Experimental Evaluation of Parametric Max-Flow Algorithms

Maxim Babenko[1⋆], Jonathan Derryberry[2], Andrew Goldberg[3], Robert Tarjan[2⋆⋆], and Yunhong Zhou[2]

[1] Moscow State University, Moscow, Russia
[2] HP Labs, 1501 Page Mill Rd, Palo Alto, CA 94304
[3] Microsoft Research – SVC, 1065 La Avenida, Mountain View, CA 94043

Abstract. The *parametric maximum flow problem* is an extension of the classical maximum flow problem in which the capacities of certain arcs are not fixed but are functions of a single parameter. Gallo *et al.* [6] showed that certain versions of the push-relabel algorithm for ordinary maximum flow can be extended to the parametric problem while only increasing the worst-case time bound by a constant factor. Recently Zhang *et al.* [14, 13] proposed a novel, simple *balancing algorithm* for the parametric problem on bipartite networks. They claimed good performance for their algorithm on networks arising from a real-world application. We describe the results of an experimental study comparing the performance of the balancing algorithm, the GGT algorithm, and a simplified version of the GGT algorithm, on networks related to those of the application of Zhang *et al.* as well as networks designed to be hard for the balancing algorithm. Our implementation of the balancing algorithm beats both versions of the GGT algorithm on networks related to the application, thus supporting the observations of Zhang *et al.* On the other hand, the GGT algorithm is more robust; it beats the balancing algorithm on some natural networks, and by asymptotically increasing amount on networks designed to be hard for the balancing algorithm.

1 Introduction

The parametric maximum flow problem is a generalization of the ordinary maximum flow problem in which the capacities of arcs out of the source (into the sink) depend on a single parameter and are monotonically increasing (decreasing) functions of the parameter. Applications of parametric maximum flow beyond those of ordinary maximum flow include product selection [3, 12], database record segmentation [5], repair kit selection [11], and flow sharing [6].

The current best time bounds for the ordinary maximum flow problem on a network with n vertices, m arcs, and integral arc capacities bounded by U are $O(nm \log_{m/(n \log n)} n)$ [10] and $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ [7]. The former algorithm is based on the push-relabel method [8]. Gallo *et al.* [6] show

⋆ Part of this work was done while the author was visiting Microsoft Research – SVC.
⋆⋆ Also Department of Computer Science, Princeton University.

how to modify certain versions of the push-relabel method using amortization and graph contraction to obtain an algorithm that solves the parametric maximum flow problem yet has the same asymptotic complexity as the original algorithm. Their idea applies to the algorithm of [10], giving an $O(nm \log_{m/(n \log n)} n)$ bound for the parametric flow problem. Tarjan *et al.* [13] give a divide and conquer approach that uses an ordinary maximum flow algorithm as a black box to achieve a running time that is a factor $\min\{n, \log(nU)\}$ worse than that of the black box algorithm. In combination with [7], this gives an $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U \min\{n, \log(nU)\})$ bound for the parametric problem. In practice, certain implementations of the push-relabel method (*e.g.* [4]) have better overall performance than those of the algorithm of [7], which makes the GGT algorithm a promising choice for the parametric flow problem.

Zhang *et al.* [14] recently introduced an algorithm for the parametric problem based on a new technique called *star balancing* (see Section 3). This algorithm solves the special case of the parametric problem in which the network is bipartite, source arcs have capacity λ, where λ is the parameter, sink arcs have constant capacity, and all other arcs have infinite capacity. This is an important special case, which includes all of the applications mentioned above except for flow sharing. The star balancing algorithm with a small enhancement suggested by Tarjan *et al.* [13] runs in time $O(mn^2 \log(nU))$ [13]. One can show that this analysis is tight for a family of long path examples (See Section 4.2). Although this bound is significantly worse than the best bounds currently known, the worst-case bound is overly pessimistic for many real-world instances. In particular, the star balancing algorithm performs well on several real-world instances of the product selection problem [14]. This motivates experimental comparison between this algorithm and the GGT algorithm.

Few experimental studies of the parametric flow problem have been published in the open literature [2, 13, 14]. Our codes are the same or better than the corresponding ones in these studies. The only other implementation we are aware of is based on an algorithm described in [9]. However, this implementation became available to us too late for comparison in the current paper.

Our comparison between the GGT and star balancing algorithms involves several steps. First, one needs to develop efficient implementations of the algorithms, which is non-trivial. Then one needs to find interesting real-world and synthetic instances that show strengths and weaknesses of the algorithms. We restricted the experiments described here to bipartite problems of the kind to which the star balancing algorithm applies. Experiments with the GGT algorithm on some other graph types can be found in [2].

