Efficiently Implementing Max Flow and Bipartite Matching for Stanford Network Analysis Platform

CS224W: Social and Information Network Analysis
Prof. Jure Leskovec

Fall 2013

Omede Firouz
Group 70
SUNET: omede
05547809

December 10, 2013

Abstract

The Max-Flow/Minimum s,t-Cut problem is one of the most well-studied and practical combinatorial optimization problems. Algorithms for solving the problem have a rich history of both theoretical and practical improvement. We have implemented performant implementations of both Push-Relabel and a k-regular Bipartite Perfect Matching algorithm of Goel (a problem that can be reduced to max-flow). Our implementations scale to the millions of edges, depending on graph type. We provide benchmarks, and our results are on the same order of other high performance free libraries, although slower than the best commercial implementations. We implement and use a novel vertex selection rule, which appears to be more robust to graph types on the test sets used.

Introduction

Max-Flow algorithms are highly pervasive in Segmentation, Clustering, and Routing. For example, Max-Flow algorithms are often used to find the minimum cut on graphs, which can be used to segment images and cluster. Max-Flow is often a subroutine in many other problems. For example, in the Traveling Salesman Problem and other integer programs, Max-Flow is often used to find the minimum separating cut between nodes, which is the tightest cutting plane that enforces connectivity.

Many optimization problems can be reformulated as network flow problems, such as bipartite matching. In Operations Research, network flow algorithms are often used for routing. For SNAP use cases, a max flow between two nodes would tell you the the minimum number of edges that must be cut to separate the two nodes (by the Max-Flow/Min-Cut Theorem, a measure of robustness in a network.

This project has explored both Max-Flow and the related problem of bipartite matching. Our results are mostly practical, rather than theoretical. However, the results on the practical efficiency of randomized matching in k-Regular Bipartite graphs is to the best of our knowledge novel.

Implementation Details

A conscious decision was made to not use existing code for the project. This was a significant handicap, since implementing Push-Relabel from scratch with all the requisite heuristics was quite challenging. However, had we chosen to modify academic code, the results would conflict with SNAP's license and not be broadly usable. Unfortunately, this meant that several advanced optimization techniques such as parallelization were out of reach for the project, since there was just not enough time to implement the algorithm from scratch and also try out these methods.

Therefore, Push-Relabel was implemented from scratch and tailored to use SNAP's functionality. The BOOST C++ Graph Library has a Push-Relabel implementation released under a non-restrictive license, which was consulted to clarify tuning constants during the implementation stage. The code to generate random k-regular bipartite graphs and randomized matching was also written completely from scratch and is so planning to be released under a non-restrictive license.

Organization

The rest of this paper is organized as follows:

In section 1, the required theoretical background is given, and should be helpful to understand the new developments.

In section 2 we discuss methodologies used to collect data and generate the required graphs.

Section 3-1 gives a survey of past results and heuristics as they relate to the Push-Relabel algorithm.

In section 3-2, we discuss the new heuristics that were tested and found to improve the performance of our implementation. We also present the results of our implementation against the "Gold Standard" commercial implementation, as well as compared to popular open source libraries.

Finally, in section 4 the Max-Flow implementation is used to solve the k-regular bipartite matching problem. Results are shown comparing the Push-Relabel algorithm to a randomized bipartite matching algorithm of Goel (6), which performed extremely well under certain conditions.

Contributions

The main contributions include high quality implementations of Push-Relabel and a randomized algorithm of Goel to solve k-Regular Bipartite Matching. These implementations are planning to be released to SNAP after some minor refactoring.

We also contribute new results on the computational efficacy of Goel's bipartite matching algorithm as well as a new vertex selection rule that was not seen in the literature, and found to be more robust to differing graph types. We summarize the results and present directions for improvement.

Finally, we show that the popular tuning variables and heuristics used are not necessarily ideal under all conditions, and show significantly different performances for different graph types.

1 Preliminaries

The max-flow problem is given a graph $G = (V, A)$ with capacities $c_e \forall e \in A$, find a feasible set of flows that maximizes the total flow out of the source, $\sum_i f_{s,i}$. A flow is called feasible if $f_e \ \forall \ e \in A$ such that $f_e \leq c_e$ and $\sum_v f_{u,v} = 0 \ \forall u \neq s, t$. That is, given a graph with capacities, we wish to push flow on the capacities such that flow is conserved at nodes and satisfies capacity constraints.

