Comparison of Maximum Flow Algorithms

Çağatay Yıldız - 2014700129
Boğaziçi University, Computer Engineering Department
07.01.2014

Abstract—In this work, I compared execution times of four different algorithms for maximum flow problem in various types of networks. More specifically, I implemented generic labelling algorithm and scaling algorithm as we learnt in class. To test my code, I made use of implementation of Edmonds-Karp Algorithm by JGraphT library and Ford Fulkerson Algorithm by Princeton University's Algorithms text book.

Index Terms—Max-flow, Complexity Analysis, Edmonds-Karp Algorithm, Ford Fulkerson Algorithm.

1 Introduction

In the class, we examined many algorithms for maximum flow problem. It was concluded that the complexity of **generic labelling algorithm** is $O(mnU)$ where m, n and U denotes respectively the number of arcs, number of vertices and the greatest capacity on any arc noting that we take absolute values of all arc capacities. We categorized such algorithms (whose complexities depend on the initial configuration of the network) as *pseudo-polynomial time* algorithms. To get a polynomial time algorithm, we studied **scaling approach**, whose complexity turns out to be $O(m^2 logU)$. On paper, this really seemed a good improvement.
When I first see pseudo-codes, I was not very convinced with the way we analysed the complexity of generic labelling algorithm and I thought that in real-life there should not be big differences in the runtimes of those algorithms. So, it seemed worth attempting to implement and compare those.

2 Methods

2.1 Coding Details

We went over pseudo-codes of generic labelling algorithm and scaling approach in the class. While writing my code, I exactly followed my lecture notes. I also wondered how using depth first search instead of breadth first search during path scans from source to sink affects performance and implemented both. In addition, I searched for different implementations of the same problem so that I can understand if I calculated max-flow correctly and efficiently. An important detail is that I did not used adjacency list in my implementation. Instead, I stored the list of outgoing arcs for each vertex, which made my code more complex. Though, I think it is important not to consider $O(m)$ and $O(n^2)$ as exactly the same thing because our complexity analysis does not work in this way.
Since I developed my code in **Java**, I had a look at the library named **JGraphT**. There[2], I see Edmonds-Karp algorithm and downloaded the source code. As stated in `.java` file, "*... Edmonds-Karp algorithm is an implementation of the FordFulkerson method for computing the maximum flow in a flow network in $O(nm^2)$ time*".
Moreover, I have encountered another piece of code for this problem in the website of a textbook named *Algorithms*[3], written by two from Princeton University. In fact, they developed a really nice library to show an example of good coding and they made use of this library in their code. It was noted in the website that "*... Program FordFulkerson.java computes the maximum flow ... in $O(m^2n)$ time using the Edmonds-Karp shortest augment path heuristic (though, in practice, it usually runs substantially faster)*".

2.2 Data Sets

In this work, I generated various networks based on three different parameters:

1) Number of vertices
2) Density (ratio of the number of arcs to the number of all possible arcs)
3) The lowest and greatest capacity on arcs

While generating data, I kept two parameters constant and one changing, which results in three distinct network types. In addition, I generated a special type of network, which is made of two subnetworks, call them S_1 and S_2. S_1 is a fully connected subnetwork made of some number of vertices, including the source, and each arc here has the same capacity, say c. Only one vertex from S_1, call it A, is connected to S_2. In S_2, we have the target and c other vertices and all of these c vertices are connected to A and target by unit flows. Here, each flow from source to target has capacity one. So, one might expect generic labelling algorithm to work faster than the scaling algorithm since scaling phase in this network is simply unnecessary.

3 Results

Results are provided in Figure 1,2,3 and 4.

