An Experimental Evaluation of Optimized Maximum Flow Implementations

Junxi Song, Luming Tang, Hongbo Zhang, Xiaoji Zhang

December 11, 2019

1 Introduction

The maximum flow problem is a classical optimization problem that has been studied for decades. Recent years have witnessed significant improvement in the efficiency of max flow algorithms, with respect to both theoretical bounds and practical implementations. The theoretical bounds of the earliest shortest augmenting path algorithm[1] and its blocking flow variation[2] have been further improved by introducing novel methods such as capacity scaling[3] and dynamic trees[4], as well as solving the problem on different types of graphs[5, 6]. On the other hand, parallel and distributed max flow algorithms that are run on both PRAMs ($|V|$-processors)[7] and modern computer architectures[8] have been proposed, and practical implementations with heuristics such as global relabeling[9] and gap relabeling[10] have been studied.

In this coding project, we implement three basic flow algorithms, namely, Ford-Fulkerson (with the shortest augmenting path implementation), Edmonds-Karp, and Dinic. We also propose numerous optimization strategies to the Ford-Fulkerson and Dinic's algorithms to improve overall performance. Subsequently, we evaluate their respective performance on different types of graphs with varying capacity settings. We conclude that the performance of algorithms conforms with the respective theoretical complexity, and that our optimizations improve the performance of Ford-Fulkerson and Dinic's algorithm by a magnitude on large, dense networks.

2 Methods and Implementation

2.1 Edmonds-Karp Algorithm

We use the Edmonds-Karp's shortest augmenting path heuristic given in Kozen's book and implement a breadth-first search. No particular optimization strategy is applied.

2.2 Dinic's Algorithm

Our implementation of Dinic's algorithm adopts a slight modification from Kozen's book[11] by incorporating the Push operation from the push-relabel algorithm in the depth-first search of the level graph. The following algorithm demonstrates the path augmentation process for a single phase of the level graph.

Algorithm 1 DinicSinglePhase

Input: $G = (V, E, c)$, sink t, cur_node (initialized to source s), acc_flow (current accumulated flow)

Output: maximum amount of flow for a single phase max_flow

```
1:  Find the level graph $L_G$
2:  Initialize $f(u,v) = \begin{cases} c(u,v) & \text{if } L_G(u) < L_G(v) \\ c(v,u) & \text{otherwise} \end{cases}$
3:  for all $v$ such that $(cur\_node, v) \in E$ do
4:      if $f(cur\_node, v) > 0$ and $L_G(v) == L_G(cur\_node) + 1$ then
5:          $\delta \leftarrow$ DinicSinglePhase$(G, t, v, min(f(cur\_node, v), acc\_flow))$
6:          if $\delta > 0$ then
7:              $f(cur\_node, v) \leftarrow f(cur\_node, v) - \delta$
8:              $f(v, cur\_node) \leftarrow f(v, cur\_node) + \delta$
9:              $acc\_flow \leftarrow acc\_flow - \delta,\ max\_flow \leftarrow max\_flow + \delta$
10:         else
11:             $L_G(v) = 0$
12:         end if
13:         if $acc\_flow == 0$ then
14:             $L_G(v) = 0$
15:             BREAK
16:         end if
17:     end if
18: end for
19: Return $max\_flow$
```

Our implementation first computes the shortest distance from the source for every node by BFS on the residual graph with all edges having positive capacity. Then, we push a flow $f(u, v)$ only if $d(s, v) = d(s, u)+1$. Compared to Edmonds-Karp that pushes flow on an augmenting path one at a time, Dinic's algorithm pushes all possible flows from source to sink across all possible augmenting paths on the currently labeled layered graph, and repeats the labeling and pushing flow process until it cannot find any augmenting path on residual graph.

We implement two optimization strategies (marked in blue in the pseudo code) for pruning nodes (i.e. force subsequent depth-first searches to skip the node) which could not be on any augmenting paths of the residual graph.

1. If no more flow could be pushed from v, by the conservation property of flow networks, there is no flow that goes from source to v, so we prune v.

