# ON IMPLEMENTING PUSH-RELABEL METHOD FOR THE MAXIMUM FLOW PROBLEM

BORIS V. CHERKASSKY

CENTRAL INSTITUTE FOR ECONOMICS AND MATHEMATICS
KRASIKOVA ST. 32, 117418, MOSCOW, RUSSIA

*CHER@SUSB.MSK.SU*

ANDREW V. GOLDBERG

COMPUTER SCIENCE DEPARTMENT, STANFORD UNIVERSITY
STANFORD, CA 94305, USA

*GOLDBERG@CS.STANFORD.EDU*

September 1994

Abstract. We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementation. We also exhibit a family of problems for which all known methods seem to have almost quadratic time growth rate.

Andrew V. Goldberg was supported in part by NSF Grant CCR-9307045 and a grant from Powell Foundation. This work was done while Boris V. Cherkassky was visiting Stanford University Computer Science Department and supported by the above-mentioned NSF and Powell Foundation grants.

## 1. Introduction

The maximum flow problem is a classical combinatorial problem that comes up in a wide variety of applications. In this paper we study implementations of the *push-relabel* [13, 17] method for the problem.

The basic methods for the maximum flow problem include the network simplex method of Dantzig [6, 7], the augmenting path method of Ford and Fulkerson [12], the blocking flow method of Dinitz [10], and the push-relabel method of Goldberg and Tarjan [14, 17]. (An earlier algorithm of Cherkassky [4] has many features of the push-relabel method.) The best theoretical time bounds for the maximum flow problem, based on the latter method, are as follows. An algorithm of Goldberg and Tarjan [17] runs in $O(nm \log(n^2/m))$ time, an algorithm of King et. al. [21] runs in $O(nm + n^{2+\epsilon})$ time for any constant $\epsilon > 0$, an algorithm of Cheriyan et. al. [3] runs in $O(nm + (n \log n)^2)$ time with high probability, and an algorithm of Ahuja et. al. [1] runs in $O\left(nm \log\left(\frac{n}{m\sqrt{U}} + 2\right)\right)$ time.

Prior to the push-relabel method, several studies have shown that Dinitz' algorithm [10] is in practice superior to other methods, including the network simplex method [6, 7], Ford-Fulkerson algorithm [11, 12], Karzanov's algorithm [20], and Tarjan's algorithm [23]. See *e.g.* [18]. Several recent studies (*e.g.* [2, 8, 9, 22]) show that the push-relabel method is superior to Dinitz' method in practice.

In this paper we study implementations of the push-relabel method. We evaluate several operation orderings and distance update heuristics. Unlike previous implementations, we use both the global relabeling and gap relabeling [4, 8] heuristics. As a result, our implementation is faster — on some problem families, asymptotically faster — than the previous implementations. We also evaluate different operation selection strategies and find the maximum distance selection best on all problems.

We also exhibit a problem instance generator on which the running time of our implementations seem to grow quadratically. On DIMACS problem families we used extensively in our tests, the growth rate is significantly smaller.

Our implementations and problem generator are available via a mail server.

This paper is organized as follows. In Section 2 we review the push-relabel method. In Section 3 we introduce global relabeling and gap relabeling heuristics. We describe the implementations we evaluated and the problem families used for the evaluation in Section 4. The experimental results appear in Section 5. We present our conclusions in Section 6.

## 2. The Push-Relabel Method

In this section we review some of the basic concepts of the push-relabel method. We assume that the reader is familiar with [17]. (See also [15].) We present the two-phase variant of the method [16], which is the one used in our implementation.

*push*$(v, w)$.
Applicability: $v$ is active **and** $(v, w)$ is admissible.
Action: send $\delta = (0, \min(e_f(v), u_f(v, w))]$ units of flow from $v$ to $w$.

*relabel*$(v)$.
Applicability: $v$ is active **and**
*push*$(v, w)$ does not apply for any $v$.
Action: replace $d(v)$ by $\min_{(v,w)\in E_f}\{d(w)\} + 1$.

FIGURE 1. The update operations. The *pushing* operation updates the preflow, and the *relabeling* operation updates the distance labeling.

A *flow network* is a directed graph $G = (V, E, s, t, u)$, where $V$ and $E$ are node set and arc set, respectively; $s$ and $t$ are the source and the sink, respectively; and $u$ is a non-negative capacity function on the arcs. We define $n = |V|$ and $m = |E|$, and assume that for each arc $(v, w)$, the arc $(w, v)$ is also present. A flow is a function on the arcs that satisfies capacity constraints on all arcs and conservation constraints on all nodes except the source and the sink. The conservation constraint at a node $v$ indicates that the *excess* $e_f(v)$, defined as the difference between the incoming and the outgoing flows, is equal to zero. A *preflow* satisfies the capacity constraints and the relaxed version of conservation constraints that requires the excesses to be nonnegative.

An arc is *residual* if the flow on it can be increased without violating the capacity constraints, and *saturated* otherwise. The residual capacity $u_f(v, w)$ of an arc $(v, w)$ is the amount by which the arc flow can be increased. The residual graph is induced by the residual arcs.