- My first observation is that details of implementation do matter. Princeton's Ford-Fulkerson Algorithm for example, runs noticeably slower than other algorithms and it is also not very stable because execution time makes sudden increases in some networks. This is I think because they have built a library that is not specifically targeted at solving the problem efficiently but coding in a nice fashion.
- When we compare breadth first search and depth first search, it is in general true that BFS runs faster than DFS, which is not very surprising. In my algorithm design course, we had noted that DFS is better to use if the searched vertex is far from the root, or the graph is very deep. While generating arcs, source and sink I made random choices and therefore, we do not expect the source and the target to be very further away and BFS is supposed to run faster, which is the case.
- If you look at BFS graphs, you will see that general labelling algorithm is really faster than scaling approach (Remember, that piece of information was the very first thing that I wondered). This is I think because too many operations are done in scaling approach. As mentioned before, I stored the outgoing arcs of a vertex in a data structure I defined, called `Vertex`. So, to go over all vertices in the network, two nested loops that are iterating over `ArrayList`'s are required and this is definitely costly. It could be nice to compare my code with another implementation of the problem in which arcs are stored in an adjacency matrix in `array`'s, which will probably run faster.
- In contrast, if you look at Figure 1 and 2, you will see in DFS graphs that scaling algorithm runs in shorter amount of time than general labelling. This is also not surprising because too many operations are done in DFS to find the sink and scaling algorithm reduces search space by not considering a big proportion of arcs in many steps, which reduces the number of operations needed.
- Figure 3 is I think interesting. It is shown that the change in the maximum capacity does not cause any difference in run-times. I am not pretty sure but it might be because networks are not that big and flows augmented are not that small. (A side note: Each network generated for this figure has 50 vertices and maximum capacity is 2500)
- Figure 4 has also very nice interpretations. As mentioned above, each flow in this network has capacity 1. In class we noted such a network as an example in which scaling algorithm performs bad and you can confirm this looking at graphs. Moreover, see that as the number of vertices increases, it becomes worse and worse because number of operations in network update (removing and adding arcs) step of scaling algorithm increases.

4 Challenges

One particular problem with this project was regarding how to detect execution times of algorithms. Precision was certainly an issue;

therefore, I decided to work with nanosecond precision. One way of getting execution time is to query Java Virtual Machine for CPU time spent so far by the running program. It took some time for me to realize that some programs run in 10-20 nanoseconds - or at least this was what JVM tells me. This did not seemed realistic, so I decided to use system clock via `System.currentTimeMillis()` function. I think it worked correctly.

After solving this problem, I started to run my algorithms and measure execution times. At a point, I noticed that no matter which kind of network I am working with, generic labelling algorithm works faster than scaling algorithm. This also seemed odd and a couple of days later I saw the reason why: The function call to run generic labelling algorithm is done after the call for scaling algorithm! When I reversed the order of call, I saw that the contrary happens. A quick internet research yielded that one needs to be very careful while measuring execution times due to various reasons such as the increase in the memory size reserved for the program, the decrease in the amount of time that program waits before allocating CPU, the activities of Java's Garbage Collector and Virtual Machine and so far. Thus, I followed the steps below:

- I created 100 different networks for each possible network parameter configuration. Note that if you construct a network using exactly same parameters, they are (with very very big probability) not the same since the arcs are assigned randomly. While generating arcs, I generate a uniformly random number between 0 and 1 for each possible arc and based on the density parameter and this random number, an arc occurs between these vertices or not. Being more clear, to measure the effect of the number of vertices on execution time, I created 100 different networks for each vertex count I am examining, which corresponds to the execution of tens of thousands of max-flow algorithms.
- I saved all the networks created to the file system. Well, this was quite boring since I needed to save, read and delete all those files, which took quite some time not only to implement but while executing.
- Here is the last trick: Instead of running an algorithm on all networks consecutively, each time I chose the algorithm, network parameters and the particular network of these parameters randomly. This way, I tried to get rid of the *bias* created by various reasons I listed above. I am not claiming this compensates the bias but that was all I can think of and findings support my claim.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Edmonds Karp Maximum Flow Algorithm, https://github.com/jgrapht/jgrapht/blob/master/jgrapht-core/src/main/java/org/jgrapht/alg/EdmondsKarp MaximumFlow.java

[3] Princeton University, Algorithms, 4. Edition, http://algs4.cs.princeton.edu/64maxflow/

