Solving Maximum Flow Problems on Real World Bipartite Graphs

Cosmin Silvestru Negruşeri* Mircea Bogdan Paşoi† Barbara Stanley‡ Clifford Stein§
Cristian George Strat¶

Abstract

In this paper we present an experimental study of several maximum flow algorithms in the context of unbalanced bipartite networks. Our experiments are motivated by a real world problem of managing reservation-based inventory in Google content ad systems. We are interested in observing the performance of several push-relabel algorithms on our real world data sets and also on some generated ones. Previous work suggested an important improvement for push-relabel algorithms on *unbalanced* bipartite networks: the *two-edge push rule*. We show how the *two-edge push rule* improves the running time. While no single algorithm dominates the results, we show there is one that has very robust performance in practice.

1 Introduction

The maximum flow problem is a central problem in graph algorithms and optimization. It models many interesting applications and it has been extensively studied from a theoretical and experimental point of view [1]. In particular, the push-relabel family has been a real success, with both good worst-case running time and efficient implementations [6].

An important special case of the maximum flow problem is the one of *bipartite graphs*, motivated by many natural flow problems (see [14] for a comprehensive list). For over 20 years, it has been known that on *unbalanced* bipartite graphs, the maximum flow problem has better worst-case time bounds. Gusfield et.al. [14] showed that the standard augmenting path algorithms are more efficient in unbalanced bipartite graphs, while Ahuja et al. [3] showed small modifications to existing push-relabel algorithms yielded better time bounds. The improvement is roughly to replace the dependency on n in the time bounds, with a dependence on n_1, the number of nodes on the smaller side of the bipartition. For example, the FIFO push-relabel algorithm, which on a graph with m edges runs in $O(nm + n^3)$ time, can be modified to run in $O(n_1 m + n_1^3)$ time. In many practical applications $n_1 \ll n$, e.g. n_1 may be $\sqrt{n}$, so these improvements yield significant advantages.

Although the improved algorithms for unbalanced bipartite graphs have been known for about 20 years, we are unaware of any published work that implements and tests the bipartite flow algorithms on either simulated data or data from a real application. In this paper, we implement three versions of the bipartite push-relabel algorithm: FIFO, Excess Scaling and Highest Level. We test them on generated data, as well as data that comes from an advertising application within Google.

1.1 Online Advertising Application Online publishers typically have areas on their web pages, called ad slots, where ads can be displayed. Advertisers can reserve a specific number of ad views, called impressions, for one or more of these ad slots. Because a web page gets a limited amount of traffic every day, publishers must verify that the impressions are available before selling them to advertisers. Accuracy is important in computing ad inventory availability. Underbooking results in loss of sales and revenue, while overbooking results in additional cost and potential advertiser dissatisfaction. For many online publishers, these ad sales constitute a critical component of the revenue.

Advertisers are selective about when and where their ads should be displayed; they reserve a number of ad impressions with a set of targeting constraints. These constraints often overlap among reservations, making it difficult to calculate how much available inventory is left to be sold without overbooking. For example, booking a reservation for sports pages impacts how many impressions are left to be sold for a *time-of-day* constraint such as afternoon because some of the sports impressions will occur in the afternoon. To avoid overbooking, afternoon sports impressions must not be counted twice.

Our interest in the unbalanced bipartite flow problem stems from its application to the following availability query problem which can be formulated as follows:

Given: a set of existing reservations, forecasts on how many available ad impressions there are for any disjoint set of targeting constraints, and a reservation r nominated as the subject of our query.

Compute the number of impressions that can be additionally

*Google Inc., Mountain View, CA, cosmin@google.com
†University of Bucharest. All of the work was done while working as an intern for Google Inc., Mountain View, CA, mircea.pasoi@gmail.com
‡Google Inc., Mountain View, CA, bstanley@google.com
§Columbia University. Much of this work was done while the author was visiting Google Inc., New York, NY, cliff@ieor.columbia.edu
¶University of Bucharest. All of the work was done while working as an intern for Google Inc., Mountain View, CA and Zurich, Switzerland, strat.cristian@gmail.com

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

booked for r such that all reservations remain feasible, i.e., impressions assigned to any set of targeting constraints do not exceed the forecasts.

This availability query problem is the simplest to define succinctly, but reservation-based inventory management systems allow additional functionality in their reservations (e.g. limiting the number of impressions delivered to individual users or spacing advertisements out over time), which are beyond the scope of this paper. In Section 4 we show how an *availability query* can be stated as a maximum flow problem [15].

2 Preliminaries

We assume some familiarity with push-relabel algorithms and we omit many details, since they are straightforward modifications of known results. The reader interested in further details is urged to consult the appropriate paper or papers discussing the corresponding result for general networks or one or both of the survey papers [1, 13].

2.1 Network Definitions A network $G = (V, E)$ is called *bipartite* if its vertex set V can be partitioned into two subsets V_1 and V_2 such that each edge has one endpoint in V_1 and the other in V_2. Let $n = |V|$, $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$, and assume without loss of generality that $n_1 \leq n_2$. We call a bipartite network *unbalanced* if $n_1 \ll n_2$ and *balanced* otherwise. We associate with each edge (v, w) in E a finite real-valued *capacity* $u(v, w)$. Let $U = \max\{u(v, w) : (v, w) \in E\}$. Let source s and sink t be the two distinguished vertexes in the network. We assume that $s \in V_2$ and $t \in V_1$. We define the *edge incidence list* $I(v)$ of a vertex $v \in V$ to be the set of edges directed out of vertex v, i.e., $I(v) = \{(v, w) : (v, w) \in E\}$.

A *flow* is a function $f : E \to \mathbf{R}$ satisfying a capacity constraint and a constraint that flow in equals flow out at each non-source, non-sink node:

$$(2.1) \qquad 0 \leq f(v, w) \leq u(v, w) \,, \qquad \forall (v, w) \in E$$

$$(2.2) \sum_{v \in V} f(v, w) - \sum_{v \in V} f(w, v) = 0 \,, \quad \forall w \in V - \{s, t\} \,.$$

The *value* of a flow is the net flow into the sink, i.e., $|f| = \sum_{v \in V} f(v, t)$. The *maximum flow problem* is to determine a flow f for which $|f|$ is maximum.

A *preflow* is a function $f : E \to \mathbf{R}$ that satisfies conditions (2.1) and a relaxation of condition (2.2), $\sum_{v \in V} f(v, w) - \sum_{v \in V} f(w, v) \geq 0 \quad \forall w \in V - \{s, t\}$, which allows flow to accumulate at vertices. The maximum flow algorithms that we study in this paper maintain a preflow during the computation. For a given preflow f, we define, for each vertex $w \in V$, the *excess* $e(w) = \sum_{v \in V} f(v, w) - \sum_{v \in V} f(w, v)$. A vertex other than t with strictly positive excess is called *active*.

With respect to a preflow f, we define the *residual capacity* $u_f(v, w)$ of an edge (v, w) to be $u_f(v, w) = u(v, w) - f(v, w)$, and the residual capacity of (w, v), where (w, v) is the reverse of edge (v, w), to be $f(v, w)$. The *residual network* induced by f is the network consisting only of edges that have positive residual capacity.

A *distance function* $d : V \to \mathbf{N} \cup \{\infty\}$ with respect to the residual capacities $u_f(v, w)$ is a function mapping the vertexes to the set of non-negative integers and infinity. We say that a distance function in a bipartite graph is *valid* if $d(s) = 2n_1$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every edge (v, w) in the residual network. We call a residual edge with $d(v) = d(w) + 1$ *eligible*. The eligible edges are exactly the edges on which we push flow. (In a non-bipartite graph, we set $d(s) = n$.)

We refer to $d(v)$ as the *distance label* of vertex v. It can be shown that if the distance labels are valid, then each $d(v)$ is a lower bound on the length of the shortest path from v to t in the residual network. If there is no directed path from v to t, however, then $d(v)$ is a lower bound on $2n_1$ plus the length of the shortest path from v to s. If, for each vertex v, the distance label $d(v)$ equals the minimum of the length of the shortest path from v to t in the residual network, if such a path exists, or otherwise $2n_1$ plus the length of the shortest path from v to s, then we call the distance labels *exact*.

2.2 Push-Relabel Algorithms All maximum flow algorithms described in this paper are *push-relabel algorithms*, i.e., algorithms that maintain a preflow at every stage. They work by examining active vertexes and pushing excess from these vertexes to vertexes estimated to be closer to t. If t is not reachable, however, an attempt is made to push the excess back to s. Eventually, there will be no excess on any vertex other than t. At this point the preflow is a flow, and moreover it is a maximum flow [9, 11]. The algorithms use distance labels to measure the closeness of a vertex to the sink or the source.