The *distance labeling* $d : V \to N$ satisfies the following conditions: $d(t) = 0$ and for every residual arc $(v, w)$, $d(v) \leq d(w) + 1$. A residual arc $(v, w)$ is *admissible* if $d(v) = d(w) + 1$.

We say that a node $v$ is *active* if $v \notin \{s, t\}$, $d(v) < n$, and $e_f(v) > 0$.

The push-relabel method maintains a preflow $f$ and a distance labeling $d$. Initially the preflow $f$ is equal to zero on all arcs and $e_f(v)$ is zero on all nodes except $s$; $e_f(s)$ is set to a number that bounds the potential flow value (*e.g.* sum of all arc capacities). Initially $d(v)$ is the smaller of $n$ and the distances from $v$ to $t$ in $G_f$. The method then repeatedly performs the *update operations*, *push* and *relabel*, described in Figure 1. When there are no active nodes, the first stage of the method terminates. (The second stage of the method is discussed at the end of this section.)

The update operations modify the preflow $f$ and the labeling $d$. A *push* from $v$ to $w$ increases $f(v, w)$ and $e_f(w)$ by $\delta = \min\{e_f(v), u_f(v, w)\}$, and decreases $f(w, v)$ and $e_f(v)$ by the same amount. The push is *saturating* if $u_f(v, w) = 0$ after the push and *nonsaturating* otherwise. A *relabeling* of $v$ sets the label of $v$ equal to the largest value allowed by the valid labeling constraints.

```
discharge(v).
Applicability: v is active.
Action:        let {v, w} be the current edge of v;
               end-of-list ← false;
               repeat
                   if (v, w) is admissible then push(v, w)
                   else
                       if {v, w} is not the last edge on the edge list of v then
                           replace {v, w} as the current edge of v by the next edge on the list
                       else begin
                           make the first edge on the edge list of v the current edge;
                           end-of-list ← true;
                       end;
               until e_f(v) = 0 or end-of-list;
               if end-of-list then relabel(v);
```

FIGURE 2. The discharge operation.

The efficiency of the push-relabel method depends on the ordering of the update operations. At the low level, these operations are combined as follows. We call an unordered pair $\{v, w\}$ such that $(v, w) \in E$ an *edge* of $G$. We associate the three values $u(v, w)$, $u(w, v)$, and $f(v, w)(= -f(w, v))$ with each edge $\{v, w\}$. Each node $v$ has a list of the incident edges $\{v, w\}$, in fixed but arbitrary order. Thus each edge $\{v, w\}$ appears in exactly two lists, the one for $v$ and the one for $w$. Each node $v$ has a *current edge* $\{v, w\}$, which is the current candidate for a pushing operation from $v$. Initially, the current edge of $v$ is the first edge on the edge list of $v$. The main loop of the implementation consists of repeating the *discharge* operation described in Figure 2 until there are no active nodes. (We shall discuss the maintenance of active nodes later.) The *discharge* operation is applicable to an active node $v$. This operation iteratively attempts to push the excess at $v$ through the current edge $\{v, w\}$ of $v$ if a pushing operation is applicable to this edge. If not, the operation replaces $\{v, w\}$ as the current edge of $v$ by the next edge on the edge list of $v$; or, if $\{v, w\}$ is the last edge on this list, it makes the first edge on the list the current one and relabels $v$. The operation stops when the excess at $v$ is reduced to zero.

The remaining issue is the order in which active nodes are processed. Two natural orders were suggested in [16, 17]. One, the *FIFO algorithm*, is to maintain the set of active nodes as a queue, always selecting for discharging the front node on the queue and adding newly active nodes to the rear of the queue. The other, the *HL algorithm*, is to always select for discharging a node with the highest label. In the worst case, the FIFO algorithm runs in $O(n^3)$ time [16, 17] and the largest-label algorithm runs in $O(n^2\sqrt{m})$ time [5].

The $HL$ algorithm implementation maintains an array of sets $B_i$, $0 \leq i \leq n-1$, and an index $b$ into the array. Set $B_i$ consists of all active nodes with label $i$, represented as a doubly-linked list, so that insertion and deletion take $O(1)$ time. The index $b$ is the largest label of an

active node. During the initialization $s$ is placed in $B_0$, and $b$ is set to 0. At each iteration, the algorithm removes a node from $B_b$, processes it using the *discharge* operation, and updates $b$. The algorithm terminates when there are no active nodes.

At the end of the first stage, the excess at the sink is equal to the minimum cut value and the set of nodes which can reach the sink in $G_f$ induces a minimum cut.

The second stage of the method converts $f$ into a flow. This is done by essentially computing the decomposition of $f$ in the standard way (see *e.g.* [15]) and reducing $f$ on paths from $s$ to nodes with flow excess. To gain efficiency, our implementation computes only a partial decomposition, reducing flow on the above-mentioned paths and on flow cycles as soon the these are discovered. In our experience, the second stage takes significantly less time than the first stage.

## 3. Heuristics

The push-relabel method, as described above, has poor practical performance. Intuitively, because relabel is a local operation, the method loses the global picture of the distances.

