Efficiently Computing Maximum Flows in Scale-Free Networks

Thomas Bläsius ✉
Karlsruhe Institute of Technology, Germany

Tobias Friedrich ✉
Hasso Plattner Institute, Universität Potsdam, Germany

Christopher Weyand ✉
Karlsruhe Institute of Technology, Germany

Abstract

We study the maximum-flow/minimum-cut problem on scale-free networks, i.e., graphs whose degree distribution follows a power-law. We propose a simple algorithm that capitalizes on the fact that often only a small fraction of such a network is relevant for the flow. At its core, our algorithm augments Dinitz's algorithm with a balanced bidirectional search. Our experiments on a scale-free random network model indicate sublinear run time. On scale-free real-world networks, we outperform the commonly used highest-label Push-Relabel implementation by up to two orders of magnitude. Compared to Dinitz's original algorithm, our modifications reduce the search space, e.g., by a factor of 275 on an autonomous systems graph.

Beyond these good run times, our algorithm has an additional advantage compared to Push-Relabel. The latter computes a preflow, which makes the extraction of a minimum cut potentially more difficult. This is relevant, for example, for the computation of Gomory-Hu trees. On a social network with 70 000 nodes, our algorithm computes the Gomory-Hu tree in 3 seconds compared to 12 minutes when using Push-Relabel.

2012 ACM Subject Classification Theory of computation → Network flows

Keywords and phrases graphs, flow, network, scale-free

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.21

Related Version *Full Version*: https://arxiv.org/abs/2009.09678 [5]

Supplementary Material *Software (code and data)*: https://github.com/chistopher/scale-free-flow; archived at swh:1:dir:48f8148b8dbabab86dfd6e86ab7dd5dffaeb3cd0

1 Introduction

The maximum flow problem is arguably one of the most fundamental graph problems that regularly appears as a subtask in various applications [2, 29, 32]. The go-to general-purpose algorithm for computing flows in practice is the highest-label Push-Relabel algorithm by Cherkassky and Goldberg [11], which is also part of the boost graph library [30]. Beyond that, the BK-algorithm by Boykov and Kolmogorov [8] or its later iteration [17] should be used for instances appearing in computer vision. Our main goal in this paper is to provide a flow algorithm tailored towards *scale-free* networks. Such networks are characterized by their heavy-tailed degree distribution resembling a power-law, i.e., they are sparse with few vertices of comparatively high degree and many vertices of low degree.

At its core, our algorithm is a variant of Dinitz's algorithm [13], which is an augmenting path algorithm that iteratively increases the flow along collections of shortest paths in the residual network. In each iteration, at least one edge on every shortest path gets saturated, thereby increasing the distance between source and sink in the residual network. To exploit the structure of scale-free networks, we make use of the facts that, firstly, shortest paths tend

© Thomas Bläsius, Tobias Friedrich, and Christopher Weyand;
licensed under Creative Commons License CC-BY 4.0
29th Annual European Symposium on Algorithms (ESA 2021).
Editors: Petra Mutzel, Rasmus Pagh, and Grzegorz Herman; Article No. 21; pp. 21:1–21:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to span only a small fraction of such networks, and secondly, a balanced bidirectional breadth first search is able to find the shortest paths very efficiently [7, 6]. Using a bidirectional search to compute shortest paths in Dinitz's algorithm directly translates this efficiency to the first iteration, as the residual network initially coincides with the flow network. Though the structure of the residual network changes in later iterations, our experiments show that the run time improvements achieved by using a bidirectional search remain high.

Scaling experiments with geometric inhomogeneous random graphs (GIRGs)[1] [9] indicate that the flow computation of our algorithm runs in sublinear time. In comparison, previous algorithms (Push-Relabel, BK, and unidirectional Dinitz) require slightly super-linear time. This is also reflected in the high speedups we achieve on real-world scale-free networks.

With the flow computation itself being so efficient, the total run time for computing the maximum flow for a single source-sink pair in a scale-free network is heavily dominated by loading the graph and building data structures. Thus, our algorithm is particularly relevant when we have to compute multiple flows in the same network. This is, e.g., the case when computing the Gomory-Hu tree [20] of a network. The Gomory-Hu tree is a compact representation of the minimum s-t cuts for all source-sink pairs (s, t). It can be computed with Gusfield's algorithm [21] using $n - 1$ flow computations in a network with n vertices. Using our bidirectional flow algorithm as the subroutine for flow computations in Gusfield's algorithm lets us compute the Gomory-Hu tree of, e.g., the `soc-slashdot` instance with 70 k nodes and 360 k edges in only 2.6 s. In this context, we observe that the Push-Relabel algorithm is also very efficient in computing the flow values by computing a preflow. However, converting this to a flow or extracting a cut from it takes significantly more time.

Contribution. Our findings can be summarized in the following main contributions.

- We provide a simple and efficient flow-algorithm that significantly outperforms previous algorithms on scale-free networks.
- It's efficiency on non-scale-free instances makes it a potential replacement for the Push-Relabel algorithm for general-purpose flow computations.
- Our algorithm is well suited to compute the Gomory-Hu tree of large instances.
- In contrast to previous observations [11, 12], situations exist where computing a flow with the Push-Relabel algorithm is significantly more expensive than computing a preflow.