2. If the accumulated flow at the current node is zero, then no flow could be pushed from *cur_node*, and we prune the node.

2.3 Ford-Fulkerson Shortest Augmenting Path (SAP)

We adopt the shortest augmenting path (SAP) implementation of the Ford-Fulkerson method, which is similar to Dinic's algorithm in that it keeps track of a level graph and finds the augmenting path with depth-first search. Despite the similarities and the same running time complexity of $O(mn^2)$, our algorithm is significantly different from Dinic's in two aspects:

1. At the beginning of every iteration (i.e. every augmenting path search), Dinic's algorithm preprocesses the graph using breadth-first search to generate a corresponding level graph, which remains constant during the subsequent augmenting path search. Conversely, SAP does not populate the level graph beforehand, but instead constructs and updates the level graph greedily during the DFS. The greedy approach in the level graph construction leads to the second key difference between the two algorithms.

2. Dinic's algorithm guarantees that the augmenting paths are always advancing with respect to the level graph, i.e. for $(u, v) \in E$, if $d(v) = d(u) + 1$ where d is the distance between a node and the source, then $L_G(v) = L_G(u) + 1$. In other words, $L_G(s) = 0$, and $L_G(t) =$ length of max augmenting path. On the other hand, the level graph constructed by SAP is the exact opposite due to its greedy nature. Initially, all entries of the level graph are set to zero. The depth-first search then

starts from adjacent nodes of the source $v \in Adj(s)$ and attempts to push the flow by checking if $L_G(s) = L_G(v) + 1$. To push the flow towards the sink, the algorithm updates $L_G(s)$ to 1. The level graph is updated using the same paradigm throughout subsequent steps in the searches, and the resulting level graph has $L_G(t) = 0$ and $L_G(u) = L_G(v) + 1$ if $d(v) = d(u) + 1$.

In addition to the differences in implementation explained above, our algorithm applies another optimization approach for pruning the search (aside from the ones in Dinic), which introduces an array corresponding to the level graph that stores the number of nodes in each level, and is updated in the DFS alongside the level graph. The pruning works as follows: Every time we increment the level of a node, we first decrement the corresponding entry in the array. Since the levels are continuous along an augmenting path, if the array entry reaches zero at a node that is not the sink, it means that there can be no augmenting path beyond this node, so we set the level of the source to an unachievable value, say the size of the graph, so that the path gets pruned in subsequent searches. Similar to Dinic's implementation, the optimization significantly improves the efficiency and makes the algorithm testable on large networks.

3 Experiments

In this section, we describe the tests that are run on the three implementations, in particular, the features of the networks and the capacity settings, as well as what we expect to see from each algorithm. Prior to running the experiments, we verify the correctness of our implementations by comparing both the output maximum flow and the residual graph after sending all flows from source to sink. Subsequently, we refer to the experiment settings of Friis[12] and Cherkassky[13] respectively, and test our implementations on three types of graphs, elaborated in the following subsections.

3.1 Connected Randomized Graphs

We use four graphs from the DIMACS and infrastructure categories in the open-source network repository[14]:

- C2000-5: A connected graph with 2000 nodes, each with 1000 degrees

- MANN-a81: A fully connected graph with 3000 nodes
- C4000-5: A connected graph with 4000 nodes, each with 2000 degrees
- inf-power: A connected graph with 5000 nodes, each with 2 degrees

To convert the graph into a network, we find a valid pair of source and sink and define the capacity for each edge. The source is sampled from a set of randomly chosen nodes (we use 200 nodes in all our experiments). Subsequently, we apply breadth-first search to find the longest acyclic path starting from each potential source in the sample, and define the other end of the longest path as the sink. Additionally, we experiment with two different capacity settings: unit capacity for all edges, and randomly sampled capacity from $[1, 10^4]$ for each edge.