The *global relabeling* heuristic updates the distance function by computing shortest path distances in the residual graph from all nodes to the sink. This can be done in linear time by a backwards breadth-first search, which is computationally expensive compared to the push and relabel operations. Global relabelings are performed periodically (*e.g.*, after every $n$ relabelings). This heuristic drastically improves the running time.

Another useful relabeling heuristic is *gap relabeling*, discovered independently by Cherkassky [4] and by Derigs and Meier [8], and based on the following observation. Let $g$ be an integer and $0 < g < n$. Suppose at certain stage of the algorithm there are no nodes with distance label $g$ but there are nodes $v$ with $g < d(v) < n$. Then the sink is not reachable from any of these nodes. Therefore, the labels of such nodes may be increased to $n$. (Note that these nodes will never be active.) If for every $i$ we maintain linked lists of nodes with the distance label $i$, the overhead of detecting the gap is very small. Most work done by the gap relabeling heuristic is "useful": it involves processing the nodes determined to be disconnected from the sink.

Gap relabeling significantly improves the practical performance of the push-relabel method, although usually not as much as global relabeling. These heuristics are not independent – global relabeling discovers nodes disconnected from the sink and makes gaps less likely. However, the overhead of gap relabeling is small. Thus even if no gaps are discovered in a run of an implementation that uses both heuristics, the running time is almost the same as in the implementation that uses only global relabeling. In some cases, however, many gaps are discovered, and the former implementation is significantly faster than the latter.

| optimization level | TEST 1 average running time | | | TEST 2 average running time | | |
|---|---|---|---|---|---|---|
| | real | user | system | real | user | system |
| w/o optm. | 1.2 | 1.2 | 0.0 | 11.1 | 10.8 | 0.1 |
| –O | 0.9 | 0.8 | 0.0 | 8.3 | 7.8 | 0.2 |

FIGURE 3. *Average running times (in seconds) of the test programs in C.*

## 4. Experimental Setup

**4.1. Computing Environment.** Our experiments were conducted on SUN Sparc-10 workstation model 41 with a 40MHZ processor running SUN Unix version 4.1.3. The workstation had 160 Megabytes of memory. All codes used in our experiments were written in C and compiled with the `gcc` compiler version 2.58 using the `-O` optimization option.

We performed the machine calibration experiment designed by the organizers of the First DIMACS International Algorithm Implementation Challenge [19]. Figure 3 shows the average running times of the test programs compiled with and without optimization.

**4.2. Problem Families.** We used seven problem families in our experimental evaluation. Six of these have been used at the First DIMACS Challenge [19]. These families are produced by three generators available from DIMACS. The first generator is RMFGEN of Goldfarb and Grigoriadis [18], the second is WASHINGTON developed by Anderson and students in his seminar, and the third is AC of Setubal (a C version of a generator of Waissi). The seventh problem family is produced by our generator AK. This generator produces problem instances that are hard for the push-relabel and Dinitz' methods.

The DIMACS generators use randomness to produce different instances for the same parameter values (except for a pseudorandom generator seed, if available). Some of these generators do not take a pseudorandom generator seed as a parameter but use system clock to obtain the seed. To make our experiments repeatable, we modified these generators to take the seed argument. For each problem class and problem size, we test five problem instances with different seeds and report the average running times.

The AK generator produces a deterministic network for each value of $n$.

The problem families are as follows.

- **Genrmf-Long.** A network with $n = 2^x$ nodes in this family is generated by the `genrmf.c` program with parameters **a** $= 2^{x/4}$ and **b**$= 2^{x/2}$.
- **Genrmf-Wide.** A network with $n = 2^x$ nodes in this family is generated by the `genrmf.c` program with parameters **a** $= 2^{2x/5}$ and **b**$= 2^{x/5}$.
- **Washington-RLG-Long.** A network with $n = 2^x$ nodes in this family is generated by the `washington.c` program with **function** $= 2$, **arg1**$= 64$, **arg2**$= 2^{x-6}$, and **arg3** $= 10^4$.

- **Washington-RLG-Wide**. A network with $n = 2^x$ nodes in this family is generated by the `washington.c` program with **function** $= 2$, **arg1**$= 2^{x-6}$, **arg2**$= 64$, and **arg3** $= 10^4$.
- **Washington-Line-Moderate.** A network in this family with $n = 2^x$ nodes is generated by the `washington.c` program with **function** $= 6$, **arg1**$= 2^{x-2}$, **arg2**$=4$, and **arg3**$= 2^{(x/2)-2} = \sqrt{n}/4$.
- **Acyclic-Dense.** A network in this family with $n = 2^x$ nodes is generated by the `ac.c` program with the options set to produce fully dense graphs and random capacities with the maximum capacity set at $10^6$.
- **AK.** A network in this family with $4k + 6$ nodes and $6k + 7$ arcs is generated by the `ak.c` program with takes only one parameter, $k$.