Related Work. We briefly discuss only the work most related to our result. For a more extensive overview on the topic of flows, we refer to the survey by Goldberg and Tarjan [19].

Our algorithm is based on Dinitz's Algorithm [13], which belongs to the family of *augmenting path algorithms* originating from the Ford-Fulkerson algorithm [16]. Augmenting path algorithms use the *residual network* to represent the remaining capacities and iteratively increase the flow by augmenting it with paths from source to sink in the residual network, until no such path exists. At every point in time, a valid flow is known and at the end of execution, non-reachability in the residual network certifies maximality.

From this perspective, the *Push-Relabel algorithm* [18] does the reverse. At every point in time, the sink is not reachable from the source in the residual network, thereby guaranteeing maximality, while the object maintained throughout the algorithm is a so-called *preflow* and the algorithm stops once the preflow is actually a flow. This is achieved using two operations

[1] GIRGs are a generative network model closely related to hyperbolic random graphs [25]. They resemble real-world networks in regards to important properties such as degree distribution, clustering, and distances.

push and *relabel*; hence the name. Different variants of the Push-Relabel algorithm mainly differ with regards to the order in which operations are applied. A strategy performing well in practice is the highest-label strategy [11]. The extensive empirical study by Ahuja et al. [1] on ten different algorithms shows that the highest-label Push-Relabel algorithm indeed performs the best out of the ten. The only small caveat with these experiments is the fact that they are based on artificial networks that are specifically generated to pose difficult instances. Our experiments show that the structure of the instance matters in the sense that it impacts different algorithms differently; potentially yielding different rankings on different types of instances. The so-called pseudoflow algorithm by Hochbaum [23] was later shown to slightly outperform (low single-digit speedups on most instances) the highest-label Push-Relabel algorithm; again based on artificial instances [10].

Boykov and Kolmogorov [8] gave an algorithm tailored specifically towards instances that appear in computer vision; outperforming Push-Relabel on these instances. It was later refined by Goldberg et al. [17]. Most related to our studies is the work by Halim et al. [22] who developed a distributed flow algorithm for MapReduce on huge social networks.

2 Network Flows and Dinitz's Algorithm

Network Flows. A flow network is a directed graph $G = (V, E)$ with source and sink vertices $s, t \in V$, and a capacity function $c : V \times V \to \mathbb{N}$ with $c(u, v) = 0$ if $(u, v) \notin E$. A *flow* f on G is a function $f : V \times V \to \mathbb{Z}$ satisfying three constrains: (I) capacity $f(u, v) \leq c(u, v)$ (II) asymmetry $f(u, v) = -f(v, u)$ and (III) conservation $\sum_{v \in V} f(u, v) = 0$ for $u \in V \setminus \{s, t\}$. We call an edge $(u, v) \in E$ *saturated* if $f(u, v) = c(u, v)$. Denote the *value* of a flow f as $\sum_{v \in V} f(s, v)$. The maximum flow problem, *max-flow* for short, is the problem of finding a flow of maximum value.

Given a flow f in G, we define a network G_f called the *residual network*. G_f has the same set of nodes and contains the directed edge (u, v) if $f(u, v) < c(u, v)$. The capacity c' of edges in G_f is given by the residual capacity in the original network, i.e., $c'(u, v) = c(u, v) - f(u, v)$. An s-t path in G_f is called an *augmenting path*.

Dinitz's Algorithm. Let $d_s(v)$ be the distance from s to vertex v in G_f. We define a subgraph of G_f called the *layered network* by restricting the edge set to edges (u, v) of G_f for which $d_s(u) + 1 = d_s(v)$, i.e., edges that increase the distance to the source. We call a flow *blocking* if every s-t path contains at least one edge that is saturated by this flow, i.e., there is no augmenting path.

Dinitz's algorithm (see Algorithm 1) groups augmentations into rounds. It augments a set of edges that constitutes a blocking flow of the layered network in each round. One can find such a set of edges by iteratively augmenting s-t paths in the layered network until source and sink become disconnected. After augmenting a blocking flow, the distance between the terminals in the residual network strictly increases.

Algorithm 1 Dinitz's Algorithm.

```
1 while s-t path in residual network do
2     build layered network
3     while s-t path in layered network do
4         augment flow with s-t path
```

Asymptotic Running Time. To better understand how our modifications impact the run time, we briefly sketch how Dinitz running time of $O(n^2m)$ is obtained. Since $d_s(t)$ increases each round, the number of rounds is bounded by $n-1$. Each round consists of two stages: building the layered network and augmenting a blocking flow. The layered network can be constructed in $O(m)$ using a breadth-first search (BFS). Finding the blocking flow is done with a repeated graph traversal, usually using a depth-first search (DFS). The number of found paths is bounded by m, because each found path saturates at least one edge, removing it from the layered network. A single DFS can be done in amortized $O(n)$ time as follows. Edges that are not part of an s-t path in the layered network do not need to be looked at more than once during one round. This is achieved by remembering for each node which edges of the layered network were already found to have no remaining path to the sink. Each subsequent DFS will start where the last one left off. Thus, per round, the depth-first searches have a combined search space of $O(m)$, while each individual search additionally visits the nodes on one s-t path which is $O(n)$.