Regarding the expected performance of algorithms on the four networks, both SAP and Dinic's algorithm have worst-case running time of $O(mn^2)$. Nevertheless, Dinic's algorithm is expected to perform particularly well on sparse graphs such as inf-power. Since there are only a limited number of ways to label sparse networks, Dinic's algorithm generally saves time on constructing and updating level graphs compared to SAP, which adjusts the level graph during the augmenting path search, as opposed to populating the graph prior to the search. On the other hand, Edmonds-Karp has time complexity of $O(m^2n)$, and will perform much worse than the other two on very dense graphs, such as the fully connected MANN-a81.

3.2 GenRmf

Contrary to the first set of experiments that incorporate extreme cases and work very well or poorly on particular algorithms, the following two sets of experiments are designed to be more random and illustrative of the typical behavior of the three algorithms. The GenRmf generator[15] produces a special type of graphs proposed by Goldfarb and Grigoriads, which takes parameters (a, b, c_{min}, c_{max}) and generates a graph with b layers of nodes and $a \times a$ nodes on each layer. In our experiments, the capacity for each edge is randomly chosen from $[1, 1000]$, and we have three different settings: long (where $a^2 = b$), square (where $a = b$) and wide (where $a = b^2$).

- GENRMF_Long: A network with $n = 2^x$ nodes, generated from running GenRmf with $a = 2^{x/4}$ and $b = 2^{x/2}$

- GENRMF_Square: A network with $n = 2^x$ nodes, generated from running GenRmf with $a = 2^{x/3}$ and $b = 2^{x/3}$
- GENRMF_Wide: A network with $n = 2^x$ nodes, generated from running GenRmf with $a = 2^{2x/5}$ and $b = 2^{x/5}$

3.3 Washington

We use the Washington library (a collection of graph generators from DIMACS) to generate random leveled graph, with parameters $(dim1, dim2, range)$ corresponding to number of rows, number of columns, and the capacity range. Similar to GenRmf, we have three different settings: long, wide and moderate, corresponding to the ratios of the number of rows to the number of columns. All three settings have edge capacities randomly sampled from $[1, 10^4]$.

- Washington_Long: A network with $n = 2^x$ nodes, generated from running Washington with $dim1 = 64$, $dim2 = 2^{x-6}$, and $range = 10^4$
- Washington_Moderate: A basic line mesh with $n = 2^x$ nodes, generated from running Washington with $dim1 = 2^{x-2}$, $dim2 = 4$, and $range = 2^{(x/2)-2}$
- Washington_Wide: A network with $n = 2^x$ nodes, generated from running Washington with $dim1 = 2^{x-6}$, $dim2 = 64$, and $range = 10^4$

3.4 Test Environment

Our experiments on Connected Randomized Graphs are run on the Dell C1100 machine, with dual Intel XeonX5570 CPUs and 72GB DDR3 ECC memory. Experiments on GenRmf and Washington are run on a macOS machine with Intel Core i7 Processor and 16GB memory. All programs are compiled g++ version 9.1 with O2 optimization flag.