Increasing the flow on an edge is called a *push* through the edge. We refer to the process of increasing the distance label of a vertex as a *relabel* operation. The purpose of the relabel operation is to create at least one eligible edge on which the algorithm can perform further pushes.

3 Flow in Bipartite Graphs

Gusfield et al. [14] showed that the time bounds of several maximum flow algorithms automatically improve when the algorithms are applied *without modifications* to unbalanced networks. The worst-case bounds depend on the number of edges in the longest vertex-simple path in the network. Ahuja et al. [3] show how to modify some push-relabel algorithms to obtain improved time bounds. The improvement can be obtained by using a *two-edge push rule*. According to this rule, we always push flow on two consecutive edges at a time and always starting from a vertex in V_1. This way,

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

procedure *bipush-relabel*(v)
  if there is an eligible edge (v, w)
  then select an eligible edge (v, w);
    if there is an eligible edge (w, x)
    then select an eligible edge (w, x);
      push $\delta = \min\{e(v), u_f(v, w), u_f(w, x)\}$
        units of flow along the path $v - w - x$
    else replace $d(w)$ by
      $\min\{d(x) + 1 : (w, x) \in I(w) \text{ and } u_f(w, x) > 0\}$
  else replace $d(v)$ by
    $\min\{d(w) + 1 : (v, w) \in I(v) \text{ and } u_f(v, w) > 0\}$

Figure 1: The procedure *bipush-relabel*.

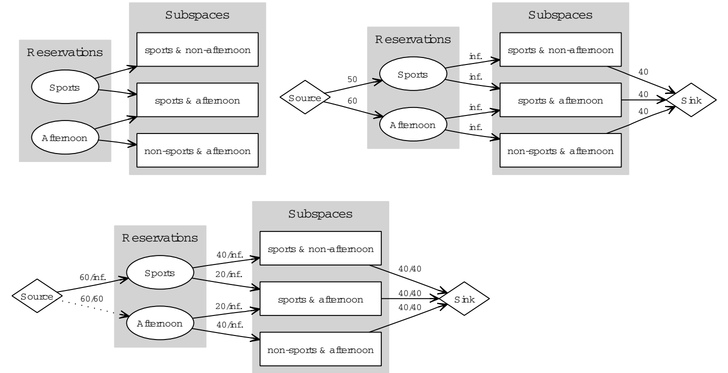


Figure 2: Left: Simple example of two reservations with overlapping constraints. Center: Simple graph initialized for max-flow calculations; Source and Sink nodes have been added and edge capacities have been initialized. Right: Simple graph after the max-flow calculations; all edge assignments have been calculated for the query.

no excess accumulates at vertexes in V_2 and we can attribute all computations to examinations of vertexes in V_1 only. As an outcome of this rule, the running times depend on n_1 rather than n (plus an additive linear term in n to initialize the graph, which is always dominated by other terms.) The bipush idea combined with slight modifications in the data structures used improves many algorithms for maximum flow, minimum-cost flow and parametric flow. (See [3] for results and [2, 4, 8, 9, 11, 12] for background.) In this paper, we focus only on maximum flow.

In Figure 1 we give the building block of the bipartite flow algorithms, the bipush-relabel procedure. Note that it is a modification of the original push-relabel procedure in which we push over two edges at once.

Different algorithms arise from the rule used to choose which vertex on which to execute the bipush-relabel procedure. Each algorithm initializes with the same procedure which sets $d(t) = 0, d(s) = 2n_1$, and $d(v) = 0$ for all other vertexes. It then saturates all edges out of the source and updates distance labels accordingly.

The remainder of the algorithm consists of repeatedly choosing an active vertex to apply the procedure bipush-relabel. The algorithms we study in this paper select the active vertex in one of three ways:

- First-In First-Out (FIFO) [9, 11] maintains a queue of active vertexes.
- Highest Label (HL) [5] chooses the active vertex of highest distance label.
- Excess Scaling (ES) [2] first employs a scaling regime which gradually scales down the amount of excess pushed then chooses the active vertex of minimum distance label.

When optimized for bipartite networks, the worst case running times of these algorithms are $O(n_1 m + n_1^3)$, $O(n_1 m + \min\{n_1^3, n_1^2\sqrt{m}\})$, and $O(n_1 m + n_1^2 \log U)$, respectively.

4 Solving Availability Queries with Maximum Flow Algorithms

The *availability query* can be modeled as a network flow problem. In particular, it can be modeled using a bipartite network where the partition V_1 includes the graph nodes representing reservations and the other partition V_2, typically larger, includes the disjoint subspaces of the reservation constraints. In the example of Section 1.1, V_1 has two nodes for the sports reservation and the afternoon reservation. V_2 contains three disjoint subspaces: sports pages in the morning or evening, sports page in the afternoon, and non-sports pages in the afternoon,

The following requirements must be satisfied when computing the subspaces:

- The subspaces must be disjoint. No two subspaces may include the same impression.
- The subspaces must provide full coverage for all of the reservations' restrictions.

We add edges between the reservation nodes and the subspace nodes for each subspace that satisfy the constraints of each reservation. This is shown in the leftmost graph in Figure 2.

We add a source node that connects to all reservation nodes, and we add a sink node that connects to all subspace nodes. For the edges between the source node and the reservation nodes, we set the capacity to the number of impressions reserved for that reservation. For the edges between the reservation nodes and the subspace nodes, we

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

set the capacity to infinity. And for the edges between the subspace nodes and the sink node, we set the capacity to the predicted number of impressions for the subspace. The center graph in Figure 2 shows the state of the graph at this stage, with edge labels representing the capacity for the edges. As you can see in the example there are two reservations: one for Sports with 50 impressions and one for Afternoon with 60 impressions. The problem we're trying to solve is finding how many more impressions can we deliver on the Sports section while keeping the number of Afternoon impressions the same.

We then run the max-flow algorithm to load the existing reservation into the graph. The allocation of impressions between the subspaces and the reservations is stored on the edges.

To calculate the availability of one reservation node, we disconnect the source node from all of the reservation nodes and attach the source node to only the node being queried. We set the capacity of the edge between source and the node to be infinity and run the max-flow algorithm again. The resulting available number of impressions will be the assignment on this edge. The rightmost diagram in Figure 2 shows the result of the availability query (i.e., the number of available impressions) for the sports page to be 60. However, we must subtract any reserved impressions for that node (i.e., 50 as you can see in the middle diagram from Figure 2) leaving a total of 10 impressions available. The edges are labeled with the capacities in the denominator and the flow assignments in the numerator.

5 Experimental Setup

5.1 Implementations We experimented with six variants of the push-relabel method based on the three node selection rules given in Section 3 and whether we use bi-pushes or pushes. We call these BI-FIFO, BI-ES, BI-HL, GEN-FIFO, GEN-ES, GEN-HL, where BI stands for bi-push version and GEN stands for the general version. As mentioned earlier, we run max-flow twice when solving the availability problem, once to load in the data corresponding to the reservations and once to find how much inventory is available for a particular reservation. In our experiments we are measuring how the algorithms perform in the loading data phase since this corresponds to a full max-flow problem while in the second we're just augmenting an existing flow. All algorithms were coded in C++ and implemented using the same style. To maintain simplicity, we used STL data structures extensively. The code includes extra checks and production logging instructions that may slow it down slightly.

Each node contains a $hash_set$ data structure containing outgoing edges. We used $hash_set$ to get a good insert and delete performance (needed for the advertising application). The $hash_set$ data structure provides an efficient iterator over the adjacency list so all relevant operations still run in expected $O(1)$ time. It is possible that we could improve cache performance by using linked lists or resizable vectors which can provide faster access.

Our implementations maintain residual capacities instead of flows, because the algorithms need the capacities, not the flows, for internal operations. Arc capacities are represented as 64-bit signed integers and the distance labels are represented as 32-bit signed integers since they have the same order of magnitude as the number of nodes.

The algorithms maintain a distance label for each node. For efficiency HL and ES both require the maintenance of a bucket for each possible distance label, each bucket containing all active vertexes at that distance. However, for FIFO we use a simple queue to determine which vertex to scan next.

Heuristics It is by now well known that efficient implementations of maximum flow use two heuristics, the *gap heuristic*, and periodic global relabeling of the entire graph via breadth first search [10]. We implemented these two heuristics in our code. We did some initial tests to verify that these heuristics are still helpful in the bipartite case, and to pick the frequency with which to globally relabel. The results of such tests are described in Section 6.4.