Practical Performance. In our experiments $d_s(t)$ remains mostly below 10, implying that the number of rounds is significantly lower than $n-1$. Also, the number of found augmenting paths during one rounds is far below m. In unweighted networks, for example, a DFS saturates all edges of the found path resulting in a bound of $O(m)$ to find a blocking flow. Dinitz's algorithm has a tight upper bound of $O(n^{2/3}m)$ in unweighted networks [14, 24].

3 Improving Dinitz on Scale-Free Networks

We adapt a common Dinitz implementation[2] to exploit the specific structure of scale-free networks. We achieve a significant speedup by using the fact that a flow and cut respectively often depend only on a small fraction of the network. The following three modifications each tackle a performance bottleneck.

Bidirectional Search. Recently, sublinear running time was shown for balanced bidirectional search in a scale-free network model [6, 7]. We use a bidirectional breadth-first-search to compute the distances that define the layered network during each round of Dinitz's algorithm. A forward search is performed from the source and a backward search from the sink, each time advancing the search that incurs the lower cost to advance one layer. A shortest s-t path is found when a vertex is discovered that was already seen from the other direction. Note that, for our purpose, the bidirectional search has to finish the current layer when such a vertex is discovered, because all shortest paths must be found. Figure 1 visualizes the difference in explored vertices between a normal and a bidirectional BFS. The augmentations with DFS are restricted to the visited part of the layered network, meaning the search space of the BFS plus the next layer.

The distance labeling obtained by the bidirectional BFS requires a change to the DFS. The purpose of the layered network is to contain all edges on shortest s-t paths. The DFS identifies edges (u, v) of the layered network by checking if they increase the distance from the source, i.e., $d_s(u) + 1 = d_s(v)$. However, we no longer obtain the distances from the source for all relevant vertices. For vertices processed by the backward search, distances to the sink $d_t(v)$ are known instead. To resolve the problem, we allow edges that either increase distance from the source or decrease distance to the sink, i.e., $d_s(u) + 1 = d_s(v)$

[2] `https://cp-algorithms.com/graph/dinic.html`

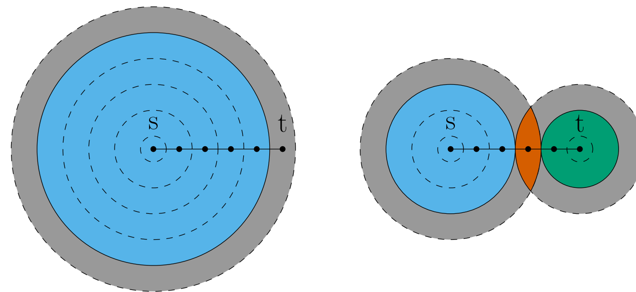


Figure 1 Search space of a breadth-first search from a source s to a sink t unidirectional (left) and bidirectional (right). The blue area represents the vertices that are explored, i.e., whose outgoing edges were scanned, by the forward search and the green area the backward search. In the gray area are vertices that are seen during exploration of the last layer, but not yet explored. Vertices in the intersection of the upcoming layers of the backward and forward search are marked orange.

or $d_t(u) - 1 = d_t(v)$. This deviates from the definition of the layered network. But since edges on shortest s-t paths must both, increase the distance from the source and decrease the distance to the sink, we do not miss any relevant edges.

Time Stamps. The bidirectional search reduces the search space of the breadth-first search and depth-first search substantially, potentially to sublinear. The initialization, however, still requires linear time. It includes distances from the source and to the sink and one progress counter per node for the augmentations. To avoid the linear initializations, we introduce time stamps to indicate if a vertex was seen during the current round. The initialization of distances and counters is done lazily as vertices are discovered during the BFS.

Skip Next Forward Layer. The DFS proceeds along edges outgoing from the last forward search layer independent from the target vertex being seen only by the forward search (gray in Figure 1) or also by the backward search (orange in Figure 1). However, the former type of vertex cannot be part of a shortest s-t path. By saving the number of explored layers of the forward search we can avoid the exploration of such vertices, thus limiting the DFS to vertices colored blue, green, or orange in Figure 1. With this optimization, the combined search space during augmentation (lines 3,4 in Algorithm 1) is almost limited to the search space of the BFS. The only additional edges that are visited originate from the intersection of the forward and backward search.

4 Experimental Evaluation

In this section, we investigate the performance of our algorithm *DinitzOPT*. First, we compare it to established approaches on real-world networks in Section 4.1. We additionally examine the scaling behavior and how the comparison is affected by problem size, i.e., is there an asymptotic improvement over other algorithms? Then, Section 4.2 evaluates to which extent the different optimizations contribute to better run times and search space. In Section 4.3 we analyze the algorithms in a specific application (Gomory-Hu trees) and compare their usability beyond the speed of the actual flow computation. To this end, we test three different approaches to obtain a cut with the Push-Relabel algorithm.

