Max Flow in a Directed Planar Graph

Jason Hoch and John Wang
6.854 Final Paper

December 12, 2012

Abstract

Maximum flow is a classical problem in computer science and many incremental improvements have been made in the last decade. However, few authors have attempted to examine the practical results of many of these algorithms. We provide insight into max-flow implementations in industry. In particular, we examine the maximum flow problem on directed planar graphs, since they seem to have the most practical application. We implement an $O(n \log n)$ algorithm presented by Erickson 2010 [1] and examine its runtime in comparison with standard push-relabel algorithms. We also examine the theoretical and practical implications of using a naive spanning tree data structure, instead of the top-tree data structure suggested in [1], as a subprocedure in the algorithm. We find preliminary results that show the naive implementation outperforms the more complicated top tree structure in small to medium sized graphs.

Contents

1   Introduction

The general maximum flow problem has many applications in operations research and supply chain management. The classic algorithms used to solve the problem are polynomial, but relatively slow. The Edmonds-Karp algorithm requires $O(nm^2)$ time, while Ford Fulkerson requires $O(m \max |f|)$. Improvements made throughout the last few decades have decreased these runtimes by using concepts from blocking-flow and push-relabel algorithms. Dinitz's blocking flow algorithm improved the runtime to $O(n^2 m)$ by using a breadth-first-search in the residual graph to build a layered graph [2]. Goldberg and Tarjan [3] introduced a general framework for solving max-flow problems by using the idea of push relabel. Many of the fastest algorithms for maximum flow are variants of this general framework. Recently, Orlin presented an algorithm which runs in $O(nm)$ time.

A practical application of a maximum flow algorithm, however, would find many of these algorithms prohibitively slow. Even the best known algorithm of Orlin for general graphs is $o(n^2)$ for reasonably connected graphs. However, these algorithms perform poorly due to their generality. In practice, only a certain subset of graphs are likely to arise. For instance, in many applications of maximum flow, it is reasonable to assume that the graphs will be planar.

This is the case for maximum flow networks on any type of two dimensional map. In these cases, solving the problem in its full generality does too much work, and enforcing constraints on the assumptions of the graph can improve runtime. This paper focuses on the algorithm proposed by Erickson 2010 [1], which exploits planarity to achieve $O(n \log n)$ runtime. This significantly reduces the execution time of the algorithm, at least theoretically.

This paper implements the Erickson algorithm and presents the observed runtime on a set of planar graphs. The set of planar graphs generated contains graphs of varying sizes. Our implementation of the Erickson algorithm is tested for how it scales with n, the number of vertices in a graph. We attempt to determine an empirical function for how the observed runtime scales with n, and compare it with standard library implementations of max-flow algorithms.

The rest of the paper presents a brief overview of the Erickson algorithm, the unique features in our implementation, and the object oriented infrastructure we created. We present the results from our implementation on a set of planar graphs and compare it with standard maximum flow algorithms.

2   Motivation

In this section, we provide motivation for our implementation. In particular, we motivate the use of an algorithm specific to planar graphs. We argue that most applications of maximum flow are constrained to planar graphs, and hence, that the Erickson algorithm provides a non-trivial improvement upon past algorithms.

2.1   Supply Chain Example

First, suppose a chain of department stores has a supply chain and is attempting to find the best way to move goods from its production facilities to its stores. The store must transport the goods across the network from one of its production facilities to one of its stores. There are a discrete number of warehouses that its trucks can stop at and a discrete number of trucks that it sends over any route. Since this production facility and store pair is not isolated in its global network, the chain cannot add or remove any trucks or change any routes. The goal is to achieve the highest throughput of goods possible from the production facility to the store.

This problem can be reduced to a single source, single sink maximum flow problem in planar graph. If trucks are allowed to stop and readjust their loads at intersections of large highway networks, then there exists a planar embedding of this problem onto a two-dimensional graph. Placing a warehouse at intersections of large highways is also not very restrictive, since most retail chains rent out retail space from a large supplier. In other transportation problems, these can be thought of as large distribution centers such as ports or airports from which the company can rent out space for a low fixed cost.