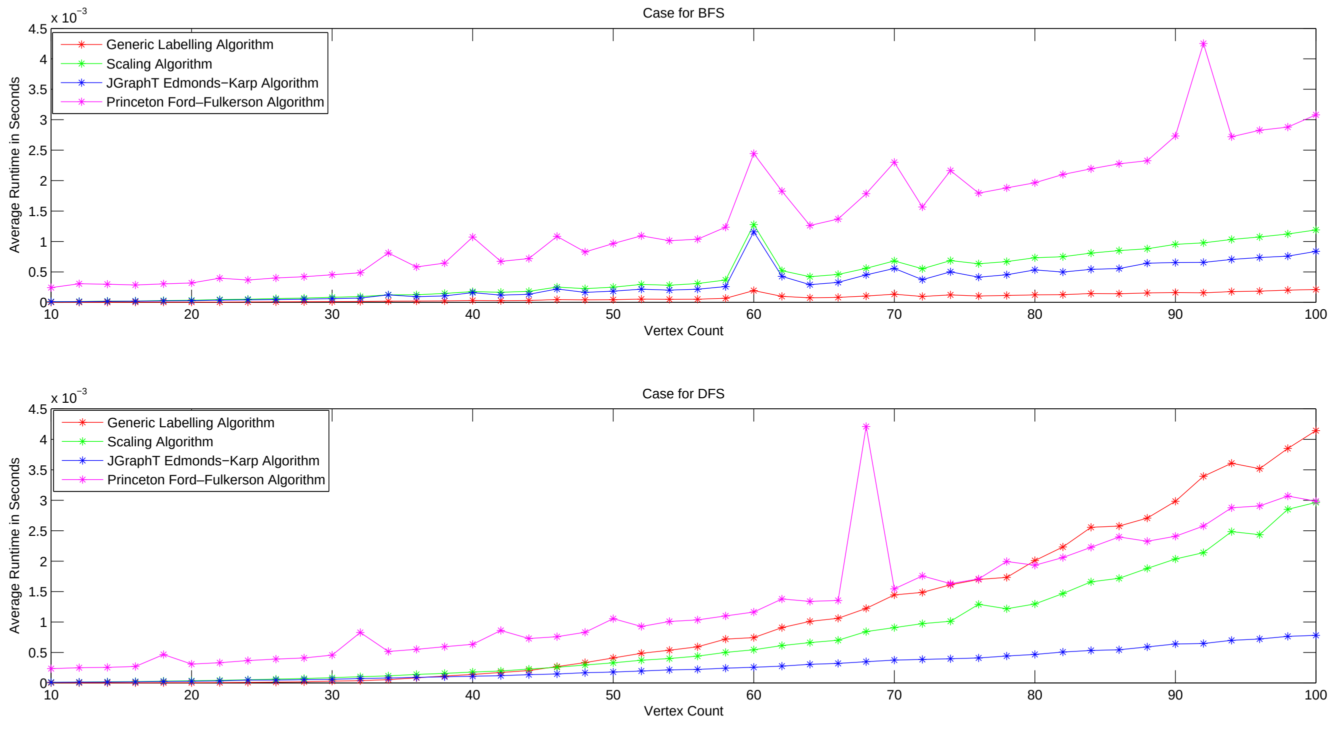

Fig. 1. Vertex Number vs Execution Time

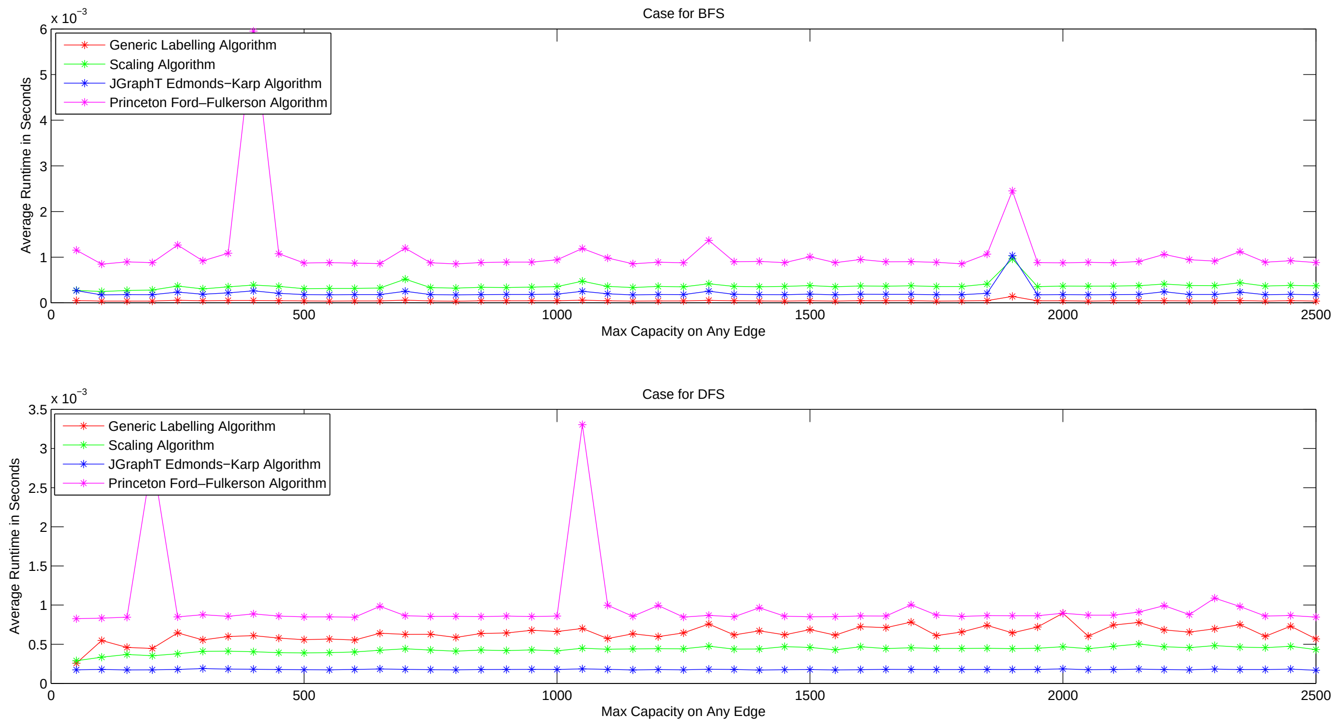