Computing Environment We have implemented the algorithms in C++ using the GCC 4.2.2 compiler (optimization level -O2). Our platform was an Intel ®Core ™2 Quad CPU with four 2.40 GHz processors, each with a cache size of 4 MB running the Ubuntu 6.06 distribution of Linux.

5.2 Data We used two different types of data sets: real world anonymized data sets from Google, and data sets that we generated ourselves. Data used in this paper and generation scripts can be found in the directory `http://www.columbia.edu/~cs2035/bpdata/`
.

We characterize the graphs by several properties: number of nodes, number of edges, average degree, ratio of n_1 to n_2, and values of edge capacities.

5.2.1 Real World Data from Advertising Application We used 33 different ad inventory graphs that arose as described in Section 4. We give the explicit parameters for each graph in the first 5 columns of Tables 5 and 6. In these graphs, the numbers of nodes range from 500 to 300000, the numbers of edges from 40000 to 2.5 million, the n_1/n_2 ratios range from $\frac{1}{5}$ and $\frac{1}{1725}$, and the average degrees are between 1 and 17. These graphs fit well our definition for unbalanced bipartite graphs.

When analyzing the graph capacities, we noticed the graphs had many nodes from V_2 where the capacity to sink was 0. We compared the algorithms on these graphs to a modified version of the graphs where the 0 capacity edges to the sink were replaced by a randomly generated number with mean 1 and deviation $\sqrt{2}$. We noticed that replacing the 0

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

nodes (x 1000)	10	25	50	75	100	150	200	250
ratio	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\frac{1}{5000}$			
edges	$2n$	$3n$	$4n$	$5n$	$10n$			
capacities	lo	hi	two					

Table 1: Parameters for Generated Data Graphs.

bi-fifo	bi-hl	bi-es	gen-fifo	gen-hl	gen-es	
14	4	15	0	0	0	Pushes
9	4	17	0	3	0	Relabels
13	8	12	0	0	0	Time

Table 2: Number of wins for real world data.

capacity edges like this increases the running time by a factor of up to 10, but doesn't change the relative performance of the algorithms, so we will only present the results for the modified graphs.

5.2.2 Generated Data We generated graphs with parameter ranges as given in Table 1. For an experiment, we took the cross-product of all relevant parameter ranges. The values for nodes, ratio and edges are self-explanatory, for capacity, we have three types of capacity distributions: hi where the capacities are random numbers between 0 and 2^{24}, lo where the capacities are random numbers between 0 and 2^8 and two where each capacity is either 10^2 or 10^7 chosen randomly with equal probability.

For each class of graphs, unless otherwise stated, we generated capacities from the sources and sinks to the nodes in V_1 and V_2 by choosing random uniformly nonnegative integers at most the total edge capacity incident to each node.

We have used four generating methods to create families of unbalanced bipartite max flow instances: UniformRandom, Hi-Lo, Rope and ZipF. The latter three are inspired by the generators with the same name from [7] which dealt with solving bipartite matching problems or unit capacity flow problems. For these generators, we did not notice a significant impact from the type of edge capacity distribution and therefore we only report our results for the hi type of capacity distribution. We now describe each family in more detail, using d to denote the average degree in the graph. We created these graphs to understand how the algorithms perform outside our real-world application.

UniformRandom After choosing the values for n_1, n_2, m, and a distribution for the capacities, we choose uniformly at random m edges from the set of all possible edges between the nodes in V_1 and V_2.

Hi-Lo This generator creates a graph with a unique max flow. The nodes in V_2 are split into groups so that each group has n_1 nodes except maybe the last group. We refer to the ith node in the jth group by u_i^j and to the nodes in V_1 by v_i. Node u_i^j has edges coming in from nodes v_p where $max(1, i-d+1) \leq p \leq i$. The capacities of the edge (s, v_i) is equal to $\sum_j c(v_i, u_i^j)$. The capacity of the edge (u_i^j, t) is equal to $c(v_i, u_i^j)$. The maximum flow in the generated graph will use the edges (v_i, u_i^j). The rest of the edges, which are $d-1$ times as many, are there to make the solution harder to find.

Rope The nodes in V_1 and V_2 are split in $t = n_1/d$ groups. V_1 is partitioned in groups X_0, X_1, ..., X_{t-1} and V_2 is partitioned in groups Y_0, Y_1, ..., Y_{t-1}. We join groups of nodes in two zig-zag patterns which meet at $t-1$. We use two strategies of adding edges between groups. The groups are $X_i, Y_{i+1}, X_{i+1}, Y_i$ and also X_{t-1}, Y_{t-1}. The first strategy is to add edges from the nodes in Y to nodes in X and make sure that the capacities from the flow to the nodes in V_1 and from the nodes in V_2 are set up so that they allow as much flow to go from X to Y. We use this strategy for groups where i is even. Then the second strategy is to add, for each node v in Y, $d-1$ random edges that go from X and end in v. These edges of the second type make finding the solution harder.

ZipF We also added another class of graphs where the edges follow a Zipfian distribution which is similar to the distributions for real world and scale-free networks. We added the edge (v_i, u_j) in this graph with a probability proportional to $1/(ij)$. This generator makes the graph dense near the nodes v_1 and u_1 while it's pretty sparse near the nodes v_{n_1} and u_{n_2}.

5.3 Testing Methodology Tests were ran using a combination of bash and Python scripts and C++ code.

We report running times, pushes and relabels, the latter two being a machine-independent measure of each algorithm. We ran each test three times to make sure the recorded time is accurate. The running time is the CPU time in seconds and excludes the input and output times.

6 Experimental Results

We addressed several questions in our experiments. First, we wanted to verify that the bipartite algorithms do indeed perform better than the general algorithms. Second, we wanted to understand the relative performance of the different bipartite variants in terms of running time, pushes, and relabels. After running these experiments on the real-world data, we used generated data in order to further validate our conclusions and to be able to control parameters of the input graphs and measure the performance of the algorithms with respect to these parameters. We also wanted to validate whether the gap and global relabelling heuristics improve performance on bipartite graphs.

6.1 Experiments on Real World Data We ran the six implementations on all 33 real world data graphs and recorded the number of pushes, number of relabels, and time. We then

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

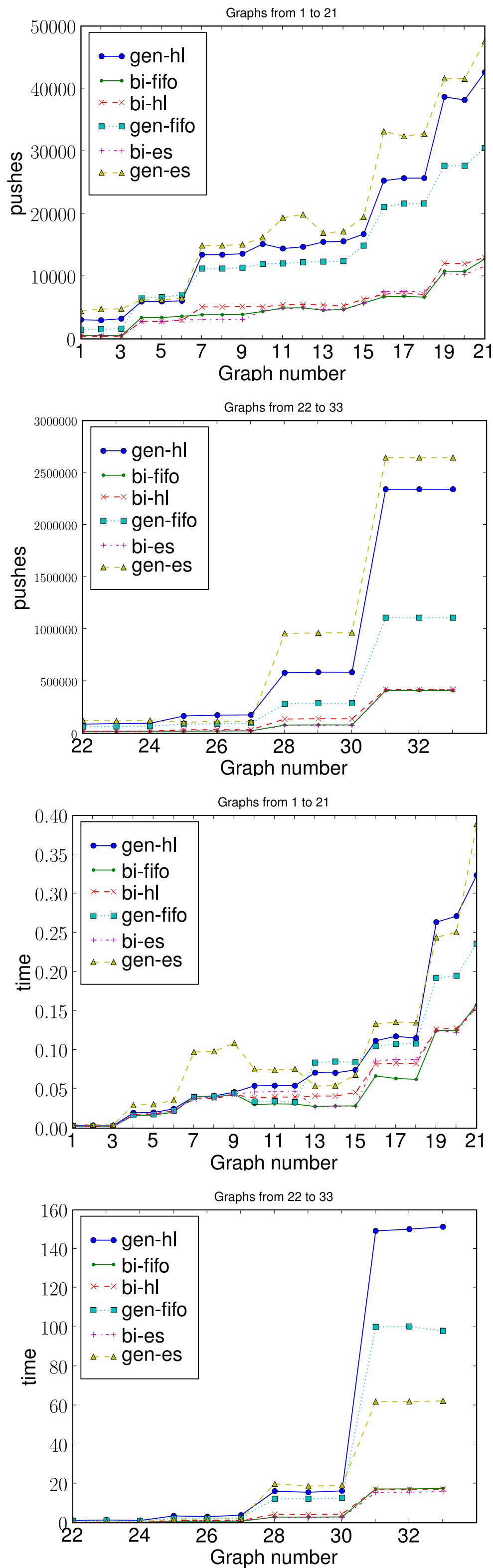


Figure 3: The number of pushes and running time for each of the six algorithms on the 33 real world data sets. Small and large are plotted separately using different scales.

sorted the graphs by the median number of pushes (taken over all algorithms). We used this order to split the graphs into two classes:

- $small$ — the first 21 graphs having median number of pushes below 50000
- $large$ — the remaining 12 graphs

We plot the results separately for the $small$ and $large$ graphs in Figure 3. We see that for both pushes and for time, the bipartite algorithms all perform significantly better than the non-bipartite ones, with the bipartite version typically performing between 3 and 10 times better than the non-bipartite ones.

