mostly uniform.
   ii. The DFS versions of Ford Fulkerson and Scaling Ford Fulkerson perform better than their BFS counterparts.
   iii. Preflow Push performs better than the BFS versions of the Ford Fulkerson and Scaling Ford Fulkerson.

b. Capacity Range:
   i. For all the algorithms, the change in execution time is negligible as the capacity range is increased
   ii. However, some spikes are noticed in Preflow Push and the BFS versions of the Ford Fulkerson and Scaling Ford Fulkerson.

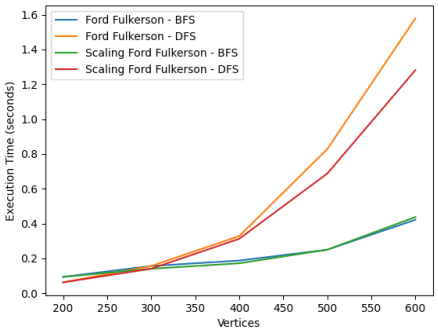
3. <u>Random Graphs:</u>

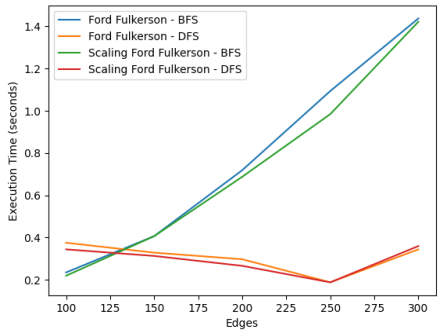


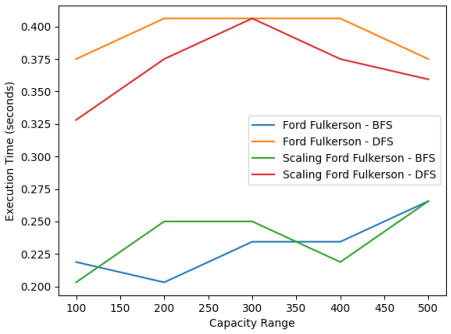




a. When comparing the implementation of all the algorithms, Preflow Push is the worst performing algorithm irrespective of which parameter is varied.







b. Ford Fulkerson vs Scaling Ford Fulkerson:
    i. The number of vertices:
        - As the number of vertices increases so does the execution time.
        - The BFS version of Ford Fulkerson and Scaling Ford Fulkerson perform much better than their DFS counterparts.

ii. The number of edges:
- As the number of edges increases so does the execution time increase for BFS versions of the Ford Fulkerson and Scaling Ford Fulkerson.
- Surprisingly, the execution drops as the number of edges increases for the DFS versions of the Ford Fulkerson and Scaling Ford Fulkerson. Though, it increases towards the end. However, this increase is negligible.
- The DFS versions of Ford Fulkerson and Scaling Ford Fulkerson perform much better than their BFS counterparts.

iii. Capacity Range:
- As the capacity along the edges is increased, there is no discernible pattern observed in the execution time as there are a lot of spikes.
- The BFS versions of Ford Fulkerson and Scaling Ford Fulkerson perform better than their DFS counterparts.

Future Work:

1. Performing empirical analysis with a common input data structure for all algorithms. For instance, using an adjacency list for all the algorithms.
2. Check the performance of algorithms by varying the input data structure, adjacency list vs adjacency matrix.
3. Check the effect on analysis by changing the data structures in the implementation of BFS and DFS traversal. For instance, using the collections.deque instead of a standard list to implement a queue.
4. Analysing execution time with other programming languages. As python is a dynamically typed language, languages like C++ might take less time to execute.
5. Analysing memory usage with other programming languages. As python does memory management by itself and reduces the workload of the developers. So performing analysis on memory usage in python will not give us accurate results of memory used by the algorithm.

Lessons Learned:

1. No algorithm is best suited for all types of graphs. The algorithm should be chosen based on the type of input graph. The appropriate algorithm can be chosen based on the above results.
2. The type of traversal (BFS vs DFS) has a major effect on the performance of Ford Fulkerson and Scaling Ford Fulkerson.
3. Optimizations that seem good on paper, might not translate to real-world performance.

**Refer to Appendix Section 3 for the Division of Labour.*

**Refer to Appending Section 4 for References*