4 Results

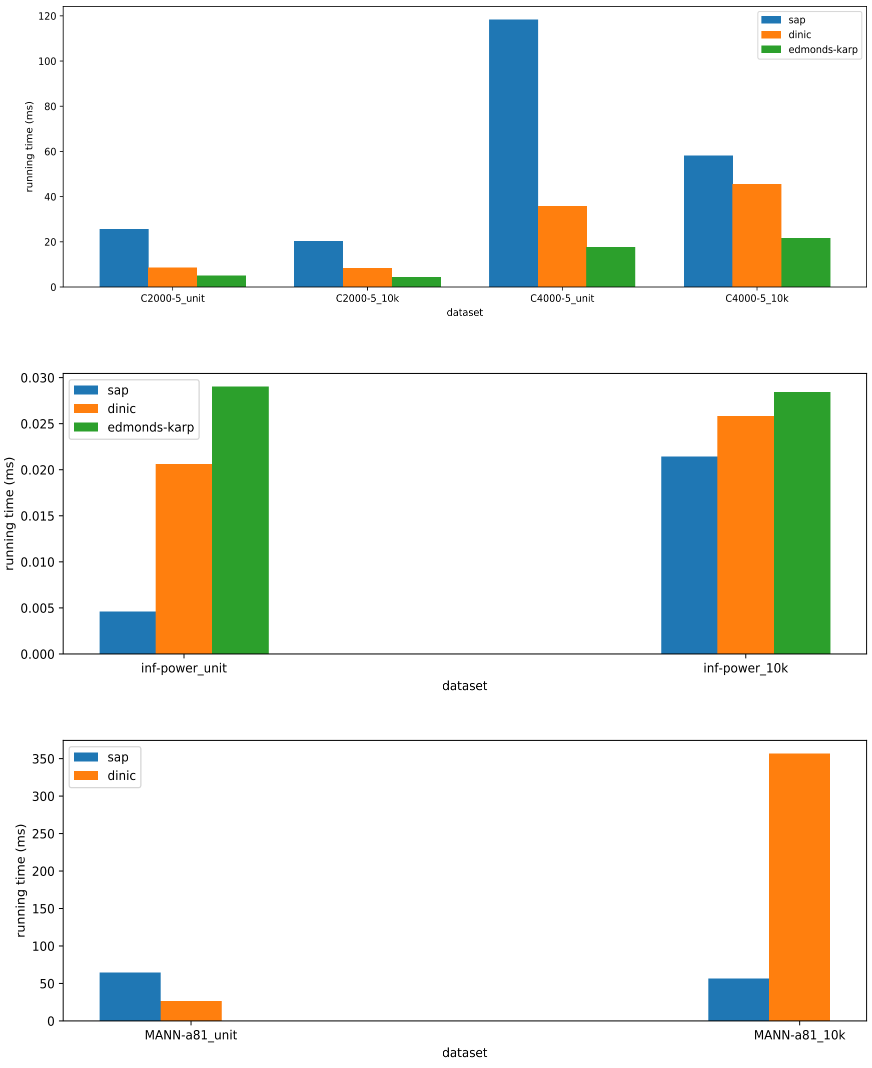

Figure 1: Running time (ms) comparison of the three algorithms on four different connected randomized networks (explained in the Experiment section). `unit` and `10k` refer to the respective edge capacity range $[1, 1]$ and $[1, 10^4]$. The running times of Edmonds-Karp on MANN-a81 are a magnitude larger than the other two and are not drawn (217209ms on MANN-a81_unit and 385894ms on MANN-a81_10k).