**4.3. Implementations Evaluated.** We experimented with several variants of the push-relabel method, but we report only two codes, H_PRF and Q_PRF, which implement the HL and the FIFO algorithms, respectively. Both codes use the global and gap relabeling heuristics. Global relabelings are performed after every $n$ relabelings. Our implementations use the adjacency list representation of the input graph.

We tried other operation selection strategies, including highest excess selection, last-in, first-out selection, and various hybrid strategies. Overall performance of these strategies was worse than that of the HL strategy. We also experimented with various global relabeling frequencies. A simple strategy of performing a global relabeling after $cn$ relabelings for some constant $c$ works quite well. The best choice of $c$ depends on the problem family: an implementation with $c = 1$ can be better than the same implementation with $c = 1.5$ on one problem class but worse on another problem class. The value $c = 1$ used in our experiments seems like a good compromise.

In our experiments, the H_PRF code was the fastest. In particular, H_PRF was faster than Q_PRF on all DIMACS families we consider. This is in contrast to the results of [2, 22], where the HL version was faster than the FIFO version on many but not all families.

To put performance of our codes in perspective, we implemented Dinitz' algorithm [10] (DF). This algorithm performs best in practice among the algorithms not based on the push-relabel method. We also obtained an implementation of the FIFO push-relabel algorithm of Anderson and Setubal [2] (ASF). This implementation uses the global relabeling heuristic only; global relabelings are performed after every $m/2$ relabelings.

When tabulating results of our experiments, we give the running times in seconds. The running time is the user CPU time and excludes the input and output times. To obtain a data point for a code, we make five runs of the code on problems produced with the same generator parameters except for the pseudorandom generator seed.[1] The data we tabulate is the average

[1]Except for the AK generator, which does not use randomness.

over the five runs. The programs exceeding CPU time limit of 2400 seconds (including i/o, which for all problems we study is below 400 seconds) were terminated and the corresponding table entries are left blank.

We plot the data in addition to tabulating it. Our plots use logarithmic scales.

## 5. Experimental Results

Our experiments show the HL implementation H_PRF to be the fastest code on all problem instances we report on. Our FIFO implementation H_PRF is second-fastest. On some problem families the latter implementation performs almost as well as the former, while on other families it is noticeably slower, but never by more than a factor of four. The difference between these two codes is biggest on long (and narrow) networks. On these networks, the highest label selection strategy tends to create many gaps, and the gap relabeling heuristic takes advantage of this. The heuristic nature of gap relabeling is especially clear with H_PRF on the long networks: the number of nodes eliminated by gap relabelings varies drastically from one problem instance to another, and so do the running times.

The gap relabeling heuristic helps more when combined with the HL algorithm than when combined with the FIFO algorithm. The reason for this is as follows. Suppose an implementation of the HL algorithm does not use the gap relabeling heuristic and a gap arises during an execution. Then the implementation wastes time processing nodes which would have been discarded by the heuristic until the distance label of these nodes increases to $n$ or a global relabeling is performed. In a similar situation, the FIFO implementation makes some progress because it processes all active nodes.

Experimental results confirm that the combination of HL selection and gap relabeling is especially effective. In implementations of [2, 22], which do not use gap relabeling, the highest label version was slower than the FIFO version on Acyclic-Dense networks. The same holds for the implementations of [2] on Washington-RLG-Wide networks, where the HL version is much slower than the FIFO version: 1081.3 seconds vs. 41.6 on 65538 node problems. In our tests, H_PRF was always faster than Q_PRF. In particular, for the 65538 node problems, the running times were 13.47 seconds vs. 26.92. With gap relabeling turned off on these problems, the performance of H_PRF degrades substantially more than the performance of Q_PRF, and the latter code becomes much faster than the former.

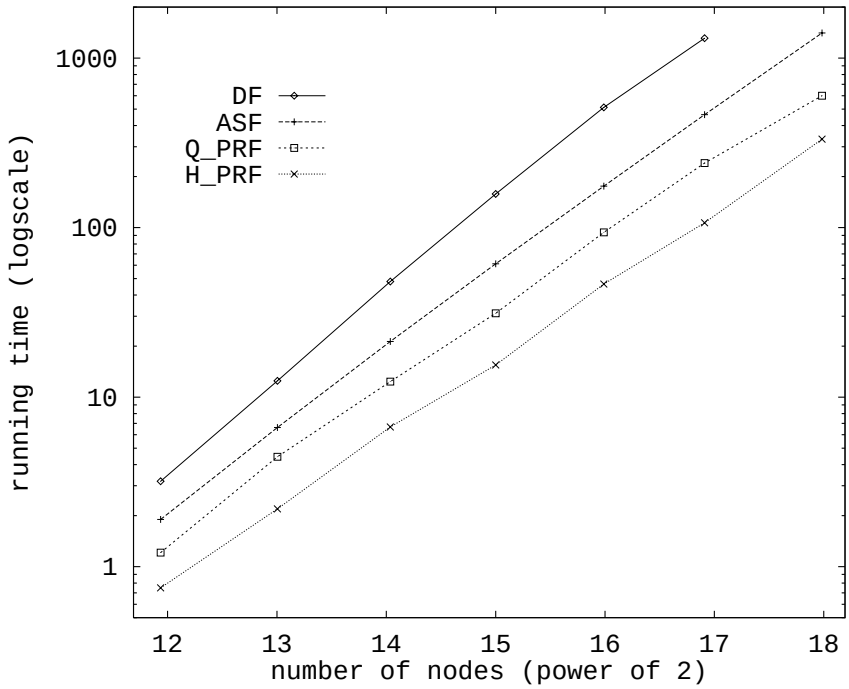
The ASF code implements the same FIFO algorithm as Q_PRF but applies global relabeling after every $m/2$ relabelings (vs. $n$ for P_PRF) and does not use gap relabeling. These differences account for the fact that, with one exception, ASF is slower than Q_PRF. On sparse networks, the relabeling frequency for the two codes is similar, and so is the code performance on many of these networks. On such networks Q_PRF is somewhat faster except for the largest Washington-RLG-Long problems, where ASF is a little faster. For this problem class, global relabeling frequency of ASF, which is about 1.5 times less than that of Q_PRF, works better. On some

