An Implementation and Empirical Analysis of Binary Blocking Flow Algorithm for Maximum Flow

Ankit Agarwal
aagarwal_601@berkeley.com

Chase Norman
c_@berkeley.edu

Vedaad Shakib
vedaad.shakib@berkeley.edu

ABSTRACT

We implemented and empirically analyzed the Binary Blocking Flow algorithm by Goldberg and Rao for max flow. Our code can be found here: https://github.com/chasenorman/BinaryBlockingFlow. This algorithm has a runtime of $O(m \min(\sqrt{m}, \sqrt[3]{n^2}) \log(\frac{n^2}{m}) \log(U))$ in the worst-case. We believe this to be the first implementation of the algorithm. We also developed a framework for empirical runtime analysis of different types of graphs while keeping different variables like number of nodes, edges, and capacities constant. We found that empirically, Binary Blocking Flow runs in linear time as a function of nodes and edges and logarithmically as a function of capacity. We also found the algorithm to be significantly slower in practice than the implementation of Dinitz' Algorithm in networkX.

1 INTRODUCTION

The maximum flow problem is a classic combinatorial problem that has been studied extensively since the 1950's. The first known solution to the problem was the Ford-Fulkerson algorithm in 1956 [4], which uses augmenting paths. Most modern solutions use either a blocking flow algorithm or a push-relabel algorithm. As an example, Dinitz, in 1970 [3], presented an algorithm that solves max-flow in $O(n^2m)$ by solving the sub-problem of finding blocking flows in $O(nm)$ time. The binary blocking flow algorithm presented by Goldberg and Rao in 1998 [5] was the first algorithm to improve upon the ballpark bound of $O(nm)$ for maximum flow. The binary blocking flow algorithm improves upon the algorithm of Dinitz with an alternate construction on graphs on which the blocking flow subroutine is run. The binary blocking flow algorithm achieves a worst-case runtime of $O(m \min(n^{\frac{2}{3}}, \sqrt{m}) \log(\frac{n^2}{m}) \log(U))$, where U is the maximum capacity of the graph, n is the number of vertices in the graph and m is the number of edges in the graph.

In this paper, we implement the Binary Blocking Flow Algorithm in Python using the popular `networkX` [2] library and analyze its empirical performance. In particular, we analyze how the algorithm performs with different blocking flow subroutines, one using the Dinitz subroutine, and the other using the subroutine defined by Goldberg and Tarjan [6]. We also compare each approach to other algorithms such as the push-relabel, Dinitz, and Ford-Fulkerson methods. We discuss implementation details and edge cases not considered by the paper and how we solved them, as well as concrete implementations of data-structures within the `networkX` library. Finally, we describe our implementation of the binary blocking flow algorithm. To our knowledge this is the first implementation of the algorithm. Our code can be found at this URL: https://github.com/chasenorman/BinaryBlockingFlow.

2 NOTATION

Let $G = (V, E)$ be a directed graph. For each edge $(i, j) \in E$ we assume $(j, i) \in E$ as well. The maximum flow problem takes in a directed graph, two starting vertices s and t, as well as function $\mu : E \to \{1...U\}$ that represents the capacity on edge a. A flow on G is a function $f : E \to \{1...U\}$ where for each edge a the capacity constraint $f(a) \leq \mu(a)$ is respected and for each vertex $v \in V \setminus \{s, t\}$, $\sum_{(i,j)} f(i, v) = \sum_{(v,k)} f(v, k)$. The maximum flow problem attempts to find a flow that maximizes $\sum_{(s,i)} f(s, i) = \sum_{(j,t)} f(j, t)$. A more concrete way of viewing this is imagining the graph to be a network of pipes, where each pipe can only handle a certain amount of water. What is the maximum rate of water we can siphon into s that is able to flow out of t while respecting each pipe's constraints. We also set the convention that either $f((i, j)) = 0$ or $f((j, i)) = 0$ since intuitively, flow in the direction (i, j) cancels with the reverse flow (j, i).

We define the *residual capacity* of an edge (i, j) with respect to flow f as $\mu_f((i, j)) = \mu((i, j)) - f((i, j)) + f((j, i))$. Note that when the flow is 0 at all edges (which is a valid flow), the residual capacity is equal to the capacity. The idea of the residual capacity is that once a flow $f(a)$ has been routed through edge a, further flows can route $f(a)$ flow back, if necessary, in order to produce a better maximum flow. Given a flow f, we can construct a residual graph G_f where the capacities are the residual capacities and the flow is 0. Solving this graph for max-flow f' is equivalent to solving for original graph of max flow and *augmenting* f' with f, where augmenting simply means adding flows on edges together, and reversing flows if necessary. In general for residual graphs, we only consider edges with **strictly positive residual capacity**, and call this edge set E_f.