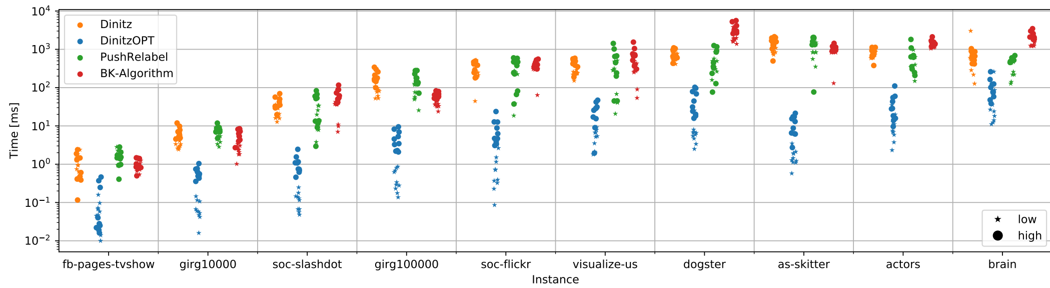


Figure 2 Runtime comparison of flow computations. The 20 computed flows per instance are divided into *low* and *high* terminal pairs. For *low*, the terminal degree is between 0.75 and 1.25 times the average degree. For *high*, it is between 10 and 100 times the average degree. Pairs are chosen uniformly at random from all vertices with the respective degree.

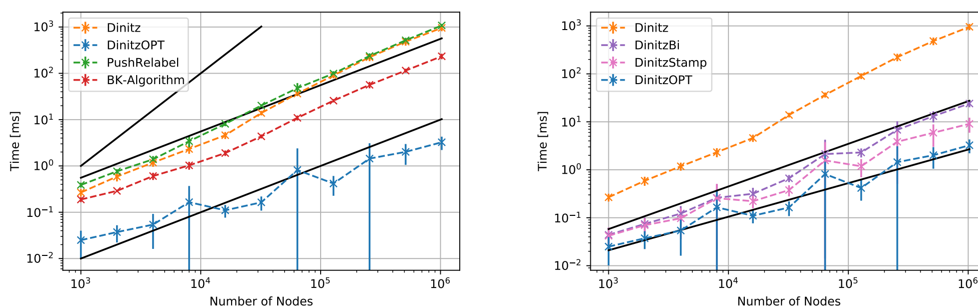
4.1 Runtime Comparison

In this section we compare our new approach to three existing algorithms: Dinitz [13], Push-Relabel [18], and the Boykov-Kolmogorov (BK) algorithm [8]. We modified their respective implementations to support our experiments. This also includes some minor performance-relevant changes listed in the long version of the paper along with further details on the datasets [5]. The experiments include two synthetic and eight real-world networks. All networks are undirected and all but `visualize-us` and `actors` are unweighted. We restrict our experiments in this section to the flow computation only excluding, e.g., loading times and resetting flow values between runs. For Push-Relabel we only measure the computation of the *preflow*, which is sufficient to determine the value of the flow/cut. Figure 2 shows the resulting run times. For this plot, the terminals were chosen uniformly at random from the set of vertices with degree close to the average (*low*) or considerably higher degree (*high*).

One can see that Dinitz and Push-Relabel display comparable times while BK is slightly slower on most large instances. DinitzOPT consistently outperforms the other algorithms by one to three orders of magnitude. The variance is also higher for DinitzOPT with *low* pairs approximately one order of magnitude faster on average than *high* pairs. This is best seen in the `girg100000` instance and suggests that DinitzOPT is able to better exploit easy problem instances. For all other algorithms the effect of the terminal degree on the run time is barely noticeable. Another observation is that all algorithms display drastically lower run times than their respective worst-case bounds would suggest.

The times in our experiments are close to what one might expect from linear algorithms. For example, Dinitz computes a flow on the `as-skitter` instance in one second. Considering the tight $O(mn^{2/3})$ bound in unweighted networks and assuming the throughput per second to be around 10^8 – which is a generous guess for graph algorithms – would result in an estimate of 30 minutes per flow. In contrast to our results, earlier studies found Dinitz to be slower than Push-Relabel and both algorithms clearly super-linear on a series of synthetic instances [1]. However, these synthetic instances exhibit specifically crafted hard structures that are placed between designated source and sink vertices. These instances thus present substantially more challenging flow problems.

Effect of the Terminal Degree. In the following, we discuss the effect of terminal degree and structure of the cut on the run time of Dinitz and DinitzOPT. Note that the terminal degree is an upper bound on the size of the cut in unweighted networks. Moreover, the



(a) Runtime scaling of flow algorithms.

(b) Scaling of Dinitz variants.

Figure 3 (a) The average time per flow over multiple GIRGs and terminal pairs. (b) This plot differs from Figure 3a only in the set of displayed algorithms.

terminal degree in our experiments is based on the average degree, which is assumed to be constant in many real-world networks [3]. Thus, the $O(mC)$ bound for augmenting path based algorithms, with C being the size of the cut, implies not only a linear bound for the eight unweighted networks in our experiments, but would also explain faster *low* pairs. Surprisingly, DinitzOPT exploits low terminal degrees much more than Dinitz. Another explanation for faster *low* pairs is that many cuts are close around one terminal, which is consistent with previous observations about cuts in scale-free networks [27, 31]. Moreover, Dinitz tends to perform well when the source side of the cut is small [28]. Although this does not fully explain why DinitzOPT is more sensitive to the terminal degree, we observe in Section 4.3 that Dinitz slows down massively when the source degree is high, even with low sink degree. Since DinitzOPT always advances the side with smaller volume during bidirectional search it does not matter which terminal has the higher degree.