The two main approaches to Max-Flow are primal and dual approaches. Primal approaches such as Ford-Fulkerson, Network Simplex, and Edmonds-Karp were discovered first and are conceptually simpler to grasp. They work by maintaining feasible flows, and augmenting these flows in some way until no more augmentations are possible, at which point the algorithm is provably optimal.

The dual approach to Max-Flow is used by Push-Relabel and HPF and work by maintaining a superoptimal solution that violates constraints and slowly fixing constraint violations.

Push-Relabel specifically works by assigning 'height' labels to all nodes, and initially pushing as much flow as possible from the source. Then, as the algorithm iterates, a node is selected which has violated constraints, and is allowed to 'discharge' some of its flow to a neighbor in order to improve the violations. However, it is only allowed to discharge to neighbors of lower height than it. If it is unable

to discharge, we can simply raise its height to slightly larger than the smallest of its neighbors.

Implementing Max-Flow efficiently is a well-studied problem in the literature (1,2,3,4,8). Specifically, Push-Relabel has been studied in the context of many variants, heuristics, and optimizations. The culmination of these studies has resulted in Push-Relabel being among the fastest Max-Flow algorithm in practice, specifically the code of Goldberg which was obtained as a benchmark. Push-Relabel's exact runtime depends on choice of data structure, but is most commonly mentioned as $O(V^3)$. In practice, it runs significantly faster with heuristics.

Recently, a couple very fast primal algorithms have been discovered (8,9,10). Boykov and Kolmogorov have discovered a primal algorithm that outperforms most other implementations in the special case of computer vision algorithms (8), where a picture's pixels form nodes, and weighted edges exist between adjacenty pixels. It is worth noting that this is a very particular type of graph that is extremely sparse (4 or 9 edges per node). Furthermore, Boykov and Kolmogorov have not provided any polynomial time bounds for there algorithm. This led to Goldberg in (9) discovering an even more efficient (10) primal-based algorithm that is theoretically justified with a polynomial worst case.

Finally, we note that Goel's randomized matching algorithm is essentially a random truncated DFS, and is such a primal-based method suited to matchings.

It is worth mentioning that although there are asymptotically faster algorithms available for max flow, they look practically slow and difficult to implement, so were omitted. For example, see (2).

2 Data Collection

All time measurements were taken with the C function time() and did not include time to initialize the graph representation, which was very significant especially for the bipartite graphs.

Data was synthesized using random graph generators. Specifically, the generators RMF-Long and RMF-Wide were selected based on their use in (1,2,3,4). Although clearly using more generators would provide a more nuanced picture of overall running speed, we decided that for now two generators is good enough to get a feel for overall efficiency. Data was synthesized using the same parameters used in (4), and includes graphs with up to 650000 nodes and 3.2 million edges.

RMF-Long was chosen, rather arbitrarily, among the given generators. The alternatives were dismissed due to being either 1. dense graphs, 2. pathological graphs for max-flow like Washington Graphs. Both of these were dismissed since the graphs most commonly used in SNAP will not be pathological or dense. Among the remaining, RMF-Long and RMF-Wide remained.

The generator used was found on Google Code and claimed to be the same generator used in (4). The code was released under the Lesser GPL and so could be used

in this project.

RMF graphs are parameterized by two values, a and b, and are several square grids of side length a adjoined b times. Wide graphs are as you would expect, and the grids are relatively large compared to the total distance. Long graphs have a large number of relatively small grids.