The rest of this paper is organized as follows. Section 1.1 reviews the ordinary and parametric maximum flow problems and describes the notations we use. Section 2 describes the GGT algorithm and an efficient implementation of it. Section 3 describes the star balancing algorithm and its implementation. Section 4 is devoted to our experiments. Finally, Section 5 contains concluding remarks, including possible future research directions.

1.1 Background and Notation

For the ordinary maximum flow problem, the input is a directed, capacitated network $\mathcal{N} = (V, A, s, t, c : A \to \mathbb{Z}^+)$, where V is a set of vertices of size n, A is a set of directed arcs (u, v) of size m, s and t are two special vertices (the source and the sink), and c is a capacity function. We assume that the capacities are integers in the range $[1, U]$. A *flow* in a network is a function $f : A \to \mathbb{R}$ that satisfies the capacity constraints $0 \leq f(a) \leq c(a) \quad \forall a \in A$ and the flow conservation constraints $\sum_{(u,v)\in A} f(u, v) = \sum_{(v,w)\in A} f(v, w)$ for all $v \in V - \{s, t\}$. The output for an ordinary maximum flow problem is a flow f such that $\sum_{(s,v)\in A} f(s, v)$ is maximized.

For the parametric maximum flow problem, the input is a directed, capacitated network $\mathcal{N} = (V, A, s, t, c : A \times \mathbb{R} \to \mathbb{Z}^+)$, where the extra input to the capacity function is a parameter λ, upon which the capacities of some arcs may depend. The capacities of arcs out of the source are monotonically increasing in λ, while those of arcs into the sink are monotonically decreasing in λ.[4] All other arcs must have constant capacities (*i.e.*, the capacities cannot depend on λ). The set of minimum cuts for all values of λ has a nested structure: as the value of λ increases, the source side of the cut grows. As a result, there are $n - 1$ or fewer *critical* values of λ, called *breakpoints*, at which the minimum cut changes. The output for a parametric problem is the sequence of breakpoints along with the corresponding nested cuts, and possibly corresponding maximum flows (or information about them).

2 GGT Algorithm

2.1 Push-Relabel Algorithm

The GGT algorithm is based on the push-relabel algorithm [8] for the maximum flow problem. The push-relabel algorithm uses two basic operations, *push* and *relabel*, and maintains a flow and integral *distance labels* on vertices. The important properties of the algorithm are that the distance labels are monotonically increasing, the value of each distance label changes by $O(n)$, and the work of the algorithm is charged to the distance label increases. We assume that the reader is familiar with the push-relabel algorithm as discussed in [8] or [6].

2.2 GGT Algorithm

In this section we describe two algorithms for the parametric flow problem, a simple algorithm based on graph contraction and the GGT algorithm, which also uses amortization to improve the worst-case complexity.

[4] Gallo *et al.* [6] show how to transform a parametric problem so that all of the arcs into the sink are of constant capacity. For simplicity, in the rest of the paper we assume that the arcs into the sink have constant capacity and the arcs out of the source all have capacities that are linear functions of λ.

A simple algorithm for computing all breakpoints works recursively. At each call, the algorithm gets an interval (λ_1, λ_3) and cuts corresponding to λ_1 and λ_3, and outputs all breakpoints in the interval. Initial values of λ_1 and λ_3 that are less than and greater than all breakpoints, respectively, are easy to find (see [6]).

Let $a_1 + \lambda b_1$ and $a_3 + \lambda b_3$ be the parametric capacities of the two input cuts. Set $\lambda_2 = (a_1 - a_3)/(b_3 - b_1)$ and compute the minimum cut corresponding to λ_2. If the parametric capacity of the cut is not equal to $a_1 + \lambda b_1$ or $a_3 + \lambda b_3$, then λ_2 is not a breakpoint, and we recursively find all breakpoints on (λ_1, λ_2) and on (λ_2, λ_3). Otherwise, it is a breakpoint, and we output it. Then, if the capacity is equal to $a_1 + \lambda b_1$, we recurse on the interval (λ_2, λ_3). In the other case, we recurse on (λ_1, λ_2). When making a recursive call for the interval (λ_1, λ_2), we contract the vertices on the sink side of the minimum cut corresponding to λ_2. Similarly, when making the other recursive call, we contract vertices on the source side. Each call of the algorithm is dominated by a minimum cut computation, and one can show that the number of calls is $O(n)$.

Next we describe the GGT algorithm. The algorithm uses amortization. One way to use amortization in the context of the simple algorithm is to note that when recursing on (λ_2, λ_3), one can use the distance labels (on the sink side of the computed cut) from the current flow computation and amortize the cost of such recursive calls over one maximum flow computation. Note that the distance labels on the source side of the cut are "infinite" so the other recursive call cannot be amortized. To obtain the desired bound, the GGT algorithm makes sure that the cost of the flow computation on the bigger graph is amortized.