Notice that this type of transportation problem occurs with incredible frequency. Every large chain of stores with a need to move goods across land has a similar problem. Retailers such as Amazon, Walmart, and Target have a large incentive to optimize their transportation networks, as do large shipping companies such as FedEx and UPS.

2.2 Computer Vision Example

There are also many examples of planar max-flow networks which are relevant in computer vision. In particular, finding graph cuts in an image is a widely used industrial application of max-flow. Greig, Porteous, and Seheult [4] provide an example of using max-flow for smoothing noisy images. They show that the maximum a posteriori estimate of a noisy image is given exactly by finding a max-flow in a specifically defined image network.

The authors in [4] create a network of pixels i, each with value $x_i \in \{0, 1\}$ corresponding to white and black. Each pixel i in the flow network has neighbors which are the pixels touching it in the actual image. The capacity of the edge between pixels i and j is defined as a log-likelihood ratio corresponding to the probabilitiy that the values of pixels i and j are correct. The paper shows that finding a maximum flow through the network results in a maximum a posteriori estimate of the image.

The network is clearly planar because the graph can be embedded onto a two dimensional plane (the image). Moreover, [4] spawned a large literature in the computer vision field using max-flows as a means to separate out nosie from images. Other max-flow algorithms in the computer vision field follow in a similar vein to [4].

3 Overview of Erickson's Algorithm

This section provides an overview of the Erickson algorithm. We summarize the main results from the paper in [1] and sketch the proofs of the main theorems. The Erickson algorithm relies on the observation that max-flow in directed planar graphs can be reformulated as a parametric shortest paths problem. The algorithm finds a parametric shortest paths spanning tree whose parameter is perturbed in multiple iterations of the algorithm. Eventually, the resulting tree will arrive at a state where the max-flow can be computed with simple arithmetic.

3.1 Definitions

First, we shall present the fundamental definitions and terminology used throughout the this paper and the Erickson [1] paper. Note that the notation is the same as in [1] to reduce confusion. A graph $G = (V, E)$ will be a directed planar graph where each edge e denoted as $u \to v$ contains a head and tail defined as $head(e) = v$ and $tail(e) = u$ respectively. The reversal of the graph is defined as $rev(e) = head(e) \to tail(e)$.

Each primal graph G has a corresponding dual G^* where each primal face is converted into a dual vertex. Two dual vertices are connected by a dual edge if and only if there is a corresponding primal edge connecting the two faces represented by the two dual vertices. In particular, consider faces a and b in the primal graph, and consider some edge e which

separates these two faces. Then the dual graph will contain an edge e^* (the dual of the edge e) which is oriented $90°$ counterclockwise to e and connects a^* and b^* in the dual.

A flow between source s and sink t is defined as a function $\phi : E \to \mathrm{R}$. The flow satisfies antisymmetry and conservation. In particular, the antisymmetric constraint specifies that $\phi(e) = -rev(\phi(e))$ while the conservation constraint specifies that $\sum_{e_w} \phi(e) = 0$ for all edges e such that $w = head(e)$. We define a feasible flow by defining a non-negative capacity function $c : E \to \mathrm{R}$ on each edge e. For a flow to be feasible on a graph, we must have $\phi(e) \le c(e)$ for all edges e in the graph.

3.2   Parametric Shortest Paths

Before moving on to Erickson's algorithm, it is instructive to examine the parametric shortest paths problem, since Erickson's algorithm is fundamentally based on the idea of using parametric shortest paths to compute the max-flow. For this problem, consider a graph $G = (V, E)$ and an additional parameter λ. We define a new cost function $c : E \to \mathrm{R}$ and obtain a subset $E' \subset E$ of the edges of the graph. The cost function will be set to $c(\lambda, e) = c(e) - \lambda$ for all edges $e \in E'$ and $c(\lambda, e) = c(e)$ for all edges $e \notin E'$. The parametric shortest paths problem asks to compute the largest value of λ for which the resulting graph with weights $c(\lambda, e)$ has no negative-weight cycles.