2.1 Generating k-Regular Bipartite Graphs

Since bipartite graphs are relatively simple, no external graph generator was used. Instead, we coded a highly performant generator which takes $O(nk) = O(m)$ time, which is clearly asymptotically optimal. For clarity, call the nodes in one partition white, and black on the other. The key insite is noting that for each white node, we are selecting k black mates, without replacement, from a bin of k copies of each black node. This can be implemented in linear time using a variation on the Knuth shuffle.

```
potentialMates = range(0,n)
numberPotentialMates = n
for u in range(0,n):
    for i in range(0,k):
        v_i = rand(0, numberPotentialMates)
        v = potentialMates[v_i]
        add_edge(u, v)
        timesMates[v] += 1
        if timesMated[v] == k:
            numberPotentialMates -= 1
            potentialMates.swap(v_i, numberPotentialMates)
```

3 Results

3.1 Known Heuristics

In practice, many details that do not contribute to asymptotic efficiency still heavily effect the actual experimental runtime. For example, Push-Relabel has been heavily studied in the context of heuristics and variants for real world performance (1,2,3,4).

The naive implementation of Push-Relabel is quite slow in practice, and following the advice of (3) we found a significant speedup by implementing the Global Relabeling Heuristic. The Global Relabeling Heuristic works by, at specified intervals, running a backwards breadth first search from the sink on the residual graph. Then, the heights of each node are set to the distance from the sink. The main invariant that a labeling must maintain is that the heights of adjacent neighbors can be no more than one apart, and therefore this will clearly be a valid labeling. Since the

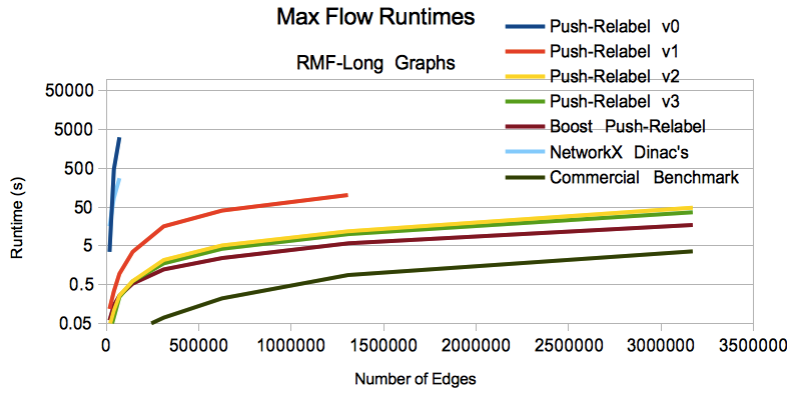
search will take linear time, which is much more expensive than a local relabeling operation, we only run it intermittedly, on the order of every $O(m)$ relabels.

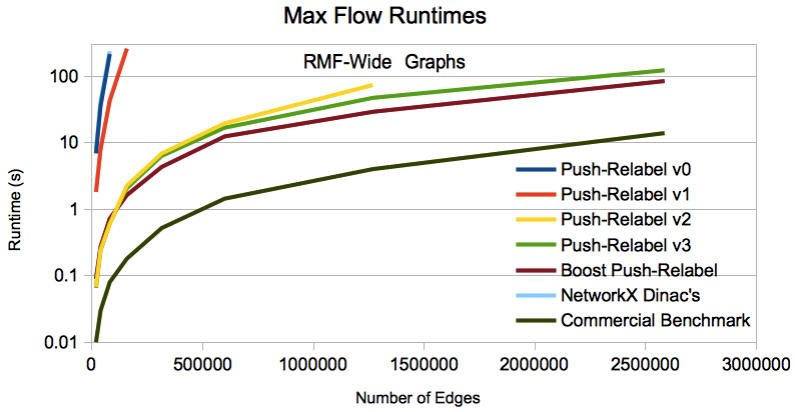
Although there are no theoretical justifications, the intuitive justification is such a global relabeling re-embeds global information into what is otherwise a very local algorithm. If the algorithm erroneously pushes flow away from the sink it is very costly for the flow to move back, and the heuristic avoids that. In practice we found about a factor of 100 speedup by using this heuristic.

The other literature heuristic we found was that of gap relabeling. Although not as dramatic as global relabeling, gap relabeling is achieved by noting that during the iteration of the algorithm, if at any point there is not a node of height u, then any nodes of height greater than u must clearly be disconnected from the sink. These nodes will therefore not contribute any more flow to the sink, and can safely be removed. In practice, this decreased runtime by a factor of 3-4 depending on the exact other details. Gap Relabeling was the most significant coding challenge within the project, as all the inactive nodes as well as active nodes must be handled at all times. The BOOST library used linked lists, one for each layer or height. Each node held a pointer to its' list element to allow for efficient random access and deletion. Our implementation mirrored this, except as we will discuss under the new heuristics section.