To achieve this, the algorithm runs two flow computations in parallel; forward from the source and backward from the sink. Assume that the forward computation finishes first; the other case is symmetric. Then if the sink side of the resulting cut has at least as many vertices as the source side, we disregard the result of the backward computation. Otherwise, we finish the backward computation and keep the labels on the source side of the cut, which is at least as big as the sink side. This way the GGT algorithm amortizes the cost of the bigger recursive call at each level, leading to the desired time bound. See [6] for details.

2.3 Implementation Issues

We implemented two versions of the Gallo-Grigoriadis-Tarjan algorithm. The complete version (GGT) uses amortization and bidirectional flow computations. Our implementation uses the gap and global relabeling heuristics (see [4], *e.g.*), but does not use the dynamic tree data structure, so its running time bound is $O(n^2\sqrt{m})$. We also implemented a simple version of the algorithm (SIMP) that starts each maximum flow from scratch and uses the forward computation only. Otherwise the implementation is similar and uses the gap and global relabeling heuristics as well. This implementation has a worse asymptotic bound but smaller constant factors. The efficiency of the resulting implementations requires careful implementation of the contraction operations, including maintaining implicit flows on contracted arcs.

Dealing with precision. The above discussion assumes unlimited precision arithmetic. Because of the multiplicative factors in the parametric cut capacities, one may need high precision to distinguish between adjacent breakpoints. However, using high-precision arithmetic is expensive, and in some applications one may not need to distinguish between breakpoint values that are close together. Our approach is to use 64-bit integer arithmetic and distinguish only between breakpoints that are far enough apart. Our implementation can miss some breakpoints, but for each missed breakpoint we find a value that is close. Note that using (even double precision) floating point arithmetic does not avoid numerical issues and may lead to correctness and termination problems.

Our implementation starts by selecting an integer multiplier M and multiplying all capacities by M. The value of M is selected so that for the highest value of λ the total capacity of arcs from the source is less than 2^{62}, and for the lowest value of λ the same holds for the arcs into the sink. This choice of M guarantees that flow excesses do not exceed 2^{62}, overflow errors will be detected, and our correctness checker, which needs an extra bit of precision, can be implemented.

During the algorithm initialization, when calculating the initial range, we round λ_1 down and λ_3 up to the nearest integer. During the algorithm execution, we round the value of λ_2 down.

Note that because of the rounding, a value x we output may not be a breakpoint. However, the following properties hold. These properties follow from the fact that we evaluate the parametric capacity function at points which are integer multiples of $1/M$.

1. If we output a value x, then there is a breakpoint in the interval $[x-1/M, x+1/M]$.
2. For every breakpoint y, we output a value in $[y-1/M, y+1/M]$.
3. For every two distinct x_1 and x_2 we output, there are corresponding minimum cuts $(X_1, \overline{X_1})$, $(X_2, \overline{X_2})$ such that the parametric capacities of the two cuts are different.

Note that if we restrict the precision of the values we output, then this is the best we can do.

In addition to outputting the approximate breakpoint parameter values, we build a data structure containing the corresponding cuts. Since the cuts are nested, the data structure is an ordered list of vertices, with a pointer to the last vertex of the source-side set for each cut. Note that if all distinct breakpoints are at least $2/M$ apart, the cuts correspond to the true breakpoint values, and can be used to compute the exact breakpoint values.

3 Star Balancing Algorithm

3.1 Algorithm Description

First, we briefly review the star balancing algorithm. For a more detailed description, the reader should consult [14] and [13]. The star balancing algorithm

is an algorithm for solving instances of the parametric maximum flow problem that meet the following constraints:

- the network is *bipartite*, that is, $V \backslash \{s, t\}$ can be partitioned into sets V_1 and V_2 such that all arcs from s are to members of V_1, all arcs to t are from members of V_2, and all other arcs are from members of V_1 to members of V_2
- all arcs from s have capacity λ
- all arcs to t have constant capacity
- all other arcs have infinite capacity.

For each arc (s, u), we define $\lambda(f, u)$ to be the unique value of λ such that $c((s, u), \lambda) = f(s, u)$, and refer to it as u's λ-*value*. Additionally, it will be useful to have notation for describing the changes made to the flow during the process of the algorithm's execution. Define a z-straddling α-move to be the process of starting from an initial flow f and pushing $\alpha > 0$ units of flow along a simple cycle (s, u_1, v, u_2, s) for which $\lambda(f, u_1) + \alpha \leq z \leq \lambda(f, u_2) - \alpha$. Any z-straddling move for any z is defined to be a *balancing move*.