We also consider a binary length function $\ell : E_f \to \{0, 1\}$. We let the *SP distance labeling* with respect to length function ℓ be $d_\ell : V \to \mathbb{Z}^+ \cup \{\infty\}$ as the shortest path distance to t on the graph with edge weights in ℓ, and infinity if the vertex is disconnected.

Finally, given a distance metric on the graph, we have the concept of admissibility. An edge (i, j) on G is admissible if it is on the shortest path from i to t, which is true if and only if $d_\ell(i) = d_\ell(j) + \ell(i, j)$. Similarly, an admissible path is a path made up of only admissible edges. This gives rise to the notion of a **blocking flow**, which is simply a flow in which every admissible path from s to t has at least one saturated edge.

In order to achieve run-time constraints, we define $\Lambda = \min(m^{\frac{1}{2}}, n^{\frac{2}{3}})$.

3 OVERVIEW OF BINARY BLOCKING FLOW ALGORITHM

The idea of this algorithm is to route flow through the graph in phases. We keep track of an error bound in our maximum flow F. After each phase, we require that our bound be cut at least in

half. We terminate our algorithm when $F < 1$ as our use of integral capacities implies that we have found an optimal solution.

For each phase, we maintain a value $\Delta = \lceil \frac{F}{\Lambda} \rceil$ which will parametrize a length function on our graph. In each phase, we run a number of iterations such that at each iteration, we either route Δ units of flow through the graph or we find a blocking flow of value less than Δ. Within $O(\Lambda)$ iterations F will be cut in half, at which point we terminate the phase and recompute Δ to start again.

At each phase, our length function on the graph is defined as follows:

$$\ell((i,j)) = \begin{cases} 0 \text{ if } \mu_f((i,j)) \geq 3 \cdot \Delta \\ 1 \text{ otherwise} \end{cases}$$

Note: if $\mu_f((i,j)) = 0$, we discard the edge from the graph and do not consider it.

In each iteration, we first compute the length of each edge and the corresponding SP distance d_ℓ of each vertex from the end vertex t on ℓ. To do this, we use Dial's algorithm which is an $O(n+m)$ algorithm to compute shortest path distances when given a length function with constant bound. If we find a blocking flow on one iteration, augmenting by this flow will result in $d_\ell(s)$ increasing in the next iteration, thereby making progress.

Once this is done, we compute the minimum canonical cut with respect to our distance function. This is simply the minimum over all cuts S_k where edge $(i,j) \in S_k$ if $d_\ell(i) = k$ and $d_\ell(j) = k+1$. I.e S_k is the cut between vertices of distance k and $k+1$. The sum of residual capacities across a cut that splits s and t in the graph creates an upper bound for the residual max flow from s to t. There are only $d_\ell(s)$ cuts to loop over. Since we loop over all of the edges and $d_\ell(s) \leq n$, this takes $O(n+m)$ time. If the minimum cut is less than $\frac{F}{2}$, then we end the phase and reset the graph.

Otherwise, we proceed and find the **special edges** in the graph. These are edges (i,j) that satisfy the following property: $2\Delta \leq \mu_f((i,j) \leq 3\Delta$. $\mu_f((j,i)) \geq 3\Delta$, and $d_\ell(j) = d_\ell(i)$. If the edge satisfies the property, we will decrease its length to 0. This defines a new length function, $\bar{\ell}$, however note this has no effect on the distances, as $d_\ell(j) = d_\ell(i)$ and decreasing the edge weight to 0 does not change this.

Once this is done, we want to find a blocking flow over this graph. However there is an issue, since the notion of admissibility is not well defined on graphs with zero lengths. To combat this, Goldberg and Rao proposed contracting the 0-length strongly connected components into a single vertex, and then routing flow through this new condensed graph C. We can use any blocking flow algorithm for this. The condensed graph thus has flow that is blocking or is of value Δ. Note if the blocking flow has value greater than Δ, we can simply route the flow backwards through the condensed graph since it is guaranteed to be acyclic, as it is not possible to have a cycle where all the edges are length 1 and admissible.

We now need some way of translating this into a blocking flow or flow of value Δ in the original graph. If a flow is blocking in the condensed graph, any saturated contracted edges must saturate its composite edge members. Thus, any blocking flow in C must be blocking in the original graph. To create this flow, we route flow though each 0-length strongly connected component by constructing an in-tree and an out-tree in each component. The tree will be composed of only 0 length edges. Since each edge holds at least 3Δ capacity, it is clear that any Δ flow could be routed through the in tree, to a single vertex, and out the out-tree to all of the other vertices. Thus, given a Δ or a blocking flow on the graph C we are able to construct a similar flow on G. Note, it is possible that the condensed graph results in s and t being contracted into a single vertex. In this case, we need not bother with the blocking flow algorithm, as we can easily route Δ flow out from s and into t using the constructed in-tree and out-tree.