Fig. 2. Max Capacity vs Execution Time

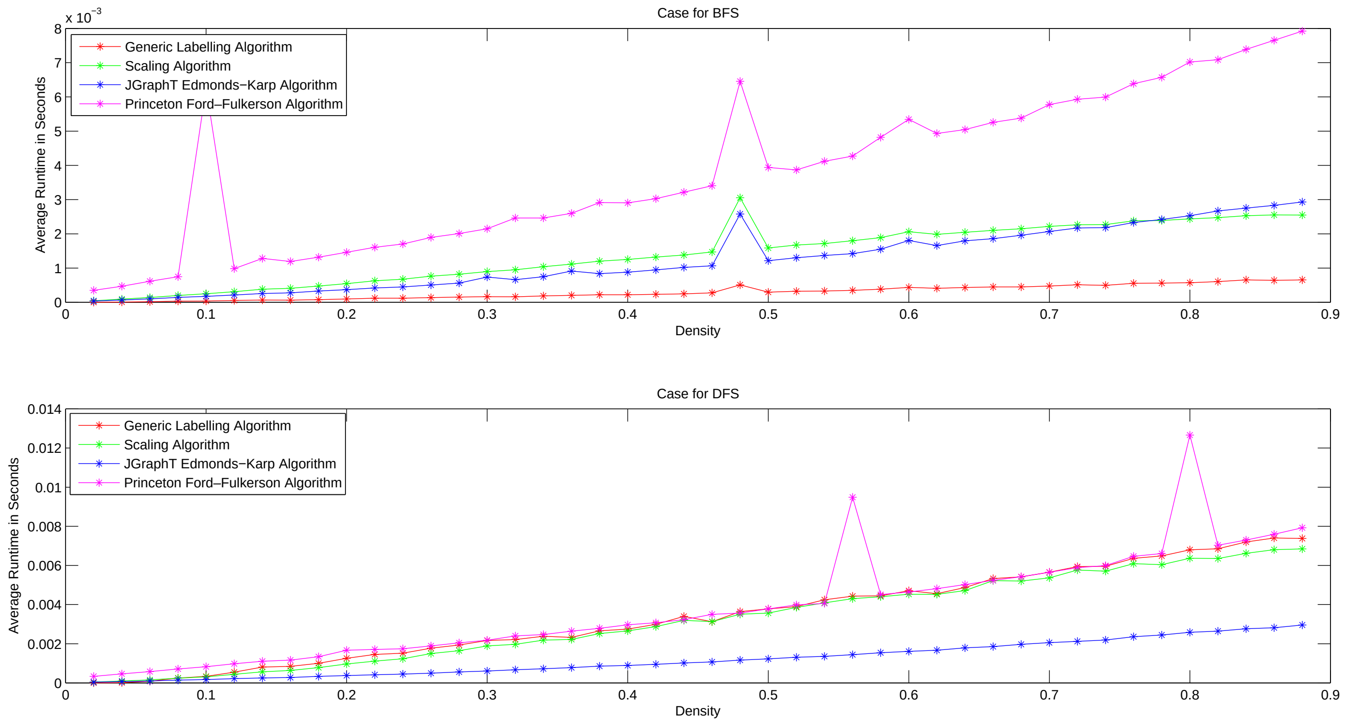

Fig. 3. Density vs Execution Time