Scaling. We perform additional experiments to analyze the scaling behavior of the algorithms. Since real networks are scarce and fixed in size, we generate synthetic networks to gradually increase the size while keeping the relevant structural properties fixed. Geometric Inhomogeneous Random Graphs (GIRGs) [9], a generalization of Hyperbolic Random Graphs [25], are a scale-free generative network model that captures many properties of real-world networks. The efficient generator [4] allows us to benchmark our algorithms on differently sized networks with similar structure. Figure 3a and Figure 3b show the results.

We measure the run time over a series of GIRGs with the number of nodes growing exponentially from 1000 to 1 024 000 with 10 iterations each. In each iteration, we sample a new random graph with average degree 10, power-law exponent 2.8, dimension 1, and temperature 0. The run time for each algorithm is then averaged over 10 uniform random pairs of vertices with degree between 10 and 20. Standard deviation is shown as error bars. The lower half of the symmetric error bars seems longer due to the log-axis. We add a quadratic and two linear functions in Figure 3a. Figure 3b shows the functions $n^{0.88}$ and $n^{0.7}$ representing the theoretical upper bound and previously observed typical run times, respectively, for the bidirectional search on hyperbolic random graphs with the chosen power-law exponent [6].

Dinitz, Push-Relabel and BK show a near-linear running time. Compared to the linear functions in Figure 3a, Dinitz and Push-Relabel seem to scale slightly worse than linear, while DinitzOPT scales better than linear. In a construction with super-sink and super-source, a similar scaling was observed for Push-Relabel on the Yahoo Instant Messenger graph [26].

Table 1 Total run times and search space of visited edges for the five intermediate versions of our Dinitz implementation during the computation of 1000 flows in `as-skitter`. Terminals are chosen like *low* pairs in Figure 2. The first seven columns show times in seconds accumulated over all flow computations. BUILD is the construction of the residual network that is reused for all flow computations, RESET means clearing flow on edges between computations, INIT includes initialization of distances and counters per round, BFS and DFS refer to the respective subroutines, FLOW is the summed time during flow computations (sum of BFS, DFS, INIT), and TOTAL is the run time of the whole application including reading the graph from file. The last three columns contain the search space relative to the number of edges in the graph in percent. Search space columns for BFS and DFS are per round, while the FLOW column lists the search space per flow, e.g., Dinitz visits on average 65.66% of all edges per BFS and every edge is visited about 5.58 times on average in one flow computation.

			MaxFlow					Search Space [%]		
	BUILD	RESET	INIT	BFS	DFS	FLOW	TOTAL	BFS	DFS	FLOW
Dinitz	0.50	56.79	14.87	405.46	426.80	847.13	904.85	65.66	63.64	558.04
DinitzBi	0.55	58.15	21.02	2.78	8.94	32.73	91.82	0.26	1.87	8.38
DinitzReset	0.50	|	20.73	2.47	8.01	31.20	32.06	0.26	1.87	8.38
DinitzStamp	0.55	|	|	2.51	10.30	12.81	13.72	0.26	1.87	8.38
DinitzOPT	0.55	|	|	2.40	1.06	3.46	4.22	0.26	0.20	2.03

4.2 Optimizations in Detail

Instead of considering all combinations of optimizations, we individually add them in a specific order, such that the next change always tackles a performance bottleneck. In fact, additional benchmarks reveal that the current optimization speeds up the computation more than enabling all other remaining changes together. The four incrementally more optimized versions of the algorithm are: DinitzBi, DinitzReset, DinitzStamp, and DinitzOPT.

Experimental Setup. The experiments and benchmarks in this section consider 1000 uniform random terminal pairs close to the average degree on the `as-skitter` instance. The average distance between source and sink in the initial network is 4.2. The average number of rounds until a maximum flow is found is 4.8, where the last round runs only the BFS to verify that no augmenting path exists. Only counting rounds before the last round, 2.9 units of flow are found on average per round. Out of the 1000 cuts, 882 have value equal to the degree of the smaller terminal. Table 1 shows profiler results and search space for Dinitz and the optimized versions of the algorithm. Figure 4 compares the search space with and without bidirectional search.

Bidirectional Search. Dinitz takes 15 minutes to compute the 1000 flows and the search space per flow is more than five times the number of edges on average. Almost all of that time is spent in BFS or DFS. The bidirectional Dinitz reduces the flow-time from 14 minutes to 30 seconds, an improvement by a factor of 25.

The search space is reduced by factors of 252 for BFS, 34 for DFS, and 67 per flow. It is interesting to note, that the search space of BFS during the last round of each flow changes even more. In this round the BFS will find no s-t path. The bidirectional search visits 39 edges on average, while the normal breadth-fist-search visits 44% of the graph. This not only emphasizes that the cuts are close around one terminal, but also shows that the bidirectional search heavily exploits this structure.