Once we have the flow, we augment our graph with the new flow value. And repeat iterations. The full algorithm is described here:

Algorithm 1 Binary Blocking Flow

```
1:  procedure BINARY BLOCKING FLOW(s, t, G)   ▷ Max flow on G from s to t
2:      F ← nU                                ▷ Maximum possible flow
3:      Λ = min(√m, ∛(n²))
4:      T = 0                                 ▷ Total flow value
5:      while F ≥ 1 do
6:          Δ ← ⌈F/Λ⌉
7:          for 1 ... 8 * Λ do                ▷ After this F must be cut in half
8:              Compute ℓ on all the edges                ▷ O(E)
9:              Compute d_ℓ with Dial's Algorithm         ▷ O(E + V)
10:             if Min Canonical cut < F/2 then
11:                 F ← F/2
12:                 break
13:             end if
14:             Find special edges and set length to 0.
15:             Condense 0-length SCCs of G into C.       ▷ O(E + V)
16:             Determine a blocking or Δ-value flow on C.
17:             T = T + Blocking Flow Value
18:             Translate C blocking flow into original.  ▷ O(E + V)
19:         end for
20:     end while
21:     return G, T.
22: end procedure
```

In order to show correctness and achieve the right run-time constraints, we need to show the following facts:

(1) Given a blocking flow f on the length function ℓ. After augmentation and computation of a new length function ℓ' (with the same value of Δ), we have $d_{\ell'}(s) > d_\ell(s)$.
(2) Given at most 8Λ iterations, where $\Lambda = \min(\sqrt{m}, \sqrt[3]{n^2})$, the augmented flow reduces F by a factor of 2.

We refer the reader to [1] for proofs of these facts.

4 BLOCKING FLOW ALGORITHM AND OPTIMIZATION

A critical subroutine of this algorithm is finding a blocking flow in an acyclic graph. The best known algorithm for this runs in $O(m \log(n^2/m))$ time by Goldberg and Tarjan [6]. For our analysis, we used an $O(n^2)$ algorithm discussed in this same paper. We also implemented the Dinitz blocking flow algorithm described earlier as $O(mn)$.

4.1 Dinitz Blocking Flow Algorithm

We implemented two blocking flow algorithms for this paper. The first algorithm was developed by Dinitz and is a key subroutine in the Dinitz' Max Flow Algorithm. In order to find a blocking flow, we consider two operations:

- `Advance`: This operation advances on the edge of a graph
- `Retreat`: This operation moves back an edge and deletes the edge on that graph.

We start at s and continue keep advancing until we either reach t or encounter a dead end. At this point we retreat. Note that this is the same algorithm as DFS except we stop once reaching t and we delete edges after traversing them. Once we find a path, we route the maximum amount of flow we can through that path by finding the minimum capacity across the edges. We then delete any saturated edges (there has to be at least 1). We then repeat the algorithm, keeping the deletions from the previous steps. For this algorithm, in order to avoid rerouting the flow if the blocking flow value was greater than Δ, we would only route a cumulative maximum of Δ over the course of the algorithm and we would terminate if the flow routed ever met this mark.

We note that there are at most $O(m)$ possible augmentations since each augmentations deletes an edge. For each edge if it is on an augmented path has one advance operation per path. If it is not on a path, then it has one advance and one retreat, and that is it. Since there are n vertices in a path, per path there are $O(mn)$ augmentations plus $O(m)$ deletions. So the algorithm runs in $O(mn)$ time.

Our implementation did not explicitly used `Advance` and `Retreat`, instead we relied on the DFS approach below:

Algorithm 2 Dinitz-Blocking Flow

```
1:  procedure Compute-Total-Blocking-Flow(s, t, G, F)   ▷ Blocking or Δ flow on G starting from s to t of max value Δ
2:      T ← 0
3:      while T < F do
4:          P = [s]                                     ▷ P represents the path through graph
5:          T += Find-Path-To-End(P, t, G, F − T)
6:      end while
7:      return T, G.
8:  end procedure
9:  procedure Find-Path-To-End(P, t, G, F)   ▷ Finds single path from s to t on acyclic graph G and augments max flow or flow of F. Returns the amount of flow sent.
10:     T ← 0
11:     a = P[−1]
12:     if a is t then
13:         F = min(F, min({c_e : e ∈ P}))              ▷ c_e is the capacity of e
14:         For all e ∈ P, f_e += F                     ▷ f_e is the flow on e
15:         return F
16:     end if
17:     for n ∈ Succ(a) do                              ▷ Loop through successors of a
18:         if c_(a,n) ≠ 0 then
19:             P.append(n)
20:             T = Find-Path-To-End(s, t, G, P, F)
21:             if T > 0 then
22:                 return T
23:             end if
24:         end if
25:     end for
26:     Delete edge (P[−2], P[−1]) if P[−2] exists.
27:     return 0
28: end procedure
```