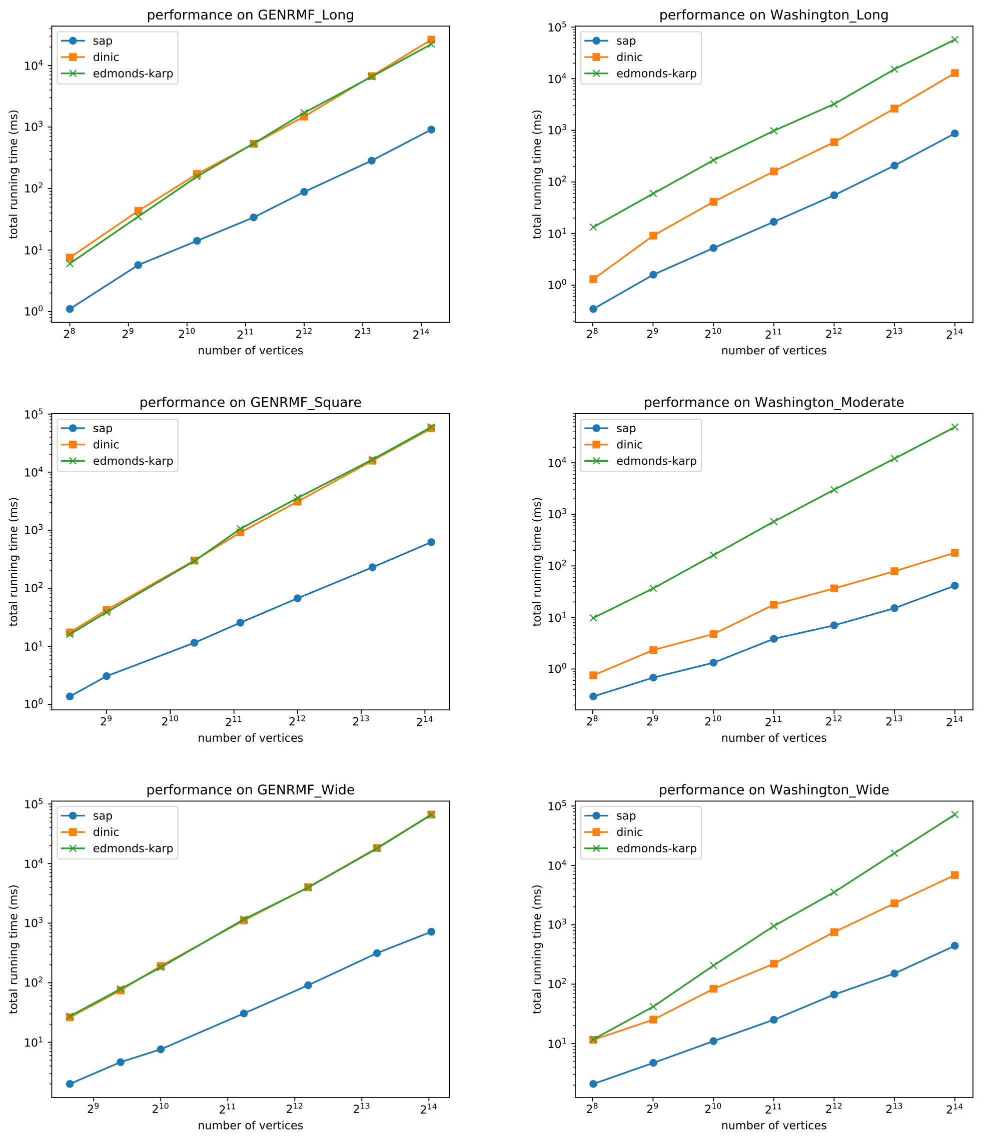

Figure 2: Running time (ms) comparison of the three algorithms on networks with varying numbers of vertices generated by GenRmf and Washington. `unit` and `10k` refer to the respective edge capacity range $[1, 1]$ and $[1, 10^4]$.

Figure 1 illustrates the performance of the three algorithms on different connected randomized graphs in DIMACS, which conforms with our expectation. The SAP algorithm does not outperform the other two on simple networks (either sparse or with few nodes), but works particularly well on large, dense graphs such as the MANN-a81 (a fully connected graph with 3000 nodes). On the contrary, Dinic's algorithm performs well on small, sparse networks, but runs significantly slower as the network grows larger and denser. The Edmonds-Karp algorithm adopts a similar but more extreme performance than Dinic's, as its performance deteriorates much faster with the increasing number of edges, which conforms with its theoretical running time of $O(m^2n)$.

Figure 2 shows the performance of the three algorithms on the networks generated by GenRmf and Washington, which corresponds to the typical behavior of the algorithms. The GenRmf networks are sparse as each node has degree from 2 to 4, while the Washington networks are much denser. Once again, the performance generally conforms with our observation for the connected random graphs: The SAP outperforms significantly on both sparse and dense networks, while Edmonds-Karp algorithm performs much worse than the other two on dense graphs.

Finally, we evaluate the effectiveness of our optimization strategies by comparing the SAP and Dinic's algorithms with and without optimization. According to Table 1, our optimizations improve the efficiency by a magnitude for both algorithms when running on complicated networks, and the improvement appears increasingly significant as the network grows larger and denser.

	C2000-5		**MANN-a81**		**C4000-5**		**inf-power**	
algorithm	**unit**	**10k**	**unit**	**10k**	**unit**	**10k**	**unit**	**10k**
sap	8826	8553	59488	29270	75201	52551	0.0032	9.272
sap-opt	25.62	20.15	64.24	56.39	118.2	58.02	0.0046	0.0214
dinic	304.65	331.09	49.50	4510	3954	3347	0.0208	0.0264
dinic-opt	8.446	8.396	26.18	356.5	35.75	45.37	0.0206	0.0258

Table 1: Running time (ms) comparison of the SAP and Dinic's algorithms with and without optimization on four networks with varying capacity range. 'unit' and '10k' stand for capacity range $[1, 1]$ and $[1, 10^4]$ respectively.