problem classes (*e.g.* on the AK problems), Q_PRF is substantially faster because of the gap relabeling heuristic. On dense networks, ASF makes too few global relabelings and performs asymptotically worse than ASF.

Our implementation DF of Dinitz' algorithm was the slowest, often asymptotically, except for the Acyclic-Dense problem family, where it was substantially faster than ASF.

Indirect comparison shows that H_PRF is faster than the implementations studied in [22] on all problem classes studied in both papers, including Genrmf-Wide, Genrmf-Long, Washington-Line-Moderate, and Acyclic-Dense families.

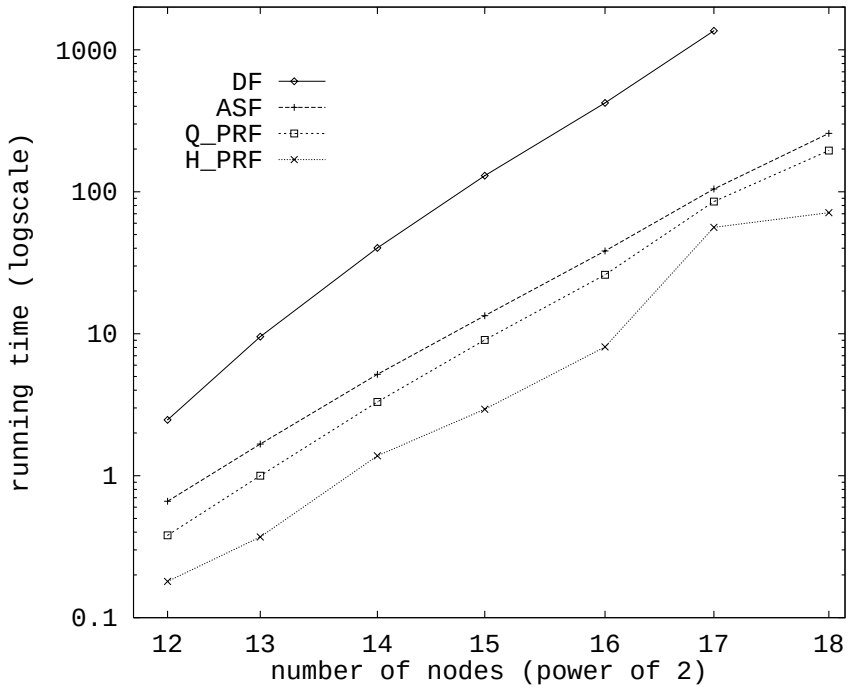
Next we present experimental data for the problem families we studied and make family-specific comments.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 3920 | 18256 | 3.19 | 1.90 | 1.21 | 0.75 |
| 8214 | 38813 | 12.48 | 6.62 | 4.45 | 2.19 |
| 16807 | 80262 | 48.06 | 21.31 | 12.34 | 6.67 |
| 32768 | 157696 | 157.86 | 61.18 | 31.25 | 15.48 |
| 65025 | 314840 | 511.72 | 175.63 | 93.77 | 46.50 |
| 123210 | 599289 | 1310.17 | 464.10 | 240.42 | 106.74 |
| 259308 | 1267875 | | 1406.00 | 599.52 | 332.57 |

FIGURE 4. Genrmf-Wide family data.