4.2 Goldberg-Tarjan Blocking Flow

The second algorithm we implemented has a better bound and is cited in the original Binary Blocking Flow paper, and was invented by Goldberg-Tarjan. This algorithm maintains a structure called a *preflow*. This is a flow which allows vertices to have more inward flow than outward flow. We call the extra inward flow *excess* and denote it with $e_f(v)$. At the end of the blocking flow algorithm, the excess will be 0 at all nodes, resulting in a flow. We call vertices that have nonzero excess *active*.

We also maintain a property called *blocked*. A vertex is blocked if all of its outgoing edges are either saturated or connected to another blocked vertex. At the start of the algorithm, all vertices are marked *unblocked* and we will block them as the algorithm progresses.

The algorithm makes use of a few $O(1)$ flow operations:

- The *push* procedure operates on an edge (u, v) and sends as much flow as possible from u to v. This amount is the minimum of the excess at u and the remaining capacity of (u, v). We then update the excess accordingly. This will only be used when v is not blocked.
- The *pull* procedure operates on an edge (u, v) and removes as much flow as possible from u to v. This amount is the minimum of the excess at v and the current flow at (u, v). We then update the excess accordingly. This will only be used when v is blocked to remove its excess.

We use these operations to construct the following subroutine *discharge*. This will take any active vertex and make it inactive through a sequence of pushes and pulls.

This is described in the Goldberg-Tarjan paper as performing as many push and pull operations on v as possible [6].

To create a blocking flow, we will apply discharge operations on vertices in a specific order. We will maintain a vertex list L which begins as a topological sorting of the graph.

This procedure runs in $O(n^2)$ with a L as a standard list. Goldberg and Tarjan discuss how this may be improved with the use of more complex data structures. They suggest the use of link/cut trees for finding eligible edges. The also suggest representing L as a doubly-linked list of heterogeneous finger trees (as described in [7]). Heterogeneous finger trees appear not to have been implemented in practice. Even with a correct implementation, Goldberg and Tarjan imply that a vertex search operation must occur in $O(1)$ time, which does not appear to be the case for the doubly-linked lists as described [6]. We speculate that another data structure may be required in practice.

To use this algorithm in the binary blocking flow algorithm, we need a subroutine to route flow backwards. This can be done in $O(m + n)$ time with the following rerouting algorithm.

Algorithm 3 Discharge

```
procedure DISCHARGE(v)
    if v is unblocked then
        for all unblocked successors w of v do
            Push(v, w)
            if e_f(v) = 0 then
                return
            end if
        end for
    end if
    Mark v as blocked. ▷ Successors are blocked or saturated.
    for all predecessors u of v do
        Pull(u, v)
        if e_f(v) = 0 then
            return
        end if
    end for                          ▷ This line will never be reached.
end procedure
```

Algorithm 4 Goldberg-Tarjan Blocking Flow

```
procedure BLOCKING-FLOW(G)
    Initialize blocked as FALSE on all vertices
    Construct a flow with all edges from the source saturated.
    Initialize the excess accordingly.
    L ← topological sorting of G
    while there are active vertices do
        v ← first active vertex in L that is neither s nor t
        Discharge(v)
        if v is blocked by Discharge then
            Move v to the front of L
        end if
    end while
end procedure
```

Algorithm 5 Rerouting Algorithm

```
procedure REROUTE(G)
    if |f| ≤ Δ then
        return
    end if
    e_f(t) ← |f| − Δ
    for v in reverse topological order do
        for predecessor u of v do
            Pull(u, v)
        end for
    end for
end procedure
```

5 IMPLEMENTATION DETAILS

Our algorithm was implemented in `networkX` which is a pedagogical library aimed for basic graph computations and graph data structures. We chose this library mainly because of its simplicity and full implementation in python. It also contains functions to generate complex graphs to test our algorithm. Our main goal for this project was simply to implement the algorithm and verify its correctness and practical run-time. It was not to compete to develop the fastest implementation of max-flow, which would likely require writing the algorithm in a different, more scalable language.

5.1 Main Data Structures

Our main data structure used was the `networkX` graph itself. `networkX` has an API so that it is possible to store arbitrary context data along each edge and vertex in the graph. For edges we would store the capacity and current flow value in the graph. We would also store the length function as well as whether or not the edge was a special edge. For the vertices, we would store the distance of each vertex. These values would get updated per iteration. To compute the residual capacity, we would simply compute $\mu((i,j)) - f((i,j)) + f((j,i))$.