A number of algorithms that solve this problem use the concept of a pivot tree. Young, Tarjan, and Orlin [6] solve the problem in $O(nm + n^2 \log n)$ by constructing a single-source shortest paths tree. The distance used in the shortest paths tree is equal to the cost $c(\lambda, e)$ for each edge e, given a parameter λ. The Young, Tarjan, and Orlin algorithm starts increasing λ from $-\infty$. At some point, $d(\lambda, v) = d(\lambda, u) + c(\lambda, v)$ for some edge $u \to v$ which is not in the shortest paths tree. When this occcurs, the shortest paths tree incorporates the $u \to v$ edge and removes an appropriate edge e where $head(e) = v$ so that the tree remains a valid shortest paths tree. The algorithm continues until a cycle appears in the tree. Notice that at this point, increasing λ by any amount will cause a negative-weight cycle in the original graph because the cycle must be of weight 0.

This algorithm therefore computes the maximum λ for which the graph G has no negative-weight cycles. Erickson's algorithm will use the same concept and apply it to max-flow. In particular, Erickson uses the parameter λ to pivot edges in a shortest paths tree. Using a well defined set of weights, one can show that the resulting algorithm solves the max-flow problem.

We also define the dual residual network G_λ^* of some graph G for some parameter λ. It is defined on the dual graph G^* and has a cost function $c : \mathrm{R} \times E^* \to \mathrm{R}$ where $c(\lambda, e^*) = c(\lambda, e)$. Note that the edge e is well defined since each edge e has a corresponding dual edge e^* which is $90°$ counterclockwise from e and connects the faces which e separates. Notice that each edge e^* corresponds to exactly one edge, so taking the cost on the primal edge $c(\lambda, e)$ is well defined for each dual edge e^*.

3.3   Erickson's Algorithm

The algorithm that Erickson provides in [1] first creates a shortest-paths tree, then runs a parameterized shortest-paths algorithm. It uses a dynamic tree data structure to find pivots and remove them from the tree. To explain this algorithm, we start by examining co-cycles. Let P be a path from s to t in the primal graph, and set $\pi : E \to \mathrm{R}$ be a unit flow function. We set $\pi(e) = 1$ for $e \in P$, $\pi(e) = -1$ for $rev(e) \in P$, and $\pi(e) = 0$ otherwise. Now define a co-cycle as some subgraph $C \subset G$ for which its dual C^* is a simple directed cycle in G^*. Let $\pi(C) = \sum_{e \in C} \pi(e)$ be called the crossing number. Erickson shows that the crossing number is confined to the values $\{-1, 0, 1\}$ for any co-cycle C.

First, if n_{plr} is the number of times that P crosses C^* from left to right and n_{prl} for the opposite direction, Erickson shows that $\pi(C) = n_{plr} - n_{prl}$. This means that one can break down the calculation of $\pi(C)$ into cases. Erickson invokes the Jordan Curve Theorem and asserts that any co-cycle's dual C^* must partition the dual into two halves. If s and t lie in the same half, then the number of times that P crosses C^* from left to right is equal to the number of times it crosses from right to left since it must eventually reach t again. Therefore, $\pi(C) = 0$ in this case. In the second and third cases, s and t lie on different halves, so either $n_{plr} = n_{prl} + 1$ or $n_{plr} + 1 = n_{prl}$, depending on the orientation of s and t in the halves. This means that $\pi(C) = \pm 1$ in these two cases. This shows that $\pi(C) \in \{-1, 0, 1\}$.

Moreover, Erickson shows that every (s, t) cut has $\pi(C) = 1$, since P contains one extra edge in C than in $rev(C)$ (by the fact that $n_{prl} = n_{plr} + 1$). Now if C were not an (s, t) cut, then there would be some path from s to t which did not go through C. Since the original P has one extra edge in C than in $rev(C)$, this cannot be the case, so C must be an (s, t) cut.

Now, if one thinks about the residual network G^*_λ, one can see that λ serves as something akin to a parameter in linear programming, in the sense that its feasibility in the dual immediately corresponds to the feasibility of a flow in the primal with value λ. By feasibility in the dual, we mean to say that the shortest paths distances induced by the costs $c(\lambda, e^*) = c(e) - \lambda\pi(e^*)$ have no negative weight cycles. Erickson formalizes this with lemma 1.