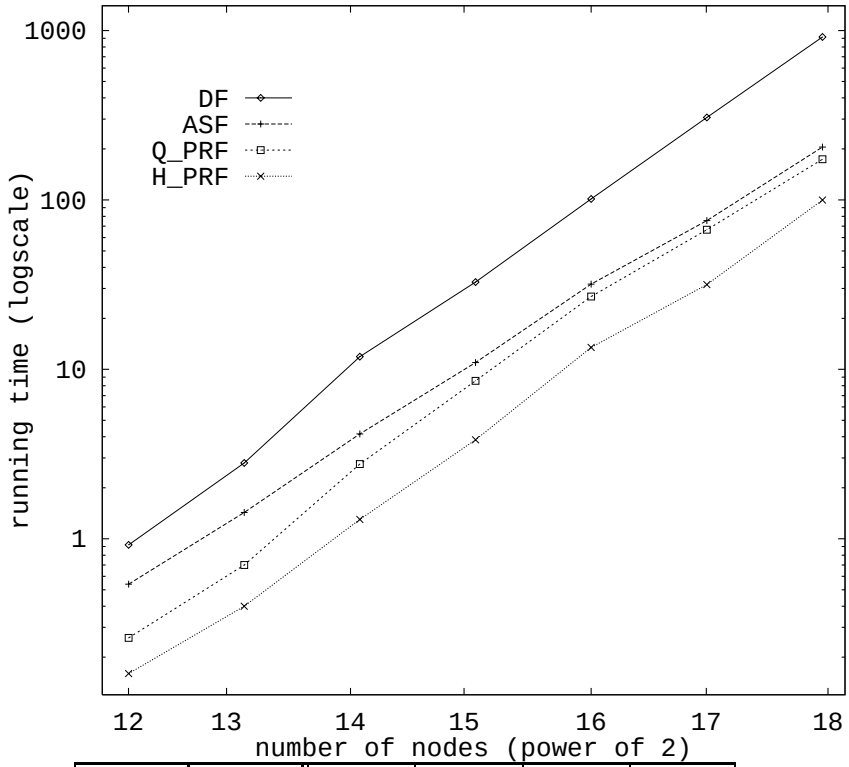
**5.1. Genrmf-Wide Family.** Figure 4 gives data for the genrmf-wide problem family. On this family, H_PRF is asymptotically faster than DF and ASF. H_PRF is faster than Q_PRF by about a factor of two.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 4096 | 18368 | 2.47 | 0.66 | 0.38 | 0.18 |
| 7371 | 33498 | 9.54 | 1.67 | 1.00 | 0.37 |
| 15488 | 71687 | 40.20 | 5.18 | 3.31 | 1.38 |
| 30589 | 143364 | 129.83 | 13.41 | 9.04 | 2.94 |
| 65536 | 311040 | 422.86 | 38.28 | 26.04 | 8.07 |
| 130682 | 625537 | 1360.41 | 104.74 | 85.35 | 56.19 |
| 270848 | 1306607 | | 258.01 | 195.11 | 71.26 |

FIGURE 5. Genrmf-Long family data.

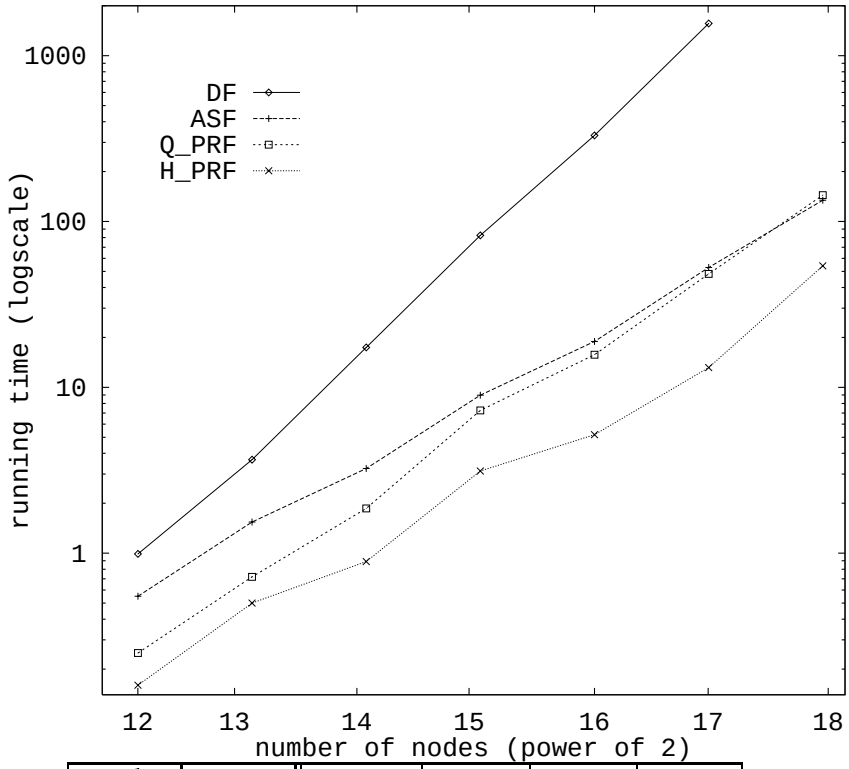
**5.2. Genrmf-Long Family.** Figure 5 gives data for the genrmf-long problem family. Although H_PRF performs best on all problem instances, its performance varies highly depending on the number of gaps discovered. Performance of the two FIFO implementations is similar, with Q_PRF slightly faster than ASF. DF is asymptotically slower than the other codes.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 4098 | 12224 | 0.92 | 0.54 | 0.26 | 0.16 |
| 8194 | 24448 | 2.80 | 1.43 | 0.70 | 0.40 |
| 16386 | 48896 | 11.88 | 4.16 | 2.76 | 1.30 |
| 32770 | 97792 | 32.77 | 10.97 | 8.56 | 3.84 |
| 65538 | 195584 | 101.38 | 31.86 | 26.92 | 13.47 |
| 131074 | 391168 | 306.73 | 75.53 | 66.74 | 31.64 |
| 262146 | 782336 | 916.51 | 205.23 | 173.85 | 99.86 |

FIGURE 6. Washington-RLG-Wide family data.