As a further comparison of the algorithms, Table 2 shows, for each algorithm, the number of graphs it "wins", that is, has the lowest count of either pushes, relabels, and time. From this table, we see that FIFO and Excess Scaling perform the best, while Highest Level does not perform as well. This conclusion is in contrast with the results for the non-bipartite case which show that highest level performs the best [6]. Tables 5 and 6, in the appendix, contain detailed results.

We also tested the efficacy of the gap and global relabelling heuristics. For about half of the graphs, they didn't seem to have any significant effect. For the other half, the number of pushes decreased up to a factor of 10 and the number of relabels decreased up to a factor of 60. For the global relabeling heuristic we found that a global relabeling every $10n_1$ steps was the best heuristic. We omit the detailed data in this extended abstract.

6.2 Experiments on Generated Data — UniformRandom We ran our algorithms on the data generated from the cross-product of all the parameters reported in Table 1. Because the results were very similar for different capacity families (hi, lo, and two) we only present here the results for hi. All appear in tabular format in the appendix.

From this data, we look at the correlations of the algorithm performance versus various graph parameters. In Figure 4, we see the correlation between pushes and number of nodes. The shape of all curves is roughly the same, but we see that the bipartite versions grow slower by a factor about 2 to 4. In the second two plots, we look at how the amount of "imbalance" in the graph affects the number of pushes (and thus the running time). The theory suggests that the more imbalanced the graph is, the more the speedup should be. In the second two plots, we see that this hypothesis is verified, by graphing the pushes versus the ratio and then the pushes divided by n_1. We see that the pushes over n_1 is linear in the ratio in the log-log scale. Thus we can see that, having more nodes in V_2 leads to fewer pushes, although the decrease slows down as we have graphs with more nodes or edges. If we plot pushes divided by n_1 we observe an al-

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

most linear behavior, meaning that the number of pushes is roughly proportional with the ratio times the number of of nodes on the smaller side of the bipartition.

We next compute the ratio between the number of bi-pushes in the bipartite version of an algorithm and the number of pushes in the general version. We compute the average ratio by taking all the ratios for different number of nodes. We plot this for each of the three selection rules (FIFO, ES, HL) and for small and large ratios (5 and 1000) and small and large densities (2 and 10). These results are in Tables 3 and 4, where we see that the ratio tends to increase as m/n increases but stays within a range of 0.25 to 0.5.

Finally, we compare the running time of the three bipartite versions (BI-FIFO, BI-HL, BI-ES). BI-ES remains the best for ratio 5, but for ratio 1000 we see Highest Label come on top, followed closely by BI-FIFO, while BI-ES becomes the worse of the three. The results appear in Figure 5 and 6. This difference occurs because BI-ES minimizes the number of pushes but, as the ratio increases the number of relabels dominates. When bi-push is used, all the relabels are done on nodes V_1; increasing the n_1/n_2 ratio and keeping a fixed number of edges increases the average degree for nodes in V_1, thereby making the relabel operations much more expensive. Since, in terms of relabels, Highest Label proves to be the best algorithm, it also performs best in terms of time for cases with ratio 1000.

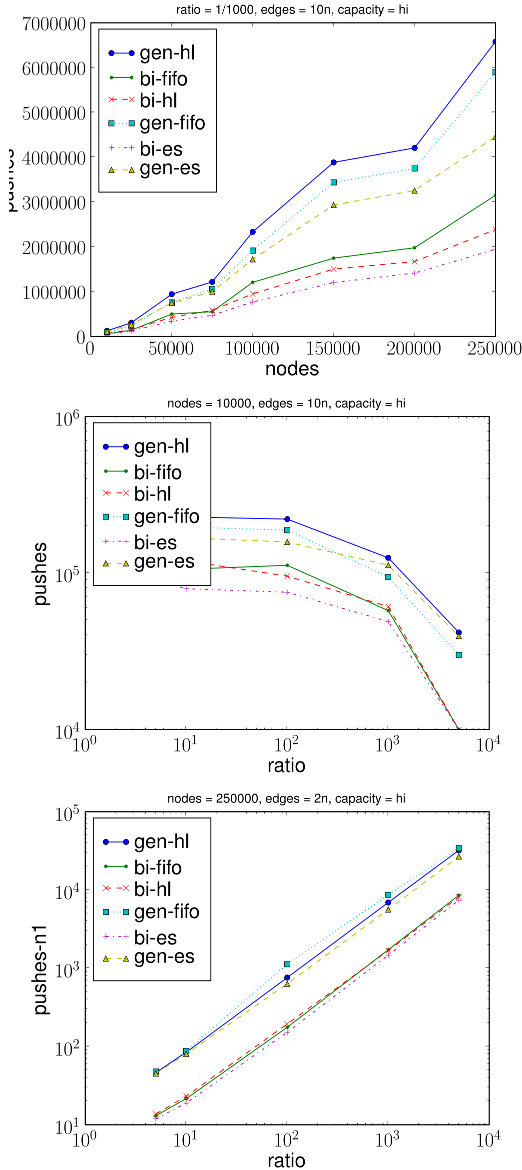


Figure 4: The number of pushes as a function of graph parameters. The first figure plots pushes vs. number of nodes; the second and third plot pushes as a function of the ratio between the left and right sides. In the second, we plot pushes vs. the ratio for a 10000 node graph. In the third, we plot pushes/n_1 vs. the ratio, on a log-log scale.

6.3 Experiments on Generated Data — Hi-Lo, Rope, ZipF

For all three classes of graphs we plot the running times in seconds, number of pushes and number of relabels as the number of nodes increases up to 250000. We do this for two different ratios ($1/5$ and $1/1000$) and for two different number of edges ($2n$ and $10n$). We found that for all three classes the graph instances are harder than UniformRandom on the same configuration.

Hi-Lo The graphs for Hi-Lo appear in Figure 8 in the appendix. For $2n$ edges Hi-Lo seems to be similar to UniformRandom being only up to 2 times bigger in terms of time, pushes or relabels. Also, the relative ordering of the algorithms matches the one for UniformRandom: Excess Scaling is fastest, followed by FIFO and Highest Label.

As we increase the number of edges to $10n$ the instances become much harder, being around 5 times harder in terms of time and around 10 times harder in terms of relabels. While for big ratios the number of pushes is smaller than for UniformRandom cases, relabels become the dominant operation. This is pretty intuitive as the idea behind this generator was to make the right solution hard to find. We now see Highest Label becoming the best performing algorithm, with FIFO keeping its median position.

Rope Rope is particularly interesting as its performance does not decrease as we increase the number of edges from $2n$ to $10n$. For time the results actually remain in the same scale, while in terms of relabels the $10n$ instances are

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

n_1/n_2	5	5	5	5	5	10	10	10	10	10	100	100	100	100	100	1000	1000	1000	1000	1000	5000	5000	5000	5000	5000
m/n	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10
FIFO	0.27	0.33	0.36	0.40	0.54	0.25	0.28	0.32	0.36	0.51	0.18	0.22	0.26	0.31	0.49	0.24	0.32	0.38	0.40	0.57	0.29	0.36	0.46	0.50	0.65
HL	0.28	0.32	0.36	0.37	0.46	0.25	0.29	0.32	0.35	0.44	0.25	0.28	0.30	0.33	0.43	0.27	0.30	0.33	0.36	0.44	0.28	0.30	0.34	0.36	0.42
ES	0.31	0.35	0.39	0.43	0.51	0.28	0.33	0.36	0.40	0.49	0.26	0.30	0.33	0.34	0.41	0.26	0.29	0.31	0.36	0.43	0.26	0.27	0.36	0.36	0.47

Table 3: The ratio of the number of pushes done in the bipartite vs. general version of each algorithm for different graph parameters. Results are averaged over all number of nodes used for testing on UniformRandom.