The star balancing algorithm begins by replacing the arcs from the source with arcs of infinite capacity, and then finding an arbitrary maximum flow in the resulting network, which can be done in linear time. Next, the algorithm repeatedly *balances* members v of V_2 by changing the current flow f to a new flow f' via modifying the flows on arcs among $\{(s, u) \cup (u, v) \mid (u, v) \in A\}$ so that there are no remaining balancing moves involving v. Note that if flows are constrained to be integral as is the case in the implementation described in this paper, there may be remaining fractional balancing moves, but no remaining *integral* balancing moves.

Balancing a star can be accomplished in time linear in the degree of the vertex [13]. Balancing can be done in any order, and is repeated until a sufficient stopping condition is reached. Theoretical analysis of the algorithm [13] assumes round-robin balancing (*i.e.*, repeatedly iterating over a list of the members of V_2), although our implementation uses a working set heuristic [14] that is different from simple round-robin balancing (See Section 3.2).

3.2 Implementation Details

Next we describe a few details of our implementation of the star balancing algorithm.

First, although balancing a vertex $v \in V_2$ can be accomplished in time linear in the degree k of v [13] using weighted selection, our implementation uses sorting and takes $O(k \log k)$ time. In practice, we found that using the sorting-based algorithm was just as good as using the linear-time algorithm, probably because only a small amount of time was spent balancing vertices of high degree for the inputs we tried and because the sorting-based algorithm had lower overhead since it uses a library sorting routine.

Second, rather than using round-robin balancing, we used the working set heuristic originally introduced and described in [14]. It does round robin balancing, but if a vertex $v \in V_2$ does not cause the flow to change during an iteration,

it is marked "dead" and left out of future iterations, until all members of V_2 are dead, at which point all members of V_2 are returned to "live" status. The upper bounds proved for round-robin star balancing apply to the working-set variant, and the same pathological long path example shows that this analysis is tight for the working-set variant as well. In practice, the working set heuristic seems to result in a significant speedup on many real-world and synthetic inputs.

Third, the stopping rule that we use for this implementation is slightly different from the stopping rules presented in the theoretical paper [13]. In our implementation, once the working set is empty, all members of V_2 become live, and if one more round of balancing does not change the flow of any arc, then balancing stops. Once balancing stops, we must use the current λ-values to determine the set of breakpoints to report. Two natural options are:

- reporting all distinct λ-values based on the final flow
- reporting the average of λ-values for each *section*, where sections partition the vertices so that every possible remaining balancing move is entirely within a section. (See [13].)

We chose to do the latter, since this guarantees that all reported breakpoints correspond to actual parametric minimum cuts.

3.3 Precision Issues

For a practical implementation, precision issues are important. One option is to work with high-precision or rational numbers. Both of these options introduce significant overhead compared to the use of hardware arithmetic operations, so we instead opted to use 64-bit integers as used in the other two implementations of this paper. In what follows, we discuss the most important issues that arise when using limited precision in the star balancing algorithm.

First, we address the question of how much precision is needed to solve the problem exactly, assuming the fixed capacities are integral. It can be shown that for two distinct breakpoints, λ' and λ'', it is the case that $|\lambda' - \lambda''| \geq 4/|V_1|^2$. This implies that multiplying arc capacities by some multiplier $M > |V_1|^2/2$ ensures that all true breakpoints differ by more than 2, so that if no augmenting path $(u_1, v_1, u_2, v_2, \ldots, v_{k-1}, u_k)$ remains (where $u_i \in V_1$ and $v_i \in V_2$) along which 1 unit of flow can be pushed so as to decrease $|\lambda(f, u_1) - \lambda(f, u_k)|$, then all reported breakpoints will be true breakpoints, and vice versa.

Because balancing only guarantees the non-existence of balancing *moves* (*i.e.*, augmenting paths of two arcs, excluding the two arcs from s), however, and does *not* guarantee that there are no remaining balancing *paths* as described above, such a multiplier is insufficient to guarantee that exactly the true breakpoints are found by the star balancing algorithm. Because of this weakness, pathological examples show that true breakpoints can be up to a distance $\Omega(|V_1|)$ from the reported breakpoints if we only report one λ-value per section.[5] Hence, an

[5] It should be noted that we can eliminate all balancing paths and achieve a precision guarantee identical to those of the SIMP and GGT implementations if we add a

additional multiplicative factor of $|V_1|$ is required (and sufficient, since no section at the end of the star balancing phase can have λ-values that differ by more than $|V_1| - 1$) to guarantee that exactly the true breakpoints are reported.