The run time does not fully reflect this drastic reduction in search space, because DFS and BFS no longer dominate the flow computation. The initialization time per round increased by 50%, which can be explained by the additional distance label per node to store the

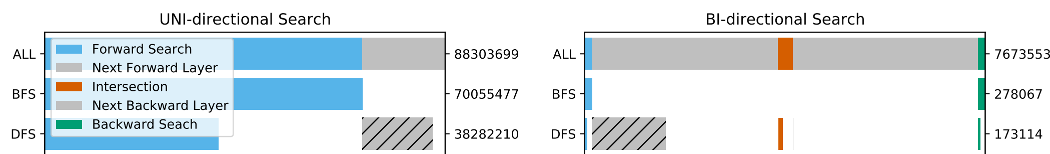


Figure 4 Average number of edges visited per flow computation for the terminal pairs used in Table 1, partitioned as in Figure 1. *Forward/Backward Search* represent the edges explored by the respective search. *Next Forward/Backward Layer* denote the edges that would be explored in the next step of the BFS. Edges in the *Intersection* originate from vertices in both upcoming BFS-layers. The BFS and DFS bars show the edges that are actually visited by the algorithm. The shaded area indicates the edges skipped by our last optimization (from DinitzStamp to DinitzOPT in Table 1) and is excluded in the sum on the right.

distance to the sink (now 3 ints instead of 2). Although the initialization is a simple linear operation in the number of nodes, it takes twice as long as BFS and DFS combined. The real bottleneck, however, is to reset the flow values between computations. RESET takes almost a full minute which is twice as long as computing the flows.

Reset flow between computations. Between flow computations, the residual capacity of all edges has to be reset before another flow can be found. After changing the BFS to a bidirectional search, resetting the flow on all edges between computations dominates the run time. To reduce the time of our benchmarks, and to make the code more efficient in situations where multiple flows are computed in the same network, we address this bottleneck. Instead of explicitly resetting flow values for all edges, we remember the edges that contain flow and reset only those. This change is not mentioned in Section 3 because it does not speed up a single flow computation.

This change reduces the time for RESET to the point that it is no longer detected by the profiler, while other operations are not affected. The total time to compute all 1000 flows is thus three times lower with the flow computation making up for almost all spent time. The slowest part of the flow computation itself is still the initialization with 21 of the 31 seconds.

Time Stamps. The distance labels and counters per node are initialized each round. Using time stamps eliminates the need for initialization while adding a small overhead to DFS. The flow computation gets 2.4 times faster with 13 seconds instead of 31. After introducing the time stamps, the DFS is the new bottleneck and makes up for about 80% of flow time.

Skip Next Forward Layer. This change prevents the DFS from visiting vertices beyond the last layer of the forward search that are not also seen by the backwards search. In Figure 4 the skipped part is shaded. This optimization reduces the average search space for DFS during one round from almost 2% of all edges to just 0.2%. The improvement in search space is reflected by the profiler results. DFS is sped up from 10 seconds to just one second, which is faster than the BFS. The resulting time to compute all 1000 flows is 3.46 seconds, which is only 7 times slower than building the adjacency list in the beginning. In total, the time to compute the flows with the optimized Dinitz is 245 times faster than the unmodified Dinitz.

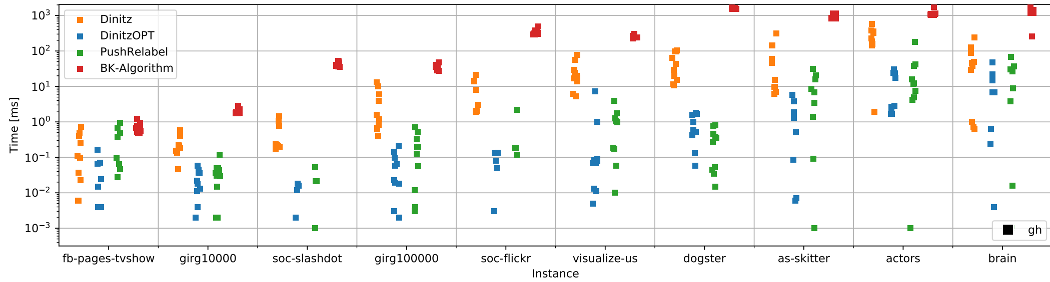


Figure 5 Runtime comparison of flow computations. The 10 terminal pairs per instance are uniformly chosen out of the $n-1$ cuts required by Gusfield's algorithm.

4.3 Gomory-Hu Trees

In the last sections we observed that heterogeneous network structure yields easy flow problems that can be solved significantly faster than the construction of the adjacency list. This performance becomes important in applications that require multiple flows to be found in the same network. Gomory-Hu trees [20] fit this setting and have applications in graph clustering [15]. A Gomory-Hu tree (GH-tree) of a network is a weighted tree on the same set of vertices that preserves minimum cuts, i.e., each minimum cut between any two vertices s and t in the tree is also a minimum s-t cut in the original network. Thus, they compactly represent s-t cuts for all vertex pairs of a graph. For the construction of a GH-tree, we use Gusfield's algorithm [21] that requires $n-1$ cut-oracle calls in the original graph.