Finally, we investigated several different node prioritizations. We reaffirmed Goldberg's result that processing the nodes with highest label first is indeed the fastest. The other alternatives tried were lowest label and a "label ignorant" approach. The difference in these algorithms was also so quite small (factor of 2-4) , and is not shown here for fear of overburdening the reader with data.

We show below our successive implementations compared to a couple benchmarks. The NetworkX Library uses an implementation of Dinac's algorithm, also known to be very fast in practice. The BOOST library is in C++, and generally considered a very high quality library among C++ users. The commercial implementation is that of Goldberg, and was chosen due to its wide use in benchmarks. Although newer algorithms such as HPF and IBFS are also quite fast, they are not significantly more so, and we decided to compare to Goldberg's code for its wide use in benchmarking. Both the BOOST library and the Goldberg's implementation use the Push-Relabel algorithm.





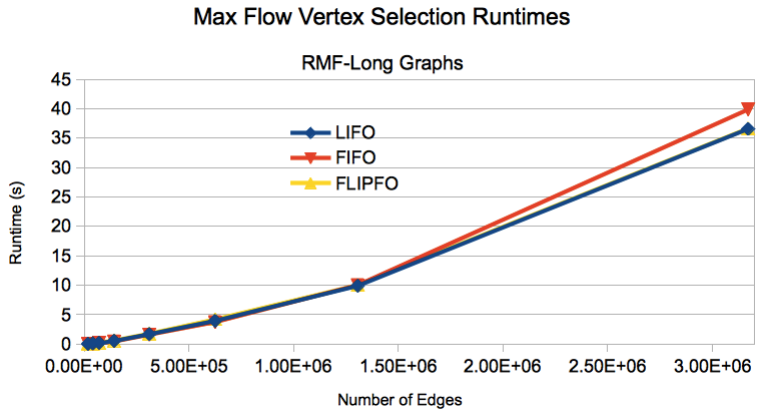
The versions are as follows:
v0 is the naive textbook implementation.
v1 adds global relabeling.
v2 adds gap relabeling.
v3 adds highest label selection instead of label-ignorant, in addition to other small optimizations.

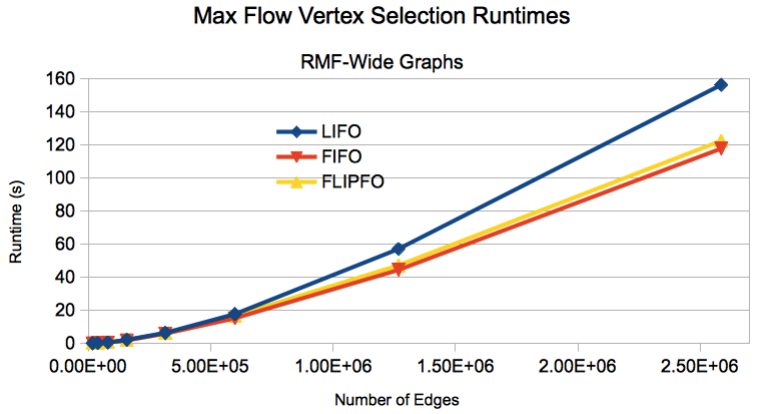
Since our best implementation, BOOST, and the commercial implementation are parallel on the log scale plot, this indicates that our results are a constant factor away from the best known. This reaffirms our implementation, since the constant factor is most likely due to the graph representation itself. The commercial representation does not offer any functionality other than the max-flow itself. Compared to SNAP, it is clear that increased functionality adds an overhead to the representation. Small optimizations such as inlining and reversing the order of "if" clauses were leading to 5-10% time savings, and so it is clear that the precise implementation is very significant if the goal is the utmost performance.

3.2 New Heuristics

Even with a node prioritization rule, such as highest label, there is the matter of what order to process a node within a label. The implementation used in BOOST library followed a Last In First Out (LIFO) rule, implemented by keeping a linked list of active nodes for each layer and appending to the front whenever a new node becomes active. We measured LIFO as well as FIFO, and found LIFO performed best on RMF-Long graphs, but FIFO performed better on RMF-Wide graphs. The speed difference is significant, about 25% in both cases.