Whatever multiplicative factor M is used for the capacities, the star balancing algorithm may work with λ-values as high as $M \cdot |V_2| \cdot U$, so this value must fit into a 64-bit integer, which we use to store λ-values. In these experiments, we used a multiplier of $|{V_1}^3|$.

If we are not concerned with finding exactly the true breakpoints, we can use a smaller multiplier and report all λ-values (scaled down by M) as breakpoints at the termination of the balancing phase. Using this approach, there will be some reported breakpoint within $1/(2M)$ of each true breakpoint, but a pathological example can be constructed that shows that reported breakpoints can be as far as $\Omega(\log n/(M \log \log n))$ from the closest true breakpoint. We believe that this lower bound on inaccuracy is tight.

4 Experimental Comparison

In this section we report on experimental performance of SIMP, GGT, and the star balancing algorithm, called SB. These implementations use the same language (C++), compiler (cygwin), optimization flags (-O4), and were run on the same computer, a Hewlett-Packard desktop with a 3.2GHz Pentium 4 processor and 2GB of RAM. However, SIMP and GGT were implemented by a different set of people than those that implemented SB. Also, while these algorithms are for the general problem, SB works only for the special case discussed earlier.

The inputs we used in this experiment were a combination of real and synthetic data. The real data we used were the inputs used in [14]. These datasets are instances of the revenue optimization problem, and correspond to sets of products and orders for various subsets of these products.

In addition, we created synthetic datasets corresponding to each of the real datasets. We computed the degree distributions of vertices on the left and right sides of the bipartitions, and the distribution of the capacities of arcs going into the sink, and used these to generate synthetic networks with statistics similar to each real dataset. The purpose of these synthetic datasets was to examine whether any underlying structure of the problems may have been affecting the relative running times of the algorithms.

We created various other simple synthetic examples to illuminate the degree to which the pathological worst-case running time of SB occurs on various simple problem instances related to the long path example as compared to the more

simple post-processing step that repeatedly pushes one unit of flow along augmenting paths from $u_1 \in V_1$ to $u_2 \in V_1$ so as to balance their λ-values. This post-processing step runs in worst-case time $O(m|V_2|^2)$ but may run significantly faster in practice. Although the post-processing step does not worsen the worst-case running time of the star balancing algorithm, we chose not to incorporate it into the implementation and instead leave the less-than-ideal precision guarantee as part of the specification of the implementation.

robust SIMP and GGT implementations. We started with various lengths of the long path example with uniform sink arc capacities, and generalized this type of example to have variable sink arc capacities and extra arcs, both random and nonrandom, between the left and right sides of the bipartition. We also experimented with some natural problem variants.

4.1 Real Data and its Synthetic Model

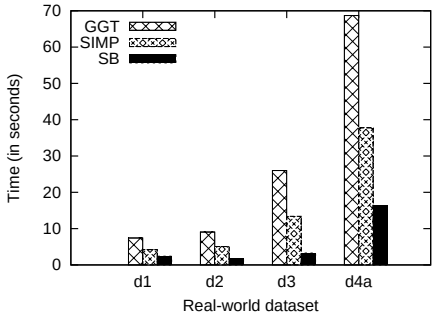

Fig. 1. A comparison of the running times of the GGT, SIMP, and SB implementations on real-world inputs.

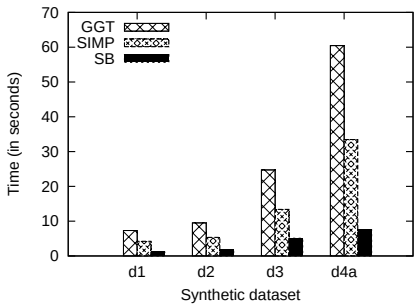

Fig. 2. Comparison results on synthetic datasets of the same sizes, degree distributions, and capacity distributions as the corresponding real-world inputs.

	d1	d2	d3	d4a
GGT	7.41	9.04	25.98	68.68
SIMP	4.21	5.07	13.41	37.75
SB	2.41	1.74	3.27	16.29

Table 1. Tabular data corresponding to Fig. 1.

	d1	d2	d3	d4a
GGT	7.32	9.48	24.71	60.44
SIMP	4.22	5.37	13.41	33.45
SB	1.12	1.83	5.11	7.49

Table 2. Tabular data corresponding to Fig. 2.

First we compare the implementations on real-life problem instances from [14]. There are four datasets, taken from the same real-world application. See Table 3 for the vertex and arc sizes of these datasets. The results of the experiments, displayed in Fig. 1 and Table 1, show that SB, despite its inferior worst-case running time, outperforms GGT and SIMP for these datasets.