In this section we evaluate the performance of max-flow algorithms for the construction of Gomory-Hu trees in heterogeneous networks. We will see that the terminal pairs required for Gusfield's algorithm yield easier flow problems than uniform random pairs. DinitzOPT is able to make use of this easy structure to achieve surprisingly low run times, so is Push-Relabel when only considering the computation of the flow value. However, we find that the need to extract the source side of the cut hinders Push-Relabel to benefit from this performance.

Flow Computation on Gusfield Pairs. Figure 5 shows the same networks and algorithms as in Figure 2 but with terminal pairs sampled out of the $n-1$ flow computations needed by Gusfield's algorithm. The run times for all algorithms except the BK-Algorithm have high variance and are spread over up to four orders of magnitude for the larger instances. Although results for different terminal pairs vary greatly, BK seems to be the slowest algorithm followed by Dinitz. DinitzOPT and PR have comparable but significantly lower run times than the other algorithms. For example, 6 out of the 10 *gh* pairs measured for the `soc-slashdot` instance are solved by DinitzOPT and Push-Relabel faster than one microsecond which is the precision of our measurements. This suggests, that these algorithms are more sensitive to the varying difficulty of the flow computations for *gh* pairs. Our speedup over the Push-Relabel algorithm on *gh* pairs is not as pronounced as for the random pairs in Section 4.1. On the `dogster` instance PR is even faster than DinitzOPT.

To further investigate why *gh* pairs are this easy to solve, we analyze a complete run of all pairs needed by Gusfield's algorithm on the `soc-slashdot` instance. In Gusfield's algorithm each vertex is the source once, thus the average degree of the source is the average degree of the graph (10.24). In contrast, the average degree of the sink is ca. 1500, which hinders the benefit of bidirectional search. Uni-directional Dinitz slows down by a factor of 15 when computing the flows with switched terminals. The average distance between two vertices

in the original network is 4.16, but interestingly here the average distance from source to sink is 1.78. Out of the 70 k flow computations, 56 k are trivial cuts around one terminal. Computing a flow for a single s-t pair takes 2.76 rounds on average with the last round only to confirm that the flow is optimal.

DinitzOPT and Push-Relabel are both extremely fast on *gh* pairs. DinitzOPT takes 2.5 seconds to compute all $n = 70\,\text{k}$ required flows, while PR needs 5 seconds. To obtain the 5 seconds for PR we exclusively measured the preflow computation, but PR is not limited by the time to compute the preflow. Actually, the entire computation of the Gomory-Hu tree on the `soc-slashdot` instance takes 12 minutes with Push-Relabel and 2.6 seconds with DinitzOPT. Instead of being caused by the Gusfield logic – which actually makes up less than 3% of the run time when using DinitzOPT as oracle – the bottleneck when using PR as a cut oracle is not the flow computation, but initialization and extracting the cut. The drastic difference in run time is in part due to the optimizations we added to DinitzOPT to reduce time between flow computations, while the Push-Relabel implementation recreates the auxiliary data structures, except the adjacency list, before each flow. However, in the following we will see that a large amount of Push-Relabels run time is actually necessary to extract the cuts for Gusfield's algorithm.

Computing Cuts with Push-Relabel. In Gusfield's algorithm we have to iterate over all vertices in the source-side of the cut. For Dinitz algorithm we can obtain a cut by doing a BFS from the source. However, the PR algorithm only computes a preflow. We outline the following three approaches to extract the cut and show that each has major drawbacks.

Convert. Compute a preflow, convert it into a flow, then run BFS from the source.
T-Side. Compute a preflow, run BFS backwards from the sink, then take complement.
Swap. Compute a preflow from sink to source, then run BFS backwards from the source.

The most straightforward way to get a cut from a preflow is to convert it into a flow. Then, as for Dinitz, one partition of a min-cut can be identified by reachability from the source in the residual network. In previous works, the conversion from preflow to flow makes up only a small fraction of the running time [11, 12]. For Gusfield pairs, however, Figure 6 shows that the conversion highly dominates the computation of the preflow. Only about 5 seconds of the 12 minutes of the complete run are spent in preflow computation.

To circumvent the conversion, we use the observation that one can obtain a cut directly from the preflow by finding all sink-reaching vertices in the residual network. Since Gusfield requires the source-side of the cut, the complement of the found set of vertices can be used. Unfortunately, doing the backward search from the sink is even more expensive than the conversion. An explanation for this is the large sink side of the cut. Using this *T-side* approach to identify the cut for DinitzOPT takes 4.5 minutes which is a factor 100 slower than identifying the cut via the source-side for DinitzOPT.

Making use of the fact that the source side of the cut is much smaller than the sink side, the drawbacks of the previous approach can be avoided in undirected networks by computing the preflow from sink to source. A cut can then be extracted by determining the vertices that can reach the original source in the residual network. The drawback of this method is that the preflow computation slows down massively from 5s to 47 minutes.

In conclusion, the *convert* approach is the fastest with just above 12 minutes followed by *T-side* with 18 minutes and *swap* with almost an hour. However, all three methods perform significantly worse than DinitzOPT, not because PR flow computations are slow, but both methods to avoid the four minutes run time of preflow-conversion imply even worse performance cost; either due to a breadth-first search that has to traverse almost the whole graph (T-side) or due to significantly slower preflow computations (Swap).