Inspired by this, we decided to implement a new vertex selection rule not seen in the literature which we call FLIPFO. The idea is to flip a coin, and if the coin is heads, append a node to the front of the queue. Else, append it to the tail. The idea was hopefully to combine the best of each method and find an even more robust method. We implemented a derandomized version of the algorithm which just alternated between head and tail appends, and it was found to perform better than either the simple implementations on our test set.



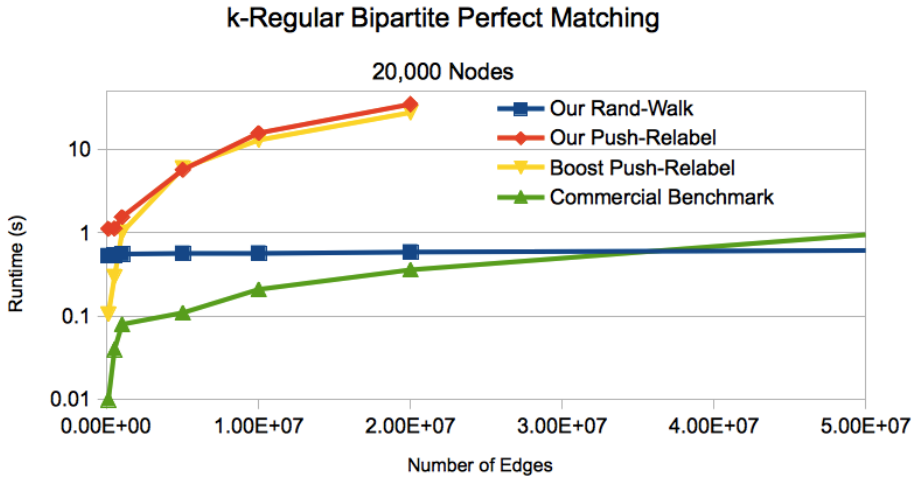


Omede Firouz
omede: 05547809

4 Bipartite Matching

A recent result of Goel (7) has found an $O(V \log V)$ algorithm for perfect matchings in bipartite graphs. The algorithm is relatively simple, and appeared to be potentially fast in practice. Therefore, we implemented the algorithm in SNAP and ran tests of random graphs. Since bipartite matching can be solved with Max-Flow, we compared the random walk approach to both our and the commercial code.

As can be seen from the results below, the random walk algorithm is very fast, especially as the graph becomes less and less sparse. We do not think this is due to the fact that a primal approach to matching becomes trivial as the graph becomes fully connected. Such an argument would not explain why the push-relabel algorithm cannot take advantage of the triviality of matching as well.



Unfortunately, we were only able to test the matching algorithms for graphs with relatively small numbers of nodes since we wanted to deal with dense graphs. Even though the matching ran quickly, loading the graphs into memory became prohibitively slow for dense graphs.

Conclusion

Despite theoretical bounds on runtimes, the actual practical performance of Flow algorithms is hard to predict. All the algorithms mentioned run much faster than their theoretical worst case. Small details such as internal representation make over an order of magnitude difference in runtimes.

This is quite a negative for graph libraries, since libraries such as SNAP implement lots of functionality, it is only natural that they cannot keep up with a custom-coded solution that only solves a single problem. In the face of extremely large optimization problems, custom software may be worthwhile.

The importance of heuristics cannot be overstated. Without global and gap relabeling, Push-Relabel is simply not tractable. Finally, choice of algorithm should reflect the problem. As was seen, certain flow algorithms (FIFO or LIFO) performed better or worse on different graphs. A stronger example is how the bipartite matching algorithm firmly trounced our flow-based approach to bipartite matching and was near in quality to the commercial-grade flow approach, even surpassing it on dense graphs. This is quite a significant development because it means that not only is Goel's algorithm fast in theory but it is fast in practice.