For graph contraction, we created a new `networkX` graph to store values related to the contracted graph. We did this using Tarjan's algorithm for strongly connected components that runs on $O(n+m)$ time. This allowed us to separate the blocking flow subroutine cleanly from the rest of the algorithm and create an abstraction barrier between the two functions. Each strongly connected component represented a single vertex in the graph contraction. In each strongly connected component, we stored a list of its members as well as a representative vertex from which to direct flow. For each contracted edge, it is possible that multiple edges were fused to make this new contracted edge, so we also stored a list of the edges that made this up. For contracted edges, we set their capacity to be the sum of the residual capacities of its member edges and initialized the flow to 0.

In the original, we also stored our in-tree and out-tree when we needed to. The reason for this is that this allowed our algorithm to use a single data structure for all of our computation, making it simpler for the overall implementation. For each vertex in a strongly connected component, we stored lists of vertex integers for their `in_children` and `out_children`.

We used these data structures heavily when translating flow between the two graphs. We would first take all the contracted edges and loop through their child edge members and for each edge (i, j) in the original, we would accumulate the flow out of i and into j in the original graph data structure. We would also set the flow into the start vertex and out of the end vertex to be the total flow routed. Then, we would loop through each component in the graph and use the `in_children` and `out_children` sub-tree structure to route flow within the component, first routing incoming flow up the in-tree to the representative vertex, and then routing out of the representative vertex through the out-tree.

5.2 Main Functions Created

In accordance to Algorithm 1, we created 5 main sub-functions for in our main for loop within our algorithm:

- `blocking_flow`: This subroutine computes a blocking flow on the contracted graph. This should be the only non-linearity in our analysis.
- `min_canonical_cut`: This subroutine computes the minimum canonical cut given the distance and length functions on the graph. This algorithm runs in linear time.

- `construct_distance_metric`: This subroutine constructs the distance values on the vertices from the length function on the edges. This was implemented with Dial's Algorithm. This algorithm runs in linear time.
- `graph_contraction`: This function contracts the 0-edge weight strongly connected components into a single node. This algorithm runs in linear time.
- `translate_flow`: This subroutine takes in the contracted and original graphs and routes flow in between the two. This should run in linear time.

We analyze these sub-functions in our analysis.

6 TESTING AND DEBUGGING INFRASTRUCTURE

6.1 Debugging

There were many ways we debugged our code, but we needed to have tools to ensure correctness and ensure run-time. The first tool we built, mainly for the very start of the algorithm was a graph visualizer. The visualizer was able to take in a graph G as well as dictionary keys to access any edge or vertex attributes. The resulting visualized graph was one which had distinct blue vertices and red edges. The attribute value for each vertex would be displayed on the blue vertex and the edge attribute value would be displayed alongside its corresponding edge. This basic piece of infrastructure helped debug during the initial development of the algorithm and helped us understand any misconceptions present in creating the algorithm.

6.2 Testing

To ensure that the algorithm we implemented was correct, we developed a comprehensive testing framework using `pytest`. We used `networkX` to generate different random graphs of varying sizes and capacities and ran them through our Binary Blocking Flow Algorithm. To check correctness, we used an already-implemented max flow algorithm in `networkX` and compared our results against theirs. Note for all of our tests, we used the Goldberg-Tarjan blocking flow algorithm, instead of the Dinitz Algorithm.

6.3 Profiling

We wanted to run a profiler on the Binary Blocking Flow Algorithm to see what parts of the algorithm were consuming the most time in practice. We ran the profiler on complete graphs with varying numbers of nodes. An abridged version of the results can be seen in Figure 1. As it turns out, the blocking flow, which is the theoretical bottleneck, does not take up that much time practically! This is because, for most d-regular graphs and complete graphs, looping over the edges is much more expensive. This is an operation done quite often by Tarjan's SCC algorithm. Another reason for this is that this algorithm is called on a contracted graph, which may have significantly fewer edges on average relative to the main graph. Finally, both the Dinitz and Tarjan blocking flow algorithms are known to be fast in practice, and rarely achieve their runtime bottlenecks. Improvements to the practical performance of this algorithm should therefore target the construction of the contracted graph and of the distance metric. Perhaps with a modified graph data structure, or some dynamic programming solution, these constructions could be made easier. The use of Python dictionaries with string keys may be slowing these subroutines.