5 Conclusion

To conclude, our main focus in the project is the development of optimization strategies for the SAP implementation of Ford-Fulkerson and Dinic's algorithm with respect to pruning in the depth-first search, which paves the way for evaluating the performance of the algorithms on large networks. We have not only demonstrated that the algorithm performance conform with the theoretical complexity and pinpointed the strengths and weaknesses of each algorithm, but also explained the subtle difference between the SAP and Dinic implementations, and verified the effectiveness of our optimizations in shortening the running time by a magnitude for large fully-connected graphs.

References

[1] Edmonds, Jack, and Richard M. Karp. "Theoretical improvements in algorithmic efficiency for network flow problems." *Journal of the ACM (JACM)* 19.2 (1972): 248-264.

[2] Dinic, Efim A. "Algorithm for solution of a problem of maximum flow in networks with power estimation." *Soviet Math. Doklady.* Vol. 11. 1970.

[3] Gabow, Harold N. "Scaling algorithms for network problems." *24th Annual Symposium on Foundations of Computer Science (sfcs 1983).* IEEE, 1983.

[4] Goldberg, Andrew V., and Robert E. Tarjan. "Finding minimum-cost circulations by successive approximation." *Mathematics of Operations Research* 15.3 (1990): 430-466.

[5] Even, Shimon, and R. Endre Tarjan. "Network flow and testing graph connectivity." *SIAM journal on computing 4.4* (1975): 507-518.

[6] Karzanov, Alexander V. "On finding maximum flows in networks with special structure and some applications." *Matematicheskie Voprosy Upravleniya Proizvodstvom 5* (1973): 81-94.

[7] Shiloach, Yossi, and Uzi Vishkin. "An O (n2log n) parallel max-flow algorithm." *Journal of Algorithms 3.2* (1982): 128-146.

[8] Hong, Bo. "A lock-free multi-threaded algorithm for the maximum flow problem." *2008 IEEE International Symposium on Parallel and Distributed Processing.* IEEE, 2008.

[9] Anderson, Richard J., and Joao C. Setubal. "On the parallel implementation of Goldberg's maximum flow algorithm." *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures.* ACM, 1992.

[10] Bader, David A., and Vipin Sachdeva. *A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic.* Georgia Institute of Technology, 2006.

[11] Kozen, Dexter C. *The design and analysis of algorithms.* Springer Science Business Media, 2012.

[12] Friis, Jakob Mark, Steffen Beier Olesen, and Gerth Stølting Brodal. "An experimental comparison of max ow algorithms." (2014).

[13] Cherkassky, Boris V., and Andrew V. Goldberg. "On implementing push-relabel method for the maximum flow problem." *International Conference on Integer Programming and Combinatorial Optimization.* Springer, Berlin, Heidelberg, 1995.

[14] Rossi, Ryan, and Nesreen Ahmed. "The network data repository with interactive graph analytics and visualization." *Twenty-Ninth AAAI Conference on Artificial Intelligence. http://networkrepository.com.* 2015.

[15] "Maximum Flow Problem Instance Generator GENRMF". *Informatik.Uni-Trier.De,* 2019, http://www.informatik.uni-trier.de/naeher/Professur/research/generators/maxflow/genrmf/index.html.

[16] Goldfarb, Donald, and Michael D. Grigoriadis. "A computational comparison of the dinic and network simplex methods for maximum flow." *Annals of Operations Research* 13.1 (1988): 81-123.

[17] Archive.dimacs.rutgers.edu. (2019). *Index of /pub/netflow.* [online] Available at: http://archive.dimacs.rutgers.edu/pub/netflow/ [Accessed 10 Dec. 2019].