Nodes	10000					25000					50000					75000				
m/n	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10
FIFO	0.28	0.35	0.40	0.40	0.53	0.26	0.31	0.39	0.46	0.57	0.25	0.29	0.39	0.39	0.63	0.25	0.32	0.36	0.40	0.58
HL	0.27	0.30	0.33	0.35	0.42	0.28	0.31	0.35	0.36	0.43	0.27	0.30	0.34	0.36	0.45	0.27	0.30	0.34	0.37	0.44
ES	0.27	0.32	0.34	0.40	0.44	0.30	0.31	0.39	0.42	0.48	0.29	0.31	0.35	0.39	0.49	0.28	0.30	0.35	0.39	0.47

Nodes	100000					150000					200000					250000				
m/n	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10	2	3	4	5	10
FIFO	0.25	0.30	0.35	0.38	0.56	0.24	0.28	0.35	0.37	0.53	0.23	0.28	0.32	0.38	0.52	0.22	0.27	0.30	0.38	0.51
HL	0.26	0.30	0.33	0.35	0.44	0.26	0.29	0.33	0.35	0.44	0.26	0.29	0.32	0.35	0.44	0.26	0.29	0.32	0.34	0.43
ES	0.26	0.33	0.34	0.38	0.46	0.27	0.29	0.36	0.36	0.46	0.26	0.30	0.34	0.35	0.46	0.26	0.30	0.34	0.35	0.44

Table 4: The ratio of the number of pushes done in the bipartite vs. general version of each algorithm for different graph parameters. Results are averaged for all ratios used for testing on UniformRandom.

easier by a factor of 0.5. For this class of graphs the best algorithm is FIFO, especially when the graph is not very unbalanced. Details appear in Figure 9 in the appendix.

ZipF Graphs generated using ZipF can differ a lot in difficulty for different number of nodes and different random seeds so the results are not as monotone as for the previous graph classes. The plots show that the algorithms behave similarly on ZipF and UniformRandom, having the results for running time, pushes and relabels in similar ranges. Thus, it comes as no surprise that the conclusion is the same: for small ratio Excess Scaling performs the best, while for larger ratios Highest Label seems to be the best choice. Details appear in Figure 10 in the appendix.

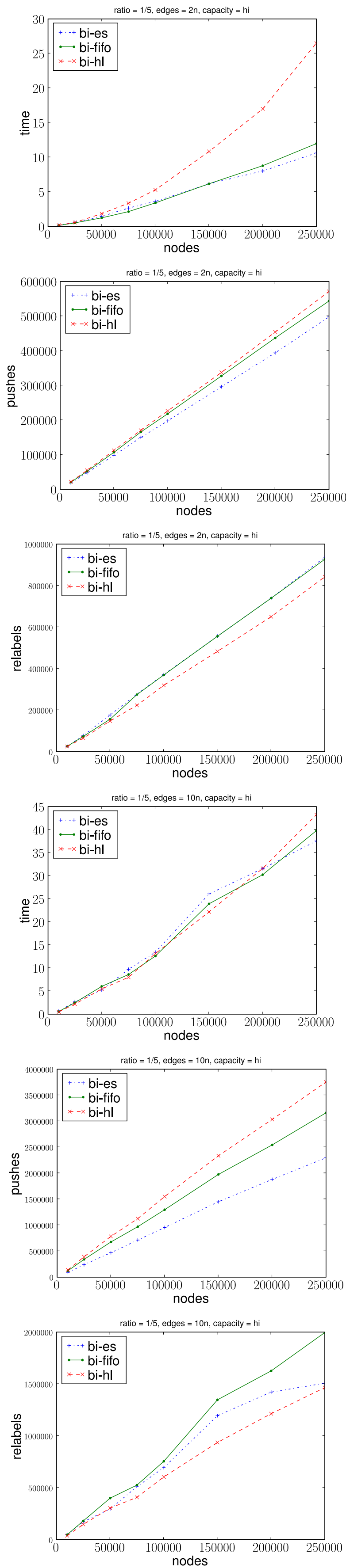
6.4 Evaluation of Heuristics

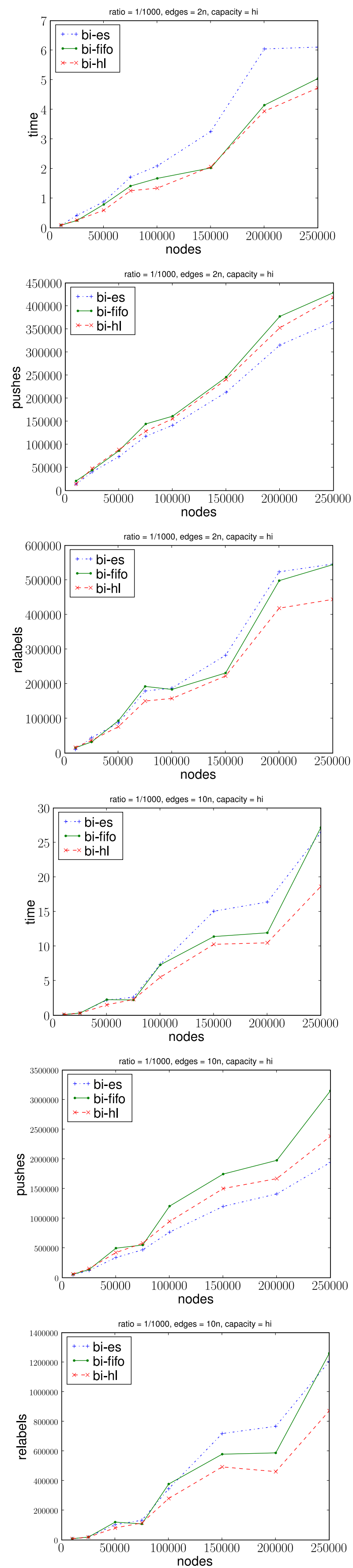
Gap Heuristic Our experimental results show that using the gap heuristic never significantly decreases performance and sometimes drastically improves it. In Figure 7, we present results for graphs up to 50000 nodes. We stop at this value because the running time reaches the order of tens of minutes when the algorithms are ran without any heuristics. Thus, the importance of heuristics would appear even more significant if we included these values. In the figure, we plot the ratio between the times of the algorithms without and with gap heuristic as the number of nodes increases. We can see these vary linearly, a fact which is also consistent for pushes and relabels. We choose to plot running time as it seems to be a linear combination of both pushes and relabels. We also see that the effect of gap heuristic dramatically decreases as the n_1/n_2 ratio increases. This phenomenon occurs because for each node in the graph the distance label has an upper bound of $4n_1$. As the graph becomes more unbalanced ($n_1 \ll n_2$), we have fewer possible distances for the nodes, thus making each distance label bucket more dense and reducing the likelihood of having gaps. Given the low overhead of the gap heuristic, we believe it should always be used, a conclusion consistent with all other work on push-relabel implementations.

Global Relabeling Unlike the gap heuristic, the global relabeling heuristic can have significant overhead. If global relabelings are performed too often, they dominate the running time. If they are performed too rarely, the number of operations performed by the algorithms doesn't get improved. One can perform a new global relabeling after the algorithm did $O(m)$ work since the last relabeling [6]. Our implementations perform a new global relabeling after the number of relabel operations since the last global relabeling is $O(n)$. In particular, we tested $10n_1$, $n/3$, $n/2$, n, $2n$ and $3n$.

While we get consistent performance gains using global relabeling, choosing the right frequency proves to be very difficult as the best frequency varies widely based on each graph's characteristics. For the same number of nodes we get very different behaviors based on the number of edges in the graph or the n_1/n_2 ratio. We found that for our data $10n_1$ for the real world graphs and n for the generated graphs were a good compromise.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.





Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Figure 5: Comparison of time, pushes and relabels versus number of nodes for different values of ratio and the number

Figure 6: Comparison of time, pushes and relabels versus number of nodes for different values of ratio and the number

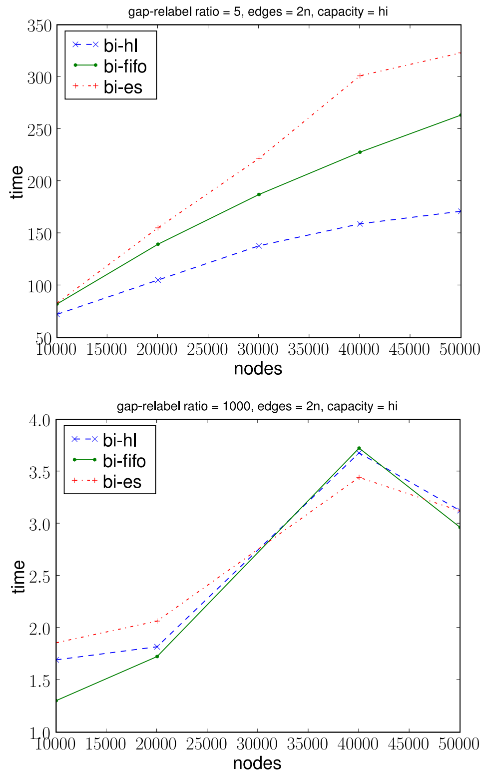


Figure 7: Graphs showing the improvement due to heuristics from the gap relabeling heuristic.

7 Conclusions

The maximum flow problem on unbalanced bipartite graphs is an important scenario in practice and one we encountered in the online advertising industry.

After we run preflow push algorithms on several types of unbalanced bipartite networks, we see that the performance of the algorithms varies for different values of the number of nodes, for different n_1/n_2 ratios and for different number of edges. We conclude that the *two-edge push rule* improves all the performance metrics we measured by a factor of two to four.

Although no single algorithm is dominant in all cases, we find the FIFO algorithm provides consistent performance in practice. Ocasionally it provides the best performance of all the algorithms tested, but never the worst performance. Given that consistent performance is a critical requirement in many real-world applications, our experimental evaluation suggest that the FIFO algorithm should be the method of choice.

In addition, the push-relabel algorithms improve by a wide margin when both the gap relabeling and global relabeling heuristics are used, although one may need to tweak the global relabeling frequency for optimal results.

We have made our test data publicly available, including the Google real-world data and the generated data. Having benefited from others sharing test data and ideas, we welcome further research with our data.

8 Acknowledgements

We thank George Nachman, Jeffrey Oldham and Mihai Pătraşcu for many helpful conversations and suggestions.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] R. K. Ahuja and J. B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37:748–759, 1989.

[3] R. K. Ahuja, J. B. Orlin, C. Stein, and R. E. Tarjan. Improved algorithms for bipartite network flow problems. To appear in *SIAM Journal on Computing*.

[4] R. K. Ahuja, J. B. Orlin, and R. Tarjan. Improved time bounds for the maximum flow problem. *SIAM J. Comput.*, 18:939–954, 1989.

[5] J. Cheriyan and S. N. Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18:1057–1086, 1989.

[6] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[7] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *ACM J. Exp. Algorithmics*, 3:1998, 1998.

[8] G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18:30–55, 1989.

[9] A. V. Goldberg. *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, MIT, Cambridge, MA, Jan. 1987.

[10] A. V. Goldberg and R. Kennedy. Global price updates help. *SIAM J. Discrete Math.*, 10(4):551–572, 1997.

[11] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[12] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. *Mathematics of Operations Research*, 15(3):430–466, 1990.

[13] A. V. Goldberg, R. E. Tarjan, and E. Tardos. Network flow algorithms. In B. Korte, L. Lovász, H. Prömel, and A. Shriver, editors, *Paths, Flows, and VLSI-Layout*, pages 101–164. Springer-Verlag, Berlin, 1990.

[14] D. Gusfield, C. Martel, and D. Fernandez-Baca. Fast algorithms for bipartite network flow. *SIAM J. Comput.*, 16(2), Apr. 1987.

[15] A. Nakamura. Improvements in practical aspects of optimally scheduling web advertising. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 536–541, New York, NY, USA, 2002. ACM.

Appendix

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

#	n_1	n_2	n_1/n_2	Edges	bi-fifo	bi-hl	bi-es	gen-fifo	gen-hl	gen-es	
1	40	23847	1/596	37123	**3408**	**2787**	**2774**	**6579**	**5963**	**6227**	Pushes
					32	9	10	24	22	26	Relabels
					0.04	0.04	0.04	0.04	0.04	0.10	Time (sec)
2	40	25880	1/647	43629	**3611**	**2959**	**3045**	**7053**	**6083**	**6507**	
					50	15	15	35	24	30	
					0.04	0.04	0.04	0.04	0.05	0.11	
3	40	23396	1/585	35759	**3424**	**2803**	**2782**	**6657**	**5981**	**6237**	
					51	15	15	35	25	30	
					0.04	0.04	0.04	0.04	0.04	0.10	
4	85	462	1/5	9004	**515**	**416**	**500**	**1467**	**3053**	**4467**	
					309	221	313	228	350	358	
					0.00	0.00	0.00	0.00	0.00	0.00	
5	85	462	1/5	8994	**516**	**416**	**519**	**1540**	**2983**	**4768**	
					310	218	323	251	337	394	
					0.00	0.00	0.00	0.00	0.00	0.00	
6	85	468	1/5	9420	**515**	**430**	**511**	**1624**	**3232**	**4788**	
					319	235	326	261	368	382	
					0.00	0.00	0.00	0.00	0.00	0.00	
7	40	5857	1/148	33252	**3860**	**5113**	**3063**	**11244**	**13458**	**14927**	
					725	830	489	2025	1940	1191	
					0.03	0.04	0.03	0.09	0.07	0.05	
8	40	5851	1/146	33246	**3860**	**5113**	**3063**	**11244**	**13458**	**14927**	
					725	830	489	2025	1940	1191	
					0.03	0.04	0.03	0.08	0.07	0.05	
9	40	5958	1/146	33609	**3914**	**5164**	**3103**	**11380**	**13610**	**15078**	
					725	830	488	2043	1958	1200	
					0.03	0.05	0.03	0.08	0.07	0.07	
10	35	10998	1/314	21086	**4388**	**5120**	**4479**	**11975**	**15162**	**16224**	
					702	1050	1057	1402	1728	1217	
					0.07	0.08	0.09	0.11	0.11	0.13	
11	35	11120	1/318	21788	**4578**	**5423**	**4703**	**12353**	**15497**	**16936**	
					702	1050	1054	1402	1719	1265	
					0.06	0.08	0.09	0.11	0.12	0.14	
12	35	11293	1/323	22279	**4705**	**5330**	**4675**	**12441**	**15599**	**17167**	
					702	1050	1054	1402	1716	1268	
					0.06	0.08	0.09	0.11	0.12	0.14	
13	289	5632	1/19	26398	**4968**	**5496**	**4957**	**12247**	**14720**	**19875**	
					1846	1963	2031	1766	1733	2424	
					0.02	0.02	0.02	0.02	0.02	0.03	
14	289	5497	1/19	25532	**4940**	**5454**	**4852**	**12048**	**14437**	**19367**	
					1828	1966	1968	1744	1694	2352	
					0.02	0.02	0.02	0.02	0.02	0.03	
15	289	6599	1/19	33051	**5728**	**6338**	**5651**	**14903**	**16754**	**19500**	
					2049	2217	2183	2236	1938	2306	
					0.02	0.02	0.02	0.02	0.02	0.04	
16	934	8792	1/9	43429	**6725**	**7126**	**7577**	**21099**	**25294**	**33166**	
					2985	3500	4081	3221	3053	3797	
					0.03	0.04	0.05	0.03	0.05	0.08	
17	934	8750	1/9	42711	**6660**	**7086**	**7507**	**21643**	**25699**	**32807**	
					2986	3511	4055	3277	3075	3688	
					0.03	0.04	0.05	0.03	0.05	0.08	
18	136	16214	1/119	189121	**10821**	**12080**	**10408**	**27672**	**38677**	**41638**	
					2214	2567	2071	3562	4258	4007	
					0.13	0.13	0.13	0.19	0.26	0.24	
19	136	16234	1/119	189897	**10797**	**12006**	**10303**	**27672**	**38190**	**41568**	
					2176	2561	2042	3565	4223	4011	
					0.13	0.13	0.12	0.19	0.27	0.25	

Table 5: Results for real world data — small graphs. Each entry lists the number of pushes, the number of relabels and the running time in seconds.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