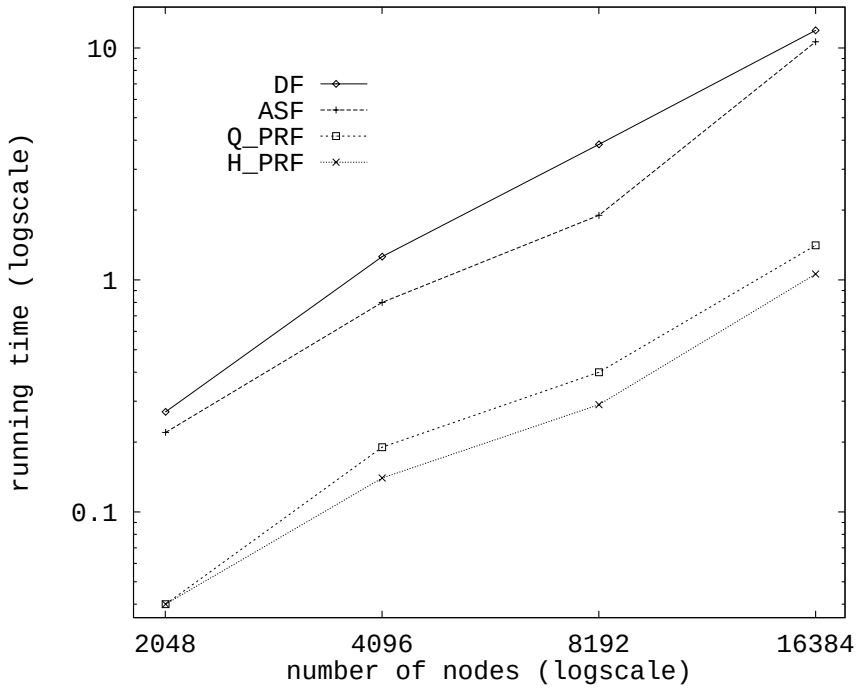
**5.3. Washington-RLG-Wide Family.** Figure 6 gives data for the Washington-RLG-Wide problem family. On this family, H_PRF is the fastest code. ASF is slightly faster assymptotically than Q_PRF; it is slower by a factor of two on the smallest problems but slightly faster on the largest problems. and about a factor of two slower on the smaller problems. DF is asymptotically slower than the other codes.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 4098 | 12224 | 0.99 | 0.55 | 0.25 | 0.16 |
| 8194 | 24512 | 3.66 | 1.54 | 0.72 | 0.50 |
| 16386 | 49088 | 17.40 | 3.24 | 1.86 | 0.89 |
| 32770 | 98240 | 82.52 | 8.97 | 7.26 | 3.13 |
| 65538 | 196544 | 330.62 | 18.92 | 15.74 | 5.18 |
| 131074 | 391168 | 1562.85 | 52.80 | 48.30 | 13.16 |
| 262146 | 786368 | | 134.30 | 144.26 | 54.05 |

FIGURE 7. Washington-RLG-Long family data.

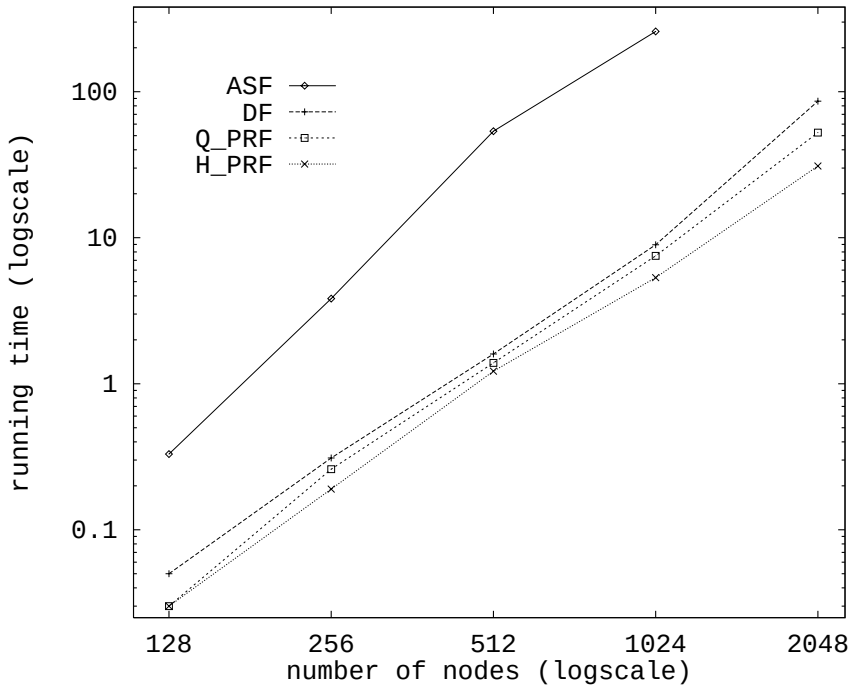
**5.4. Washington-RLG-Long Family.** Figure 7 gives data for the Washington-RLG-Long problem family. The relative performance of the codes is similar to that for the Washington-RLG-Wide family, but the performance difference is somewhat greater. Also, H_PRF exhibits a large running time variation from one instance to another, similar to that on the Genrmf-Long problems.