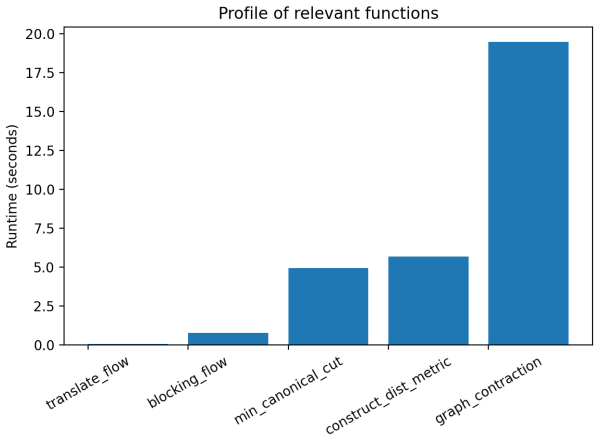

Figure 1: Profile of important functions

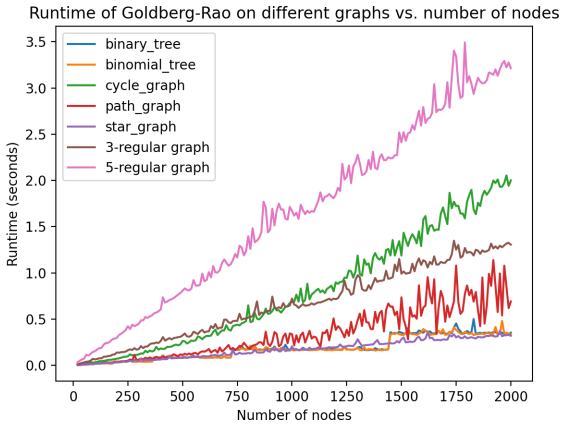

Figure 2: Runtime vs. number of nodes for special graph types for the Binary Blocking Flow Algorithm

7 EMPIRICAL RUNTIME ANALYSIS

We also conducted an empirical runtime analysis to see how the bounds provided by the authors held up in practice. First, we ran the algorithm on seven special types of graphs—a binary tree, a binomial tree, a cycle graph, a path graph, a star graph, a 3-regular graph, and a 5-regular graph. For each, we varied the number of nodes, while keeping the capacities constant, and plotted their average runtime as a function of number of nodes. The results for this analysis can be seen in Figure 1. As expected, the star graph, path graph, binomial tree, and binary tree performed the best, because they have roughly the same number of edges as nodes. Interestingly, the cycle graph performed worse than these graphs, even though it also has roughly the same number of edges as nodes.

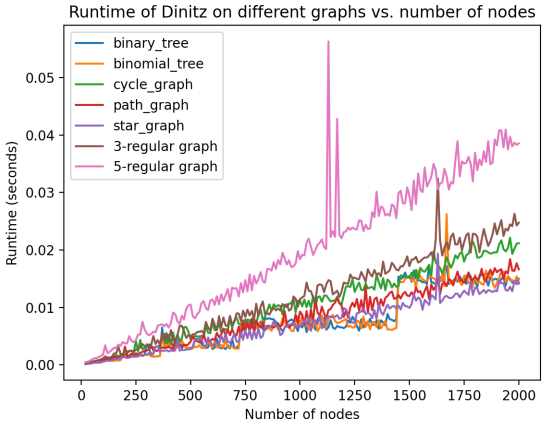

Figure 3: Runtime vs. number of nodes for special graph types for Dinitz' Max Flow Algorithm

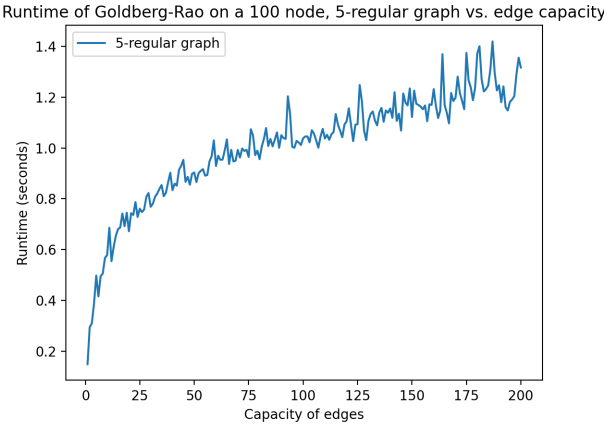

Figure 4: Runtime vs. edge capacity, with everything else constant

As expected, the 3-regular and 5-regular graphs had the worst constants in the runtime.

In Figure 2, we see a graph of the same analysis done for the `networkX` max flow algorithm, which uses Dinitz' Max-Flow algorithm. As seen, the Dinitz' Max Flow algorithm is also linear in practice, although it is $O(n^2m)$ in theory. Additionally, the `networkX` implementation of Dinitz' Max Flow algorithm runs at two to three orders of magnitude faster than our implementation of Binary Blocking Flow, even after the optimizations we made to speed it up.