#	n_1	n_2	Edges	bi-fifo	bi-hl	bi-es	gen-fifo	gen-hl	gen-es	
20	934	8804	1/9	43445	**6805** 3022 0.03	**7374** 3630 0.04	**7564** 4102 0.05	**21629** 3390 0.03	**25694** 3136 0.05	**32406** 3678 0.07
21	136	17994	1/132	222951	**12779** 2375 0.16	**13021** 2575 0.15	**11628** 2369 0.16	**30513** 3971 0.24	**42618** 4722 0.32	**47573** 4080 0.39
22	50	17484	1/350	61989	**20953** 3084 0.27	**22014** 3500 0.31	**22397** 1970 0.19	**70404** 14223 1.16	**96921** 16529 1.41	**124551** 5000 0.52
23	50	15636	1/313	56841	**18043** 3579 0.26	**22597** 2802 0.20	**20586** 2505 0.25	**66942** 14399 0.98	**93976** 15172 1.10	**119184** 5500 0.51
24	50	15900	1/318	56577	**19285** 3645 0.28	**19838** 3000 0.23	**19658** 2006 0.18	**65720** 13927 0.98	**89089** 15108 1.15	**126132** 5500 0.53
25	196	87138	1/444	1012334	**21442** 2701 0.72	**34702** 4717 1.10	**24642** 3929 1.13	**89491** 9819 2.32	**167103** 14882 3.10	**105681** 5884 1.88
26	196	90234	1/444	1081231	**24124** 3689 0.77	**35076** 4272 1.20	**27230** 4161 1.21	**97777** 10225 2.41	**177120** 15692 3.88	**114617** 5956 2.03
27	196	87144	1/460	1012256	**23022** 3477 0.72	**35342** 5078 1.12	**25465** 3951 1.12	**92986** 10107 2.24	**175439** 15977 3.50	**117342** 6282 1.84
28	125	54876	1/439	844598	**80091** 11265 2.87	**137927** 18247 4.18	**74444** 10166 2.66	**282921** 52697 12.25	**581287** 66882 15.60	**958992** 70000 18.73
29	125	56586	1/446	864780	**81794** 11256 2.99	**140523** 18136 4.45	**77197** 10156 2.70	**288733** 52854 12.75	**587315** 67415 16.30	**963152** 69000 19.16
30	125	55854	1/452	854585	**80955** 11265 2.91	**139933** 18264 4.34	**75683** 10136 2.66	**288948** 53056 12.23	**586624** 67724 16.15	**966479** 70000 19.91
31	164	282931	1/1725	2523313	**410259** 27221 17.26	**422564** 28039 17.15	**417807** 25500 15.54	**1109226** 162789 100.20	**2341693** 240403 149.28	**2645364** 93481 61.95
32	164	282931	1/1725	2523313	**410259** 27221 17.32	**422564** 28039 17.11	**417807** 25500 15.64	**1109226** 162789 100.39	**2341693** 240403 150.17	**2645364** 93481 62.00
33	164	282963	1/1725	2523477	**410283** 27221 17.55	**422588** 28039 17.14	**417838** 25505 15.88	**1109286** 162794 98.15	**2341728** 240411 151.37	**2645467** 93482 62.30

Table 6: Results for real world data — large graphs. Each entry lists the number of pushes, the number of relabels and the running time in seconds.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

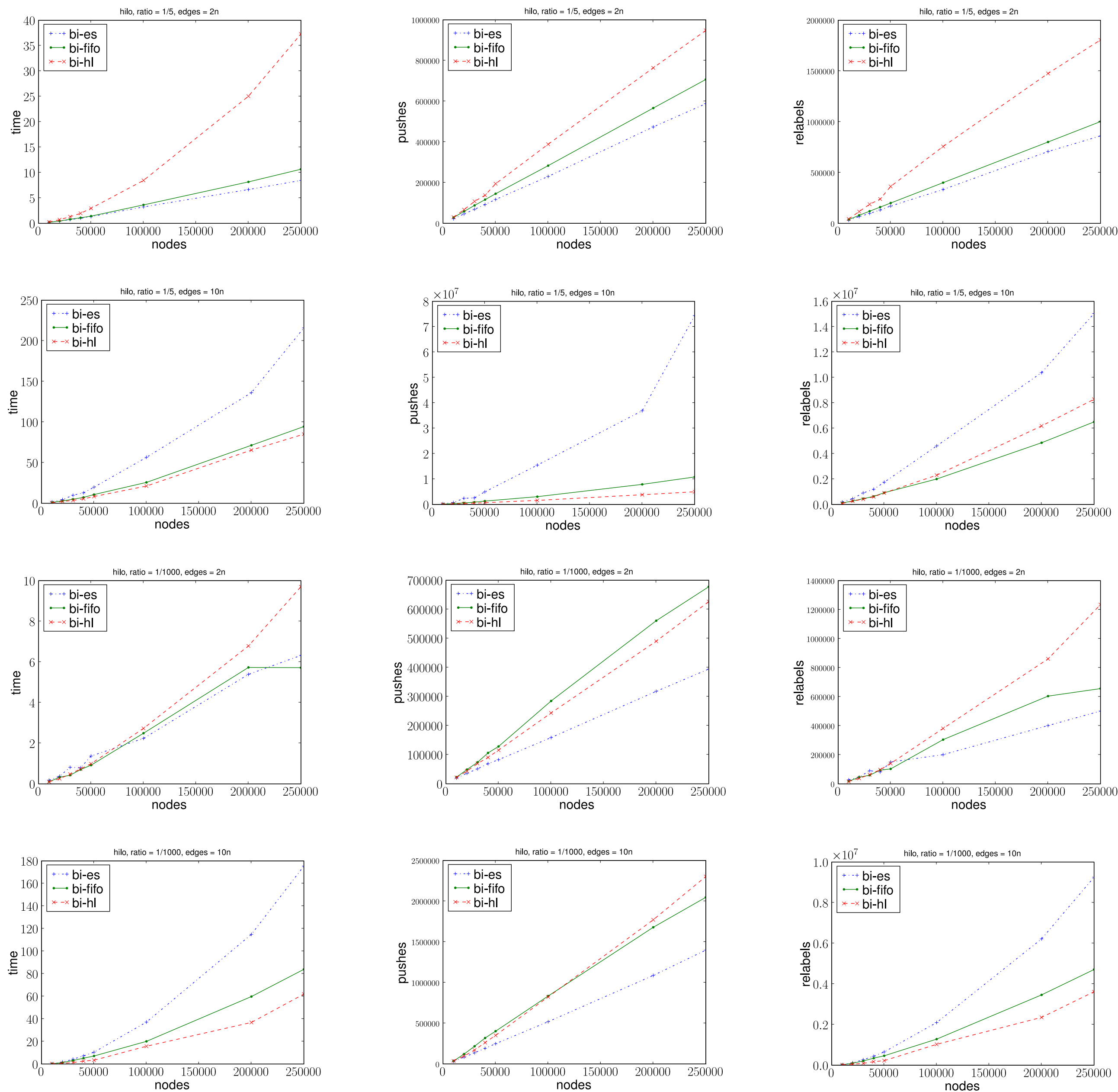


Figure 8: Comparison of time, pushes and relabels versus the number of nodes for different values of ratio and the number of edges for Hi-Lo.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

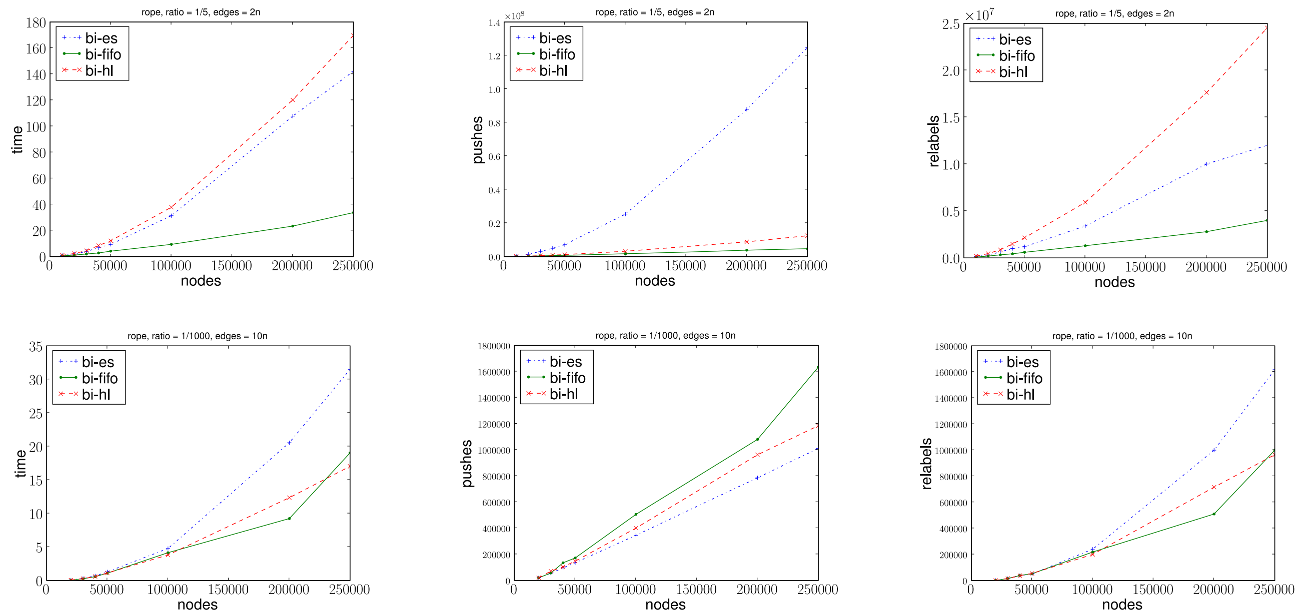


Figure 9: Comparison of time, pushes and relabels versus the number of nodes for different values of ratio and the number of edges for Rope.