Further Questions

Adaptive tuning of implementation. As seen, FIFO and LIFO perform better on different graph types. What is interesting is that this effect is noticeable even at small graph sizes, and becomes more significant as the graph increases in size. Therefore, we propose an adaptive tuning method, where small graphs are used to optimize parameters and choice of selection rules so that large graphs of similar structure can be run efficiently. However, the use of the FLIPFO selection rule would seem to negate some of this advantage, since as was seen above, FLIPFO has on the measured graphs the best of both worlds.

An interesting question would then be to implement this new vertex selection rule on Goldberg's code. Since the algoriths are similar and only differ in implementation, one would expect the same results to hold which is quite a significant improvement for a gold standard. However, it is very likely that this vertex selection rule underperforms on many other types of graphs, so furhter tests are needed.

Finally, it would be interesting to see how much faster the randomized walk algorithm could be if explicitly tuned. One would expect a factor 10 at least improvement, which would place it firmly ahead of a flow-based approach using the best push-relabel implementation.

References

1. B. Chandran, D. Hochbaum. A Computational Study of the Pseudoflow and Push-relabel Algorithms for the Maximum Flow Problem. 2009. http://riot.ieor.berkeley.edu/ dorit/pub/Chandran-Hochbaum.pdf

2. R. Ahuja, M. Kodialam, A. Mishra, J. Orlin. Computational investigations of maximum flow algorithms. 1996. http://jorlin.scripts.mit.edu/docs/publications/58-comput%20investigations%20of.pdf

3. B. Cherkassky, A. Goldberg. On Implementing the PushRelabel Method for the Maximum Flow Problem. 1997. https://www.ads.tuwien.ac.at/teaching/archiv/praktika/CherkasskyGoldberg-1995-MaxFlow.pdf

4. U. Derigs, W. Meier. Implementing Goldberg's Max-Flow-Algorithm, A computational Investigation. 1989.
http://link.springer.com/article/10.1007/BF01415937#page-1

5. D. Hochbaum. Lecture Notes on Graph Algorithms and Network Flows. 2012.
http://www.ieor.berkeley.edu/ hochbaum/files/ieor266-2012.pdf

6. J. Orlin. Max flows in O(nm) time, or better. 2012.
http://jorlin.scripts.mit.edu/docs/papersfolder/O(nm)MaxFlow.pdf

7. A. Goel, M. Kapralov, S. Khanna. Perfect Matchings in O(n log N) Time in Regular Bipartite Graphs. 2010.
http://arxiv.org/pdf/0909.3346v3.pdf

8. Y. Boykov, V. Kolmorogov. An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision.
http://www.csd.uwo.ca/ yuri/Papers/pami04.pdf

9. Maximum Flows by Incremental Breadth-First Search.
A. Goldberg, et al. http://research.microsoft.com/pubs/150437/ibfs-proc.pdf

10. T. Verma, D. Batra. MaxFlow Revisited: An Empirical Comparison of Maxow Algorithms for Dense Vision Problems.
2012. https://filebox.ece.vt.edu/ dbatra/papers/vb_bmvc12.pdf

Appendix 1: Summarized Raw Data

a	b	Nodes	Edges	Flow Value	Our Runtime (s)	Ours / Commercial
8	64	4096	18368	27117	0.028261125	2.8261125
10	91	9100	41760	42987	0.072072625	7.2072625
11	128	15488	71687	53590	0.239100375	23.9100375
13	181	30589	143364	74822	0.603236625	20.1078875
16	256	65536	311040	1168984	1.759673	25.1381857143
19	362	130682	625537	162460	4.189110125	19.0414096591
23	512	270848	1306607	244854	10.016994875	11.3829487216
30	724	651600	3170220	424046	36.64859575	10.1801654861
28	5	3920	18256	3907817	0.065828625	6.5828625
37	6	8214	38813	6769701	0.241937625	8.0645875
49	7	16807	80262	11888755	0.613158875	7.6644859375
64	8	32768	157696	20206201	2.071028875	11.5057159722
85	9	65025	314840	35752489	6.334782875	12.1822747596
111	10	123210	599289	60958410	16.774528	11.6489777778
147	12	259308	1267875	107505248	47.138703875	11.7846759688
194	14	526904	2586020	187362312	122.44422825	8.7962807651