To help understand why this is happening, we implemented a synthetic problem generator that models these real-life problems, as discussed earlier. This provided some robustness to the results that used the real data. The results of these experiments, given in Fig. 2 and Table 2, show that the performance gap remains roughly the same.

	d1	d2	d3	d4a
$\|A\|$	454k	625k	1,401k	3,386k
$\|V_1\|$	263	232	344	439
$\|V_2\|$	39k	53k	123k	286k

Table 3. A description of the problem sizes of the real-world problem instances of the product selection problem used in these experiments.

	6400	12800	25600	51200	102400	204800	409600
GGT	0.51	1.22	2.81	6.35	14.41	32.11	70.58
SIMP	0.23	0.57	1.35	3.07	7.04	15.76	34.44
SB	0.48	1.14	2.66	5.88	13.21	30.91	77.86

Table 4. Tabular data corresponding to Fig. 5.

4.2 The Long Path Example and Its Variations

One type of bad example for SB is one in which there is a long path over which many iterations of balancing are required to propagate modest changes in λ-values through the graph. Tarjan *et al.* [13] showed that the running time of the balancing algorithm using round-robin balancing on examples such as the one shown in Fig. 3 is $\Omega(n^3 \log n)$. This is troubling, especially considering that such examples are extremely easy for most parametric maximum flow algorithms to solve. Indeed, as the experimental results show in Fig. 4, SB performs drastically worse on this family of examples than GGT and SIMP.

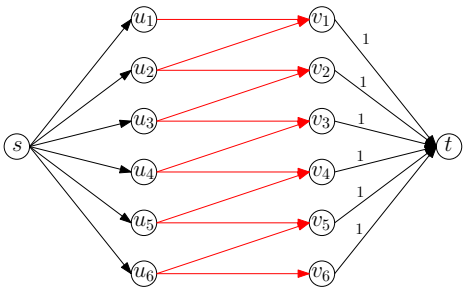

Fig. 3. The long path example, which with unlucky initialization requires $\Omega(n^3 \log n)$ time to finish balancing.

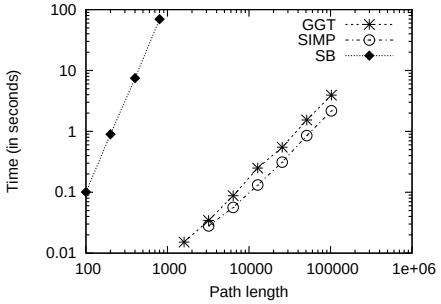

Fig. 4. A comparison of the running times of GGT, SIMP, and SB on increasingly large long path examples. Note that the running time of GGT and SIMP grows roughly linearly, while the running time of SB grows roughly cubically.

The behavior of the balancing algorithm on inputs resembling long paths is troubling, but why might long paths not be a problem in practice? One intuitive explanation for this is that real data may have variability in the capacities of the arcs incident to the sink. Intuitively, this variability can keep SB from needing to push flow over long distances to reach the balanced state. Using capacities

distributed uniformly at random on [1, 1000], this intuition was confirmed as shown in Fig. 5 and Table 4, where such variability improves the performance of SB almost to that of GGT. Fig. 6 and Table 5 also show how the competitiveness of SB increases as the variability of the sink arc capacities increases.

Another reason why long paths may not be a problem in practice is that additional connections in the graph may ameliorate the problem. For example, if a random matching is overlaid on top of a long path graph (*e.g.* Fig. 3), the long path remains but there are many shortcuts for flow to take to reach one end of the long path from the other end. The results, shown in Fig. 7 and Table 6, indicate that this variation on the long path example also eliminates much of the difference in performance between SB and the other two implementations.

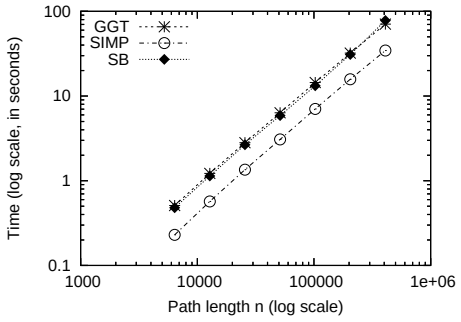

Fig. 5. A comparison of the running times of the GGT, SIMP, and SB implementations on long path inputs (as in Fig. 3) with sink capacities drawn from [1, 1000] uniformly at random. See Table 4 for the data corresponding to this graph.