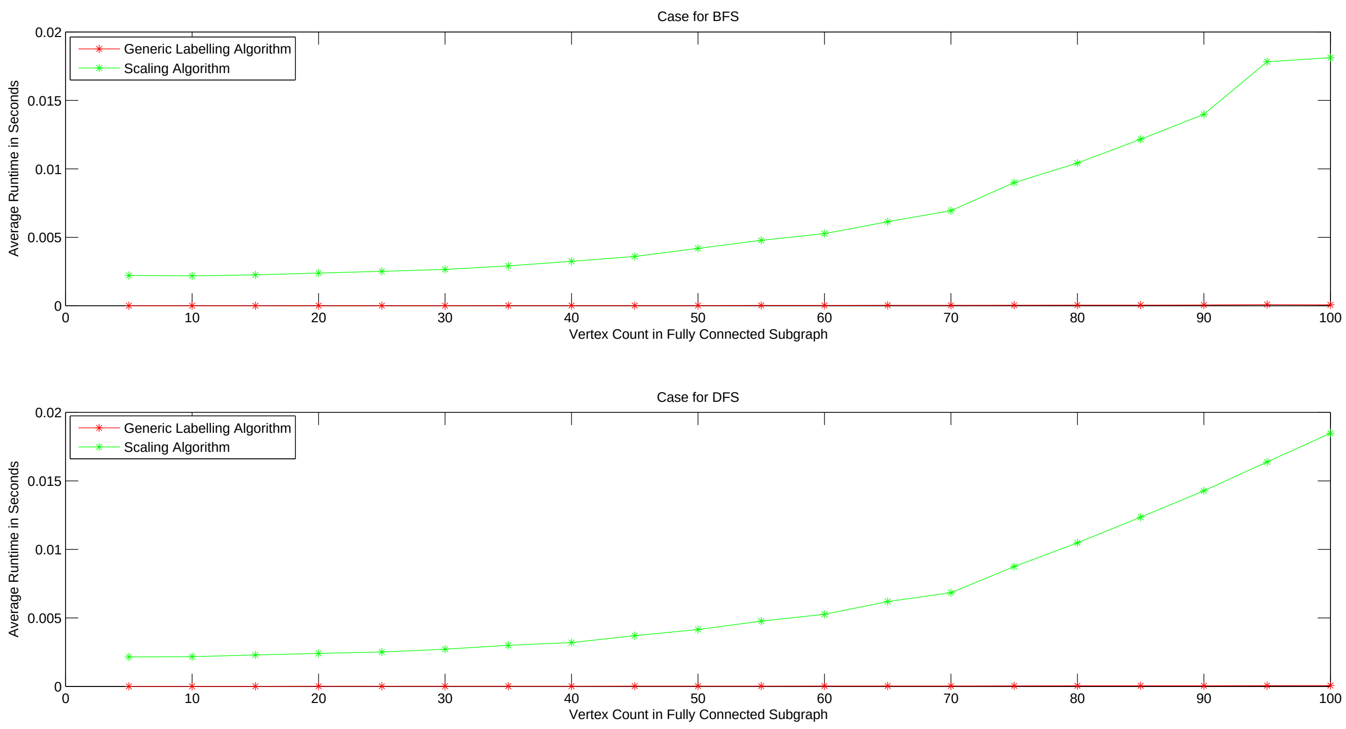

Fig. 4. Vertex Count in Fully Connected Subnetwork vs Execution Time