The reason for this is likely because most practical graphs have their "non-linear" subroutines take up little of the runtime. We can see this directly by looking at the profile in Figure 1. The `blocking_flow` algorithm has virtually no time spent versus the other linear sub-components of the algorithm. What this means is that simpler algorithms with fewer linear subroutines may perform better than an algorithm with a better theoretical asymptotic run-time. Binary Blocking Flow has a lot of linear subroutines including the computation of the distance metric, the computation of minimum canonical cuts, the construction of the condensed graph, as well as the flow translation and re-routing. As a result, Binary Blocking Flow, while still linear has a much higher constant factor than the Dinitz Max Flow Algorithm, for example. It is also possible that with a faster implementation, in a more memory conscious language such as C++, we would achieve better results, however we will leave that for future work. Our algorithm iterates over the edge set many hundreds of times in a typical graph, leading to a large constant factor.

We also tested the difference in runtime between our two blocking flow algorithms. Despite the theoretical difference, the difference in the runtime of Goldberg-Rao with each algorithm was negligible.

7.1 Runtime Analysis of Single-Variable Fluctuation

We were also curious as to how the runtime scaled with individual variables, namely the edge capacities, the number of edges, and the number of nodes. We ran three more experiments to investigate this, which can be seen in Figures 3, 4, and 5.

In Figure 3, we ran Binary Blocking Flow on a 100-node, 5-regular graph and varied the edge capacities from 1 to 200. The scaling is roughly logarithmic, in conformance to the theoretical bounds.

In Figure 4, we varied the number of edges while keeping the number of nodes and the edge capacities constant. To do this, we generated d-regular graphs, where d ranged from 1 to n, and ran Binary Blocking Flow on the resulting graphs.

In Figure 5, we varied the number of nodes while keeping the number of edges and the edge capacities constant. We generated d-regular graphs on n nodes, where $\frac{d*n}{2}$ equaled 1000, 2000, or 3000. In this case, although the theoretical bounds guarantee that the run-time scales with the number of nodes, we found that in practice, the algorithm runs faster on sparser graphs with more nodes, if the number of edges is held constant. The reason for this is likely because sparser graphs have more connected components which means that parts of the graph are never touched. This might indicate stronger bounds may be proven for this algorithm on sparse graphs.

7.2 Analysis of Empirical Performance by Graph Type

We also ran our algorithm on a few special types of graphs. Our method runs very slowly on complete graphs, as shown in Figure 7. In contrast, the algorithm performs around 200 times faster on lattice graphs, due to their simpler structure. This can be seen in Figure 8. We anticipate this improvement is caused by the constant number of successors from any given node in a lattice. This also has the effect of decreasing the total possible flow in the graph, resulting in faster runtime.

8 CONCLUSION AND FUTURE WORK

Overall, we were able to fully implement the Binary Blocking Flow Algorithm for maximum flow in Python using the `networkX` library. We also discovered that the practical run-time of the algorithm is linear, and on average, the algorithm performs worse than the Dinitz