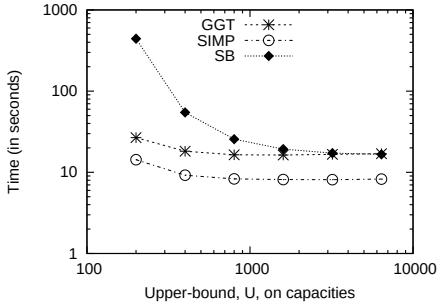

Fig. 6. A comparison among the algorithms on a long path example with 102,400 vertices and capacities distributed uniformly in $[100, U]$. See Table 5 for the data corresponding to this graph.

In fact, as Fig. 8 and Table 7 show, adding additional random matchings continues to improve the performance of SB algorithm relative to those of GGT and SIMP. This leads one to speculate that it might be possible to prove a running time bound that depends on some kind of expansion property of the underlying bipartite graph. This is discussed in Section 5.

	200	400	800	1600	3200	6400
GGT	26.77	18.19	16.44	16.35	16.65	16.98
SIMP	14.30	9.21	8.29	8.13	8.11	8.25
SB	442.10	54.64	25.56	19.31	17.23	16.63

Table 5. Tabular data corresponding to Fig. 6.

	6400	12800	25600	51200	102400	204800	409600
GGT	0.14	0.31	0.67	1.37	2.81	5.78	12.08
SIMP	0.08	0.19	0.40	0.84	1.72	3.58	7.42
SB	0.21	0.71	1.24	3.10	6.93	24.38	111.44

Table 6. Tabular data corresponding to Fig. 7.

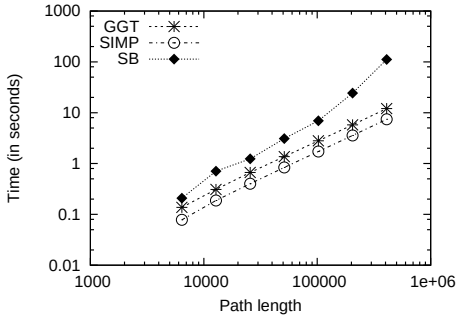

Fig. 7. A comparison of the running times among GGT, SIMP, and SB on long path inputs (as in Fig. 3) in which a random matching is overlaid on top of the long path.

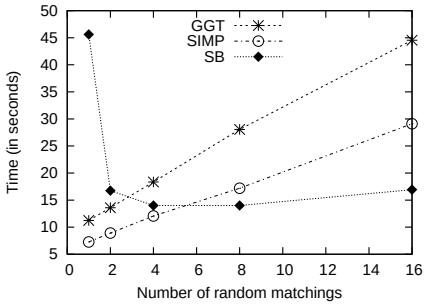

Fig. 8. A comparison of the running times among GGT, SIMP, and SB on long path inputs in which k random matchings are overlaid on top of the long path. The number of vertices in the path was fixed to 409,600.

Another way to view the long path example is as a 1-dimensional checkerboard in which red squares correspond to members of V_1 and black squares correspond to members of V_2 and there are arcs from each member of V_1 to the members of V_2 corresponding to adjacent squares on the checkerboard. Based on this view, we can extend the long path example to higher-dimensional checkerboards in which the paths are not as long for graphs of roughly the same size (See Fig. 9). More specifically, in a d-dimensional checkerboard example with n vertices, the diameter of the graph is $\Theta(n^{1/d})$.

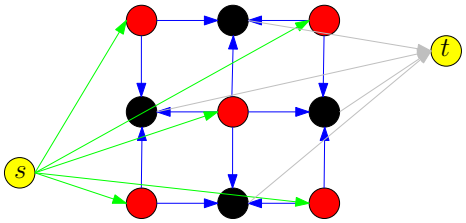

Fig. 9. An example of a 3×3 two-dimensional checkerboard example.

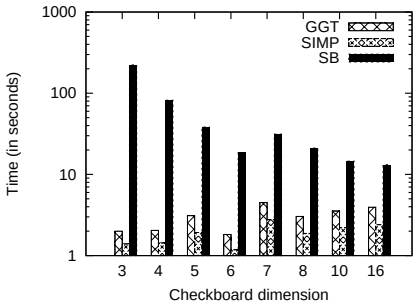

Fig. 10. A comparison of the running times of the GGT, SIMP, and SB implementations on hypercube checkerboard inputs of similar size (about 60,000 vertices each) but increasing dimension.

Indeed, as we increase the number of dimensions of the checkerboard while holding the number of vertices roughly constant, the performance of SB improves relative to that of the GGT and SIMP (See Fig. 10 and Table 8).

	1	2	4	8	16
GGT	11.24	13.59	18.35	28.05	44.53
SIMP	7.26	8.91	12.07	17.19	29.12
SB	45.62	16.75	14.00	14.00	16.93

Table 7. Tabular data corresponding to Fig. 8.

	3	4	5	6	7	8	10	16
GGT	2.00	2.05	3.10	1.82	4.49	3.04	3.53	3.95
SIMP	1.39	1.43	1.92	1.18	2.77	1.87	2.20	2.40
SB	223.05	82.00	38.39	18.56	31.29	21.08	14.59	13.18

Table 8. Tabular data corresponding to Fig. 10.

5 Conclusions and Future Work

Our comparison of the push-relabel algorithms SIMP and GGT with the star balancing algorithm shows that no algorithm dominates the others. This is despite the fact that the push-relabel algorithms have significantly better worst-case bounds. Also, SIMP outperforms GGT in all our experiments. The push-relabel codes are more robust – when they are slower, they are not slower by as much – probably due to the better worst-case bound. For real-life instances from the one application domain we tried, the balancing algorithm was fastest, confirming the earlier claims of Zhang *et al.* [14, 13] based on indirect estimates of GGT performance. In addition, the star balancing algorithm is easier to implement.

Our results show that the pathological behavior of the balancing algorithm when running on long path examples disappears as various changes are made to the network , such as adding sink arc capacity variability, adding random edges, or parameterizing the dimension of the long path example so as to extend it to a higher number of dimensions. This suggests two directions to take for future work regarding using the balancing framework for parametric max-flow.

First, it is clear that the pathological long path behavior is moderated when additional connections between the two sides of the partition provide shortcuts to the long path, or when the long path is cut by variable sink arc capacities. This suggests proving a bound better than the existing bound when the graph has sufficient expansion or some other property. Proving such a bound would be nice in that we would be able to give a better guarantee on the running time of this extremely simple algorithm. Second, in addition to proving a better bound, it would be interesting to see if there were a way to remove the pathological long path behavior by devising a hybrid algorithm that is fast on inputs with long paths, and remains fast on the types of inputs for which the balancing algorithm shows good performance. Such an algorithm would probably only be interesting if it did something other than running two different algorithms in parallel, and it may even be possible to prove a better worst-case running time for such a hybrid algorithm than the current best known worst-case running time.

For the push-relabel algorithms, it seems hard to construct a worst-case example. In fact, it is hard to construct an example on which GGT is significantly faster than SIMP, which should be the case in the worst-case example if the sophisticated amortization used by GGT is needed to achieve the time bound. The only known example where GGT beats SIMP [2] uses the fact that the underlying push-relabel algorithm is asymmetric and can take very different time when solving the equivalent problem on the reverse graph. The GGT algorithm runs on the original and the reverse graph in parallel and on such an example it is faster for this reason. An interesting question is whether the clever amortizations used in GGT is reflected in practice. For example, it would be interesting to see if there exist instances of the parametric maximum flow problem for which GGT and SIMP have similar running times if the parameter value is fixed to any value, but GGT saves a logarithmic factor over SIMP when the entire parametric problem is solved all at once. Finally, it would be interesting to see if there is a faster implementation of a push-relabel algorithm for the special bipartite version of the problem studied in this paper. In this regard, see [1, 6] for results on bipartite maximum flow.

References

1. R. K. Ahuja, J. B. Orlin, C. Stein, and R. E. Tarjan. Improved algorithms for bipartite network flow. *SIAM Journal on Computing*, 23(5):906–933, 1994.
2. M. A. Babenko and A. V. Goldberg. Experimental evaluation of a parametric flow algorithm. Technical report, Microsoft Research, 2006.
3. M. L. Balinski. On a selection problem. *Management Science*, 17(3):230–231, 1970.
4. B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19:390–410, 1997.
5. M. J. Eisner and D. G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *J. ACM*, 23(4):619–635, 1976.
6. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.*, 18(1):30–55, 1989.
7. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.
8. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
9. D. S. Hochbaum. The Pseudoflow Algorithm and the Pseudoflow-Based Simplex for the Maximum Flow Problem. In *6th IPCO*, pages 325–337, 1998.
10. V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *J. Algorithms*, 17:447–474, 1994.
11. J. Mamer and S. Smith. Optimizing field repair kits based on job completion rate. *Management Science*, 28(11):1328–1333, 1982.
12. J. M. W. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17(3):200–207, 1970.
13. R. Tarjan, J. Ward, B. Zhang, Y. Zhou, and J. Mao. Balancing applied to maximum network flow problems. In *Proc. ESA, LNCS 4168*, pages 612–623, 2006.
14. B. Zhang, J. Ward, and Q. Feng. Simultaneous parametric maximum flow algorithm with vertex balancing. Technical Report HPL-2005-121, HP Labs, 2005.