Lemma 1. *There exists a feasible $s - t$ flow in G with value λ if and only if the dual residual network G^*_λ does not contain a negative cycle.*

Lemma 1 follows because of our insight that $\lambda \in \{-1, 0, 1\}$. Suppose G^*_λ contains a negative-weight cycle. Then one can decompose the cycle into its constituent edges. This implies that for a negative weight cycle C^*, one must have $\sum_{e \in C^*} c(e) - \lambda\pi(C) < 0$ by the definition of the costs. This means that $\pi(C) > 0$ since $c(e) \geq 0 \forall e$ and $\lambda \geq 0$. However, from our previous result, we know that $\pi(C) > 0$ implies that $\pi(C) = 1$, which occurs exactly when C is an (s, t) cut. This cut C, with capacity less than λ, does not allow for a feasible flow of value λ, since there is no possible way of getting from s to t without going over the cut (which does not have enough capacity).

Lemma 2. *λ_{max} is the first critical value of λ whose pivot introduces a directed cycle into T_λ.*

We define λ_{max} as the maximum value of λ for which G^*_λ has no negative weight cycles and is well defined. Note that lemma 1 implies that λ_{max} is exactly the maximum flow of the original graph G with capacities $c : E \to \mathrm{R}$. To show this, suppose there existed some flow of value $f > \lambda_{max}$. Then, it must be feasible so that by lemma 1, G^*_f does not contain a negative cycle. However, this implies that $f > \lambda_{max}$ is a value of λ for which G^*_λ has no negative weight cycles, which is a contradiction.

The algorithm computes T_λ, which is the shortest paths tree with parameter λ, computed on G^*_λ. Notice that using a version of the parametric shortest paths algorithm presented in the above section, one can obtain λ_{max} and hence the max-flow of G. This can be found using T_λ and by increasing λ and pivoting edges into T_λ until a cycle is found. Using lemma 2, once a directed cycle occurs in T_λ, it must be true that λ will be equal to the max flow in G.

The overall structure of the algorithm is now set. First, one computes the minimum-distance spanning tree in G^*_λ. Next, iteratively decrease λ and maintain the minimum-distance spanning tree with parameter λ. Eventually, a directed cycle will be introduced, and when that occurs, λ will be equal to the maximum flow value.

3.4   Formalized Algorithm

To make the above overview more formal, the pseudocode for Erickson's algorithm is provided below. First, however, we define the distance $dist(\lambda, p)$ for a vertex p in the dual with parameter λ as the shortest path distance from some arbitrary starting node o to p. Note that the shortest path distance is computed over the cost function $c(\lambda, e^*) = c(\lambda, e)$ as defined earlier in the paper (each dual edge retains the cost function of its primal). We define slackness on an edge $e^* = u \to v$ in the dual with parameter λ as $slack(\lambda, e^*) = dist(\lambda, u) + c(\lambda, e^*) - dist(\lambda, v)$.

The slackness of an edge in the dual provides a basis for how pivots in the shortest paths tree are introduced. In particular, Erickson notes that the algorithm has an invariant which keeps $slack(\lambda, e^*) = 0$ exactly when $e^* \in T_\lambda$ or when λ is a critical value corresponding to a new pivot.

We are now ready to present Erickson's algorithm in pseudocode (taken from Erickson [1]):

```
PlanarMaxFlow(G, c, s, t):
    Initialize spanning tree L
    While s and t are in the same component of L:
        LP = path in L from s to t
        min_edge = edge in LP* with minimum slack
        d = slack(min_edge)
        forall edges e in LP:
            slack(e*) = slack(e*) - d
            slack(rev(e*)) = slack(rev(e*)) + d
        L.remove(e*)
        if pred(head(e)) != null:
            L.insert(Edge(head(e),e)*)
        pred(head(e)) = tail(e)
    forall edges e:
        flow(e) = c(e) - slack(e*)
    return flow
```

Note that the algorithm runs in $O(n \log n)$ time when one uses dynamic tree data structures for maintaining the spanning tree. For instance, Tarjan and Werneck [5] present top trees which are able to do all necessary spanning tree operations in $O(\log n)$ time. These operations include determining whether two vertices exist in the same component of the spanning tree, finding the minimum weight edge in a path, and finding a path between two vertices (in addition to being able to insert and delete edges from the tree).

Erickson proves a $O(n \log n)$ runtime for the algorithm by bounding the number of pivots by $O(n)$ (see lemma 2.6 of his paper for more details of the proof). Once the number of pivots is bounded, there are only a constant number of $O(\log n)$ operations per pivot, which implies $O(n \log n)$ total time for the while loop. Initializing the spanning tree requires $O(n)$ for computing the dual and $O(n \log n)$ for computing shortest paths in a planar graph (by using Dijkstra's). The top tree can also be initialized in $O(n \log n)$ time using Tarjan and Werneck's initialization scheme.

4   Framework and Infrastructure

The Erickson algorithm was implemented in Java 1.7 and is available on `https://github.com/jrshoch/msmsmaxflow`. The architecture was created so that each piece of code was as modular and extensible as possible. Special emphasis was put on readable and self-documenting code so that the algorithm could be tweaked easily. Since design decisions usually favored writing

good code, some computational speed was lost. However, the most noticeable loss of speed due to the object oriented architecture would prevail across all graph algorithms under our particular framework, which means that each algorithm could be compared in relation to each other.

4.1   Design Decisions

We chose to implement the Erickson algorithm in Java because of its object-oriented nature. Java represented a middle ground between performance and modularity. Implementation in a lower level language like C or C++ would have made it difficult to generalize the architecture, while a higher level language like Python or Ruby would have been too slow for automated testing at a large scale. Java provides enough object oriented functionality to make our architecture easily extensible and modular, as well as enough speed to be able to run the Erickson and other max-flow algorithms on large graphs.

The graph G and its dual G^* were abstracted away as objects so that all max-flow algorithms that we implemented would be able to operate under the same API. The graph class operates in relation to other objects, each representing a physical portion of a graph. Vertex was a fundamental object, with edges being composed of two vertices. Faces were then constructed by a cycle of vertices and edges.

Most of the work in the graph was handled by the Graph object. The graph contained information about neighboring vertices and served as a medium for finding and manipulating flows along edges. The original Graph interface created an immutable object so that vertices and edges were predefined and linked up together in the graph. The immutability enabled the algorithm to assume that vertices and edges would never get destroyed, which greatly simplified the amount of work for most algorithms. Moreover, this immutability improved the speed of the underlying data structure since only calls to the structure would be made. Flows were overlayed on top of the graph by creating subclasses which contained information about flow and capacity.

We tried to demarcate objects and methods in a manner which had natural analogies to graphs. Since a vertex, face, or edge without a graph does not have very much useful information, most of the knowledge was abstracted to the graph level. Neighboring vertices could be found only at the graph level, with each vertex completely unaware of its connections with other vertices. An edge had information about its head and tail vertex, but could not perform any operations.

Flow, slackness, and other additions to the graph and its subcomponents were represented by creating subclasses from the basic interfaces. This enhanced the modularity of the design. For instance, any breadth first search on the graph would only need to know about the vertices and edges, along with their neighbors, and would not be given informatoin about flow or slackness.

4.2   Dynamic Tree Structure

One aspect of Erickson's max flow algorithm that required optimization was the dynamic tree structure for the main while loop. In particular, one needs to be able to compute a number of attributes from the spanning tree. These attributes include finding the path LP between s and t in the tree, determining whether s and t exist in the same component of the spanning tree, finding the minimum edge on a given path, and adding a given value to all edges in a given path. All of these operations need to be done while the spanning tree is having edges added and removed.

A naive algorithm for this would be to simply keep a spanning tree, and adjust its edges

accordingly. There are $O(n)$ vertices and edges incorporated in this spanning tree, so most operations could be done in $O(n)$ time (such as adding and removing edges, finding a minimum edge on a path, and adding a given value to all edges in a path). Moreover, finding the path from s to t can be done in $O(n)$ by finding the least common ancestor of the two vertices s and t. One can implement this operation by walking up the edges of the tree in reverse order, so taking $tail(e)$ for each each e leading into a vertex along the path to either s or t. Eventually, $tail(e_1)$ will be equal to $tail(e_2)$ for vertices e_1 and e_2 corresponding to the vertices along the path of s and t to the root.

This naive data structure would therefore be able to complete each while loop in the Erickson algorithm in $O(n)$ time, leading to a total of $O(n^2)$ time for entire algorithm. Theoretically, this erases all the gains made by assuming planarity (since Orlin's algorithm achieves $O(n^2)$ on planar graphs). However, in practice, the constants on using the naive data structure are likely very small, especially in comparison to more complicated dynamic tree structures which perform all of the above operations in $O(\log n)$.

This paper, then, examines the constants associated with both the naive spanning tree data structure and the more complicated top tree data structure presented in [5]. At some point, the top tree data structure will overtake the naive algorithm due to its superior computational complexity. However, a priori, it is unclear whether this point occurs within a reasonable graph size.

In practice, it would be nice the magnitude of the graph size at which the crossover occurs. If crossover occurs for graphs with millions of vertices, in practice it would be much more advantageous to implement Erickson's algorithm using the naive spanning tree due to the higher efficiency and lower development time.

Therefore, we implement Erickson's algorithm using both the naive and the top tree data structures for storing spanning trees. The actually use of the data structure was abstracted away so that the only change in the Erickson algorithm code when using one data structure or the other is the initialization of a different class of spanning trees.

This abstraction allows us to test the Erickson algorithm using the two data structures without having to worry about interactions due to other parts of the code. In essence, we can isolate the differences in run time which come directly from implementing the spanning tree with different data structures.

4.3   Automated Testing

To examine the runtime of different max-flow algorithms, we generated planar graphs of different sizes. A parallelized graph generator and testing framework called SLOTIN was created to examine the runtime of Erickson's algorithm with different implementation details. SLOTIN allows one to specify the distribution of graphs to test, the most commonly used being a uniform distribution over the number of vertices n in a graph where $n \in [a, b]$. SLOTIN first creates graphs in the lower range of the distribution and continues to create larger and larger graphs. Graph creation is done in parallel while two versions of the Erickson algorithm are running. The first version uses the naive implementation of the spanning tree while the second version uses Tarjan and Werneck's top tree data structure to implement the spanning tree. There is a final max-flow algorithm, which is a standard implementation of push-relabel.

The current graph generation algorithm is an open source planar graph generator written in Java. The generator, located at `http://sourceforge.net/projects/planargraph/` generates planar graphs from a Voronoi diagram package. The advantages of this particular generation algorithm are that it is easily interfaced with our fundamental graph classes. In particular, the library outputs graphs in a standard format, and a simple analysis showed that a graph of N thousand vertices can be produced in N seconds. The generator allows one

to specify the number of vertices in the graph. However, the library cannot generate graphs above about 12,000 vertices without memory errors and also runs slowly. Another downside of the library is that points are returned in an unnatural manner (a pair of coordinates in the cartesian plane) and the density of the graph is not exposed as a parameter.

As a first step for graph generation, however, this small library is sufficient to create a set of graphs of differing sizes. The generation algorithm will be run in its own thread and will place its results in a producer queue.

Each created graph is put into the queues for both versions of the Erickson algorithm and the standard max-flow algorithm. The graph producer process finishes after it has generated some specified number of graphs from its given distribution. The SLOTIN program finishes when all of the queues are emptied, but provides periodic summary statistics on the runtimes of each program with respect to different size graphs.

This automated testing structure allows us to examine the performance of multiple max-flow algorithms over the same graphs. In particular, we can vary the types of graphs that the set of max-flow algorithms receives by changing the distribution given to SLOTIN. This allows us to examine the effectiveness of each max-flow algorithm over different sets of inputs and identify which algorithms are faster, depending on context.

5   Results and Analysis

This section examines the performance of the Erickson algorithm in comparison to other standard max-flow algorithms. This section attempts to provide practical guidance for people attempting to implement max-flow algorithms. Because of this, we also take into account the number of lines of code associated with each algorithm, where the lines of code *loc* are defined recursively. The next subsection provides the methodology for how we count the lines of code, and we proceed to present the results from our SLOTIN testing framework.

5.1   Counting Lines of Code *loc*

Since each algorithm uses the fundamental data structures, such as Graph, Vertex, and Edge in a different manner, we need to control how each algorithm used each data structure. In particular, consider an algorithm which uses a specialized method on a graph that no other algorithm uses. The developer of the graph interface would have to spent some amount of extra time solely to cater to that algorithm's needs. Therefore, we define the number of lines of code associated with a method i in a particular class c using the following formula:

$$loc_i = raw\_loc_i + \frac{1}{4}raw\_loc_c + \sum_{j \in i} loc_j \tag{1}$$

Here, raw_loc_i is the total lines of code found in method i, while loc_c is the total lines of code found in class c (where i is a method in c). The sum term $\sum_{j \in i} loc_j$ adds the *loc* of all methods j which are called from method i (and are not recursive calls). This simple formula gives a crude measure of the amount of work a developer would need to invest into creating a given max-flow algorithm. By examining performance results with development time, we can achieve a better understanding of which algorithms are best implemented in practice. Take the hypthothetical instance when algorithm x is 5% faster than algorithm y on the average graph found in practical applications, but algorithm x requires 500% more development time. Different developers will implement different algorithms, depending on the scope and goals of the project. This paper provides useful information to these developers in the context of maximum flow algorithms.

5.2 SLOTIN Results

The final results from the SLOTIN framework are still being generated. However, preliminary observations provide a number of interesting patterns. First, the implementation of Erickson's algorithm using the naive spanning tree data structure is faster than the implementation with the dynamic top tree structure for almost all graph sizes tested. Of course, we were unable to test large graphs (where $n > 12,000$). However, this result provides preliminary evidence that the naive implementation is more useful for applications.

In particular, the top-tree data structure has a *loc* which is far greater than the *loc* of the naive spanning tree structure. At least for small to medium sized graphs, it does not make sense to implement a Tarjan and Werneck's top tree data structure.

6 Conclusion

This paper presented the Erickson algorthim from [1] for finding max-flows in directed planar graphs. We presented motivation for the usefulness of this algorithm and summarized its main results, along with the algorithm's operation and runtime. Next, we implemented the Erickson algorithm in two different flavors, and compared these to a standard push-relabel algorithm to solve max-flows.

We created SLOTIN, our own automated testing framework for examining max-flows in directed, planar graphs. The framework generates planar graphs and tests an arbitrary number of max-flow algorithms on each planar graph. The results from the automated testing were found and analyzed, using the number of lines of code as a benchmark for the amount of development time required. We found that the implementation of the Erickson algorithm using the naive spanning tree data structure outperformed the implementation used Tarjan and Werneck's top tree data structure for small and medium sized graphs. These results, however, are still preliminary.

References

[1] Jeff Erickson. "Maximum Flows and Parametric Shortest Paths in Planar Graphs." *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*. 794-804. 2010.

[2] Yefim Dinitz. "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation." *Doklady Akademii nauk SSSR*. 11: 1277-1280. 1970.

[3] Andrew Goldberg and Robert Tarjan. "A New Approach to the Maximum Flow Problem." *Annual ACM Symposium on Theory of Computing*. Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. 136-146. 1986.

[4] D.M. Greig, B.T. Porteous, and A.H. Seheult. "Exact Maximum A Posteriori Estimation for Binary Images." *Journal of the Royal Statistical Society. Series B (Methodological)*. 51(2): 271-279. 1989.

[5] R.E. Tarjan and R.F. Werneck. "Self-Adjusting Top Trees." *Annual ACM Symposium on Discrete Algorithms.* Proceedings of the Sixteenth Annual ACM Symposium on Discrete Algorithms. 812-822, 2005.

[6] N.E. Young, R.E. Tarjan, and J.B. Orlin. "Faster Parametric Shortest Path and Minimum Balance Algorithms." *Networks.* 21(2): 205-221. 1991.