| nodes | arcs* | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 2050 | 22300 | 0.27 | 0.22 | 0.04 | 0.04 |
| 4098 | 65000 | 1.26 | 0.80 | 0.19 | 0.14 |
| 8194 | 187400 | 3.84 | 1.90 | 0.40 | 0.29 |
| 16386 | 522200 | 11.91 | 10.63 | 1.41 | 1.06 |

FIGURE 8. Washington-Line-Moderate family data. The number of arcs is approximate, since the exact number depends on the seed.

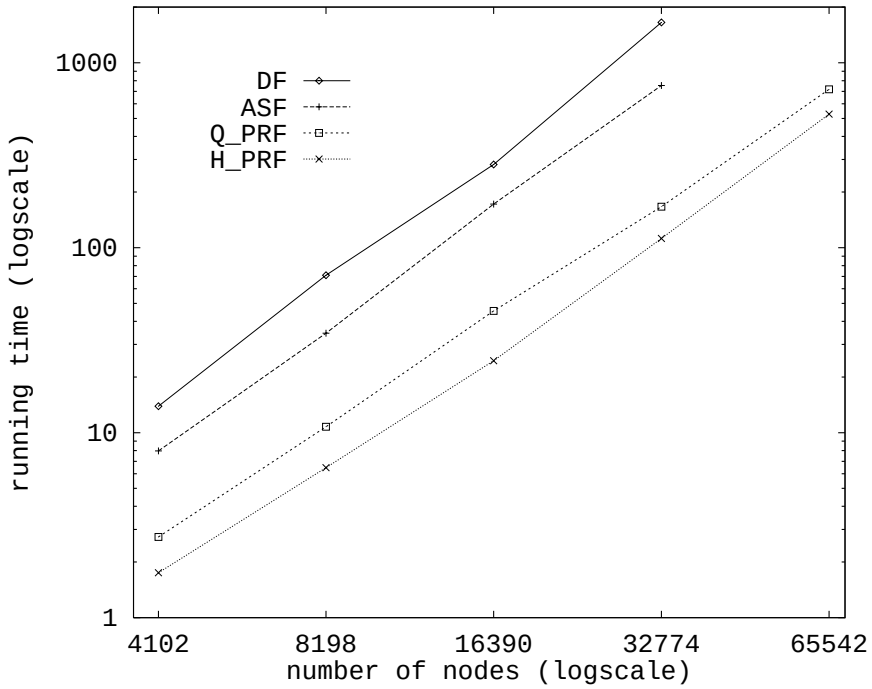
**5.5. Washington-Line-Moderate Family.** Figure 8 gives data for the Washington-Line-Moderate problem family. On this family, H_PRF is the fastest code; Q_PRF is a little slower. The other two codes are significantly slower; DF is the slowest code.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 128 | 8128 | 0.05 | 0.33 | 0.03 | 0.03 |
| 256 | 32640 | 0.31 | 3.83 | 0.26 | 0.19 |
| 512 | 130816 | 1.60 | 53.71 | 1.39 | 1.22 |
| 1024 | 523776 | 8.95 | 258.59 | 7.52 | 5.33 |
| 2048 | 2096128 | 86.13 | | 52.46 | 31.02 |

FIGURE 9. Acyclic-Dense family data.

**5.6. Acyclic-Dense Family.** Figure 9 gives data for the Acyclic-Dense problem family. On this family, H_PRF, Q_PRF, and DF exhibit very similar performance; H_PRF is the fastest and DF the slowest out of these three codes. ASF is asymptotically slower.




| nodes | arcs | DF | ASF | Q_PRF | H_PRF |
|---|---|---|---|---|---|
| 4102 | 6151 | 13.90 | 7.97 | 2.73 | 1.75 |
| 8198 | 12265 | 71.00 | 34.50 | 10.77 | 6.47 |
| 16390 | 24583 | 281.98 | 172.15 | 45.53 | 24.50 |
| 32774 | 49159 | 1651.90 | 753.52 | 166.65 | 112.20 |
| 65542 | 98311 | | | 718.88 | 527.40 |

FIGURE 10. AK family data.

**5.7. AK Family.** Figure 10 gives data for the AK problem family. On this family all codes exhibit a roughly quadratic growth rate. However, the fastest code, H_PRF, is an order of magnitude faster than the slowest code, DF.

## 6. Concluding Remarks

Our best implementation of the push-relabel method, H_PRF