Figure 6 Distribution of spent time during Gusfield's algorithm on the `soc-slashdot` instance with three approaches to use the Push-Relabel algorithm as a min-cut oracle. We split the measurements into initialization, preflow, conversion, and cut identification. The time overhead for measurement, logging, and the logic of Gusfield's algorithm is included in the numbers on the right but excluded in the bars.

5 Conclusion

We presented a modified version of Dinitz's algorithm with greatly improved run time and search space on real-world and generated scale-free networks. The scaling behavior appears to be sublinear, which matches previous theoretical and empirical observations about the running time of balanced bidirectional search in scale-free random networks. While these theoretical bounds apply during the first round of our algorithm, it is still unknown whether the analysis can be extended to account for the changes in the residual network. Our experiments, however, indicate that the search space remains small in subsequent rounds.

We observe that the low diameter and heterogeneous degree distribution lead to small and unbalanced cuts that our algorithm finds very efficiently. The flow computations required to compute a Gomory-Hu tree are even easier, making usually insignificant parts of the tested algorithms be a bottleneck. For example, the preflow conversion leads to Push-Relabel being greatly outperformed by our algorithm in this setting.

References

1 Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, 97(3):509–542, 1997. `doi:10.1016/S0377-2217(96)00269-X`.

2 Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: Theory, algorithms and applications*. Prentice-Hall, Inc., 1993.

3 Albert-László Barabási. *Network science*. Cambridge university press, 2016.

4 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs. In *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. `doi:10.4230/LIPIcs.ESA.2019.21`.

5 Thomas Bläsius, Tobias Friedrich, and Christopher Weyand. Efficiently computing maximum flows in scale-free networks. *CoRR*, abs/2009.09678, 2020. `arXiv:2009.09678`.

6 Thomas Bläsius, Cedric Freiberger, Tobias Friedrich, Maximilian Katzmann, Felix Montenegro-Retana, and Marianne Thieffry. Efficient Shortest Paths in Scale-Free Networks with Underlying Hyperbolic Geometry. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018. `doi:10.4230/LIPIcs.ICALP.2018.20`.

7 Michele Borassi and Emanuele Natale. KADABRA is an ADaptive Algorithm for Betweenness via Random Approximation. In *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.ESA.2016.20`.

8 Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max- flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004. `doi:10.1109/TPAMI.2004.60`.

9 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theoretical Computer Science*, 760:35–54, 2019. `doi:10.1016/j.tcs.2018.08.014`.

10 Bala G. Chandran and Dorit S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358–376, 2009.

11 B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. `doi:10.1007/pl00009180`.

12 U. Derigs and W. Meier. Implementing Goldberg's max-flow-algorithm — A computational investigation. *Zeitschrift für Operations Research*, 33(6):383–403, 1989. `doi:10.1007/BF01415937`.

13 Yefim Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

14 Shimon Even and R. Endre Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, 1975. `doi:10.1137/0204043`.

15 Gary William Flake, Robert E. Tarjan, and Kostas Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2004. `doi:10.1080/15427951.2004.10129093`.

16 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. `doi:10.4153/CJM-1956-045-5`.

17 Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum Flows by Incremental Breadth-First Search. In *19th Annual European Symposium on Algorithms (ESA 2011)*, Lecture Notes in Computer Science, pages 457–468. Springer, 2011. `doi:10.1007/978-3-642-23719-5_39`.

18 Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988. `doi:10.1145/48014.61051`.

19 Andrew V. Goldberg and Robert E. Tarjan. Efficient maximum flow algorithms. *Commun. ACM*, 57(8):82–89, 2014. `doi:10.1145/2628036`.

20 R. E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

21 Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990. `doi:10.1137/0219009`.

22 Felix Halim, Roland H.C. Yap, and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *2011 31st International Conference on Distributed Computing Systems*, pages 192–202, 2011. ISSN: 1063-6927. `doi:10.1109/ICDCS.2011.62`.

23 Dorit S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, August 2008. `doi:10.1287/opre.1080.0524`.

24 Alexander V. Karzanov. On finding a maximum flow in a network with special structure and some applications. *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973.

25 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), 2010. `doi:10.1103/physreve.82.036106`.

26 Kevin Lang. Finding good nearly balanced cuts in power law graphs. Technical Report YRL-2004-036, Yahoo! Research Labs, 2004.

27 Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009. doi:10.1080/15427951.2009.10129177.

28 Lorenzo Orecchia and Zeyuan Allen Zhu. Flow-based algorithms for local graph clustering. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2014. doi:10.1137/1.9781611973402.94.

29 Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007. doi:10.1016/j.cosrev.2007.05.001.

30 Boris Schäling. *The boost C++ libraries*. Boris Schäling, 2011. URL: https://theboostcpplibraries.com/.

31 S.-W. Son, H. Jeong, and J. D. Noh. Random field ising model and community structure in complex networks. *The European Physical Journal B*, 50(3):431–437, 2006. doi:10.1140/epjb/e2006-00155-4.

32 Tanmay Verma and Dhruv Batra. MaxFlow revisited: An empirical comparison of maxflow algorithms for dense vision problems. In *Procedings of the British Machine Vision Conference 2012*. British Machine Vision Association, 2012. doi:10.5244/c.26.61.