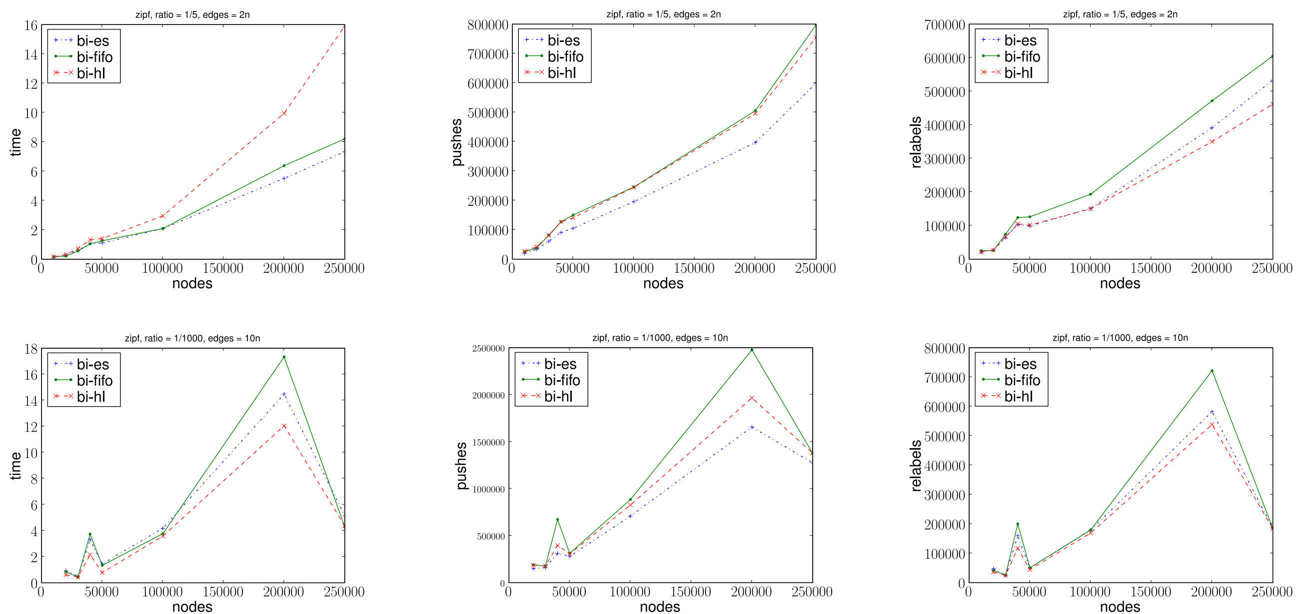


Figure 10: Comparison of time, pushes and relabels versus the number of nodes for different values of ratio and the number of edges for ZipF.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.

Nodes	10000															25000				
m/n	2	2	2	3	3	3	4	4	4	5	5	5	10	10	10	2	2	2	3	3
Capacity	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo
FIFO	0.28	0.29	0.27	0.35	0.35	0.33	0.40	0.41	0.39	0.40	0.39	0.37	0.53	0.53	0.51	0.26	0.29	0.28	0.31	0.40
HL	0.27	0.27	0.29	0.30	0.31	0.32	0.33	0.33	0.34	0.35	0.35	0.36	0.42	0.42	0.43	0.28	0.27	0.29	0.31	0.31
ES	0.27	0.27	0.29	0.30	0.31	0.32	0.33	0.33	0.34	0.35	0.35	0.36	0.42	0.42	0.43	0.28	0.27	0.29	0.31	0.31

Nodes	25000										50000									
m/n	3	4	4	4	5	5	5	10	10	10	2	2	2	3	3	3	4	4	4	5
Capacity	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi
FIFO	0.37	0.39	0.41	0.36	0.46	0.46	0.40	0.57	0.64	0.58	0.25	0.26	0.25	0.29	0.29	0.28	0.39	0.36	0.34	0.39
HL	0.32	0.35	0.35	0.36	0.36	0.37	0.38	0.43	0.42	0.43	0.27	0.26	0.28	0.30	0.30	0.32	0.34	0.34	0.35	0.36
ES	0.32	0.35	0.35	0.36	0.36	0.37	0.38	0.43	0.42	0.43	0.27	0.26	0.28	0.30	0.30	0.32	0.34	0.34	0.35	0.36

Nodes	50000					75000														
m/n	5	5	10	10	10	2	2	2	3	3	3	4	4	4	5	5	5	10	10	10
Capacity	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two
FIFO	0.39	0.38	0.63	0.64	0.58	0.25	0.24	0.24	0.32	0.30	0.27	0.36	0.36	0.34	0.40	0.40	0.36	0.58	0.53	0.51
HL	0.36	0.37	0.45	0.44	0.46	0.27	0.26	0.28	0.30	0.31	0.31	0.34	0.33	0.34	0.37	0.36	0.38	0.44	0.44	0.47
ES	0.36	0.37	0.45	0.44	0.46	0.27	0.26	0.28	0.30	0.31	0.31	0.34	0.33	0.34	0.37	0.36	0.38	0.44	0.44	0.47

Nodes	100000															150000				
m/n	2	2	2	3	3	3	4	4	4	5	5	5	10	10	10	2	2	2	3	3
Capacity	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo
FIFO	0.25	0.25	0.23	0.30	0.30	0.29	0.35	0.35	0.33	0.38	0.37	0.36	0.56	0.56	0.53	0.24	0.24	0.23	0.28	0.28
HL	0.26	0.26	0.27	0.30	0.30	0.31	0.33	0.32	0.34	0.35	0.34	0.37	0.44	0.44	0.46	0.26	0.26	0.27	0.29	0.29
ES	0.26	0.26	0.27	0.30	0.30	0.31	0.33	0.32	0.34	0.35	0.34	0.37	0.44	0.44	0.46	0.26	0.26	0.27	0.29	0.29

Nodes	150000										200000									
m/n	3	4	4	4	5	5	5	10	10	10	2	2	2	3	3	3	4	4	4	5
Capacity	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi
FIFO	0.28	0.35	0.36	0.33	0.37	0.38	0.35	0.53	0.55	0.51	0.23	0.23	0.23	0.28	0.27	0.27	0.32	0.31	0.30	0.38
HL	0.30	0.33	0.33	0.33	0.35	0.34	0.36	0.44	0.43	0.46	0.26	0.27	0.27	0.29	0.30	0.31	0.32	0.33	0.34	0.35
ES	0.30	0.33	0.33	0.33	0.35	0.34	0.36	0.44	0.43	0.46	0.26	0.27	0.27	0.29	0.30	0.31	0.32	0.33	0.34	0.35

Nodes	200000					250000														
m/n	5	5	10	10	10	2	2	2	3	3	3	4	4	4	5	5	5	10	10	10
Capacity	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two	hi	lo	two
FIFO	0.37	0.34	0.52	0.52	0.49	0.22	0.23	0.21	0.27	0.26	0.26	0.30	0.31	0.30	0.38	0.38	0.36	0.51	0.51	0.49
HL	0.34	0.36	0.44	0.44	0.45	0.26	0.26	0.27	0.29	0.29	0.31	0.32	0.32	0.34	0.34	0.33	0.37	0.43	0.43	0.45
ES	0.34	0.36	0.44	0.44	0.45	0.26	0.26	0.27	0.29	0.29	0.31	0.32	0.32	0.34	0.34	0.33	0.37	0.43	0.43	0.45

Table 7: Improvement of bipartite versions for different numbers of nodes and densities. Results are averaged over all n_1/n_2 ratios used for testing on UniformRandom.

Copyright © by SIAM.
Unauthorized reproduction of this article is prohibited.