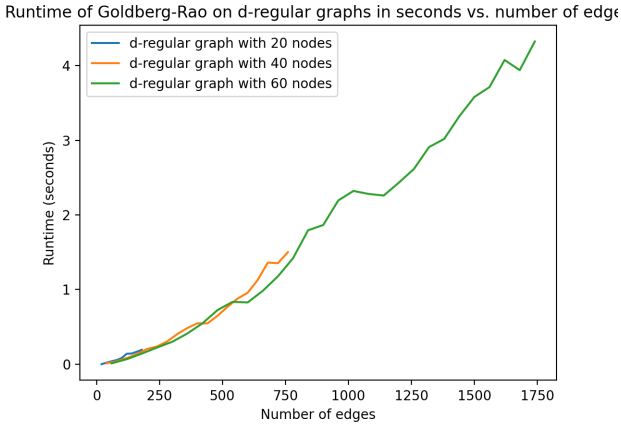

Figure 5: Runtime vs. number of edges, with everything else constant

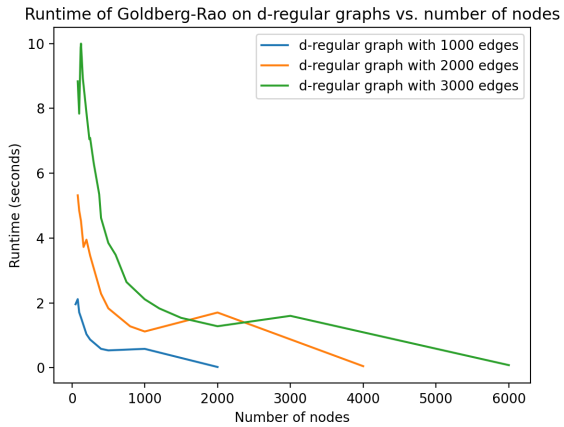

Figure 6: Runtime vs. number of nodes, with everything else constant

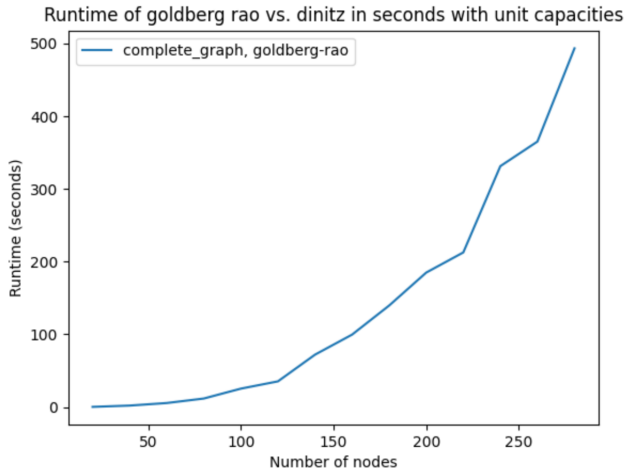

Figure 7: Runtime vs. number of nodes complete graphs

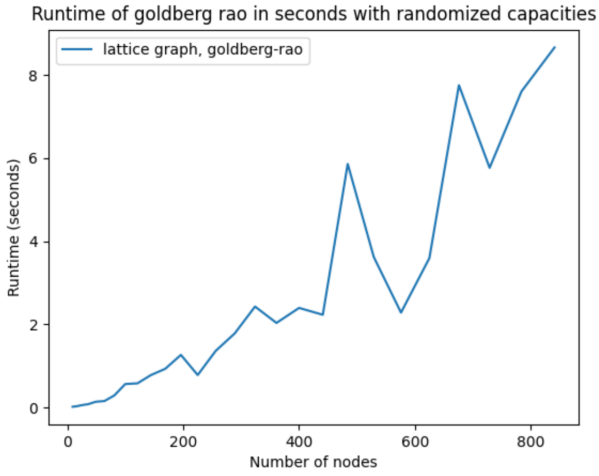

Figure 8: Runtime vs. number of nodes for lattice graphs

algorithm in practice. We attributed this to the fact that for most maximum-flow algorithms, their run-time looks linear, and so often the most simplistic algorithms win out practically. In addition to this, we build testing, visualization, and profiling infrastructure to analyze our algorithm and presented empirical results on a variety of graphs varying a variety of parameters including number of nodes, number of edges, and capacity. We found that the graphs are linear in $O(V + E)$ and logarithmic with the maximum capacity of the graph U, which matches the paper.

In the future, we would like to implement this algorithm and re-run the analysis in C++ as to cut down the linear time contributions due to expensive constant time operations involving Python and `networkX` dictionaries. We would also like to extend our analysis to very large graphs and construct the "worst-case" graph for the Dinitz Max Flow and Binary Blocking Flow algorithms in order to analyze how they perform. Finally we would like to take a survey of practical graphs where flow is likely to be applied. This includes real world networking / mesh graphs, or graphs involving node-balancing of distributed systems. Using these real-world graphs, we can look at how this algorithm has the potential of impacting industry.

REFERENCES

[1] [n.d.]. The Goldberg–Rao Algorithm for the Maximum Flow Problem. http://www.cs.princeton.edu/courses/archive/fall06/cos528/handouts/Goldberg-Rao.pdf

[2] [n.d.]. NetworkX documentation. https://networkx.org/

[3] Yefim Dinitz, Naveen Garg, and Michel X. Goemans. 1999. On the Single-Source Unsplittable Flow Problem. *Combinatorica* 19, 1 (1999), 17–41. https://doi.org/10.1007/s004930050043

[4] Lester Randolph Ford and D. R. Fulkerson. 1963. Flows in Networks. (1963). https://doi.org/10.1515/9781400875184

[5] A.v. Goldberg and S. Rao. 1998. Beyond the flow decomposition barrier. *Proceedings 38th Annual Symposium on Foundations of Computer Science* (Sep 1998). https://doi.org/10.1109/sfcs.1997.646087

[6] A V Goldberg and R E Tarjan. 1986. A new approach to the maximum flow problem. *Proceedings of the eighteenth annual ACM symposium on Theory of computing - STOC 86* (1986). https://doi.org/10.1145/12130.12144

[7] R. Tarjan and C. V. Wyk. 1988. An O(n log log n)-Time Algorithm for Triangulating a Simple Polygon. *SIAM J. Comput.* 17 (1988), 143–178. https://doi.org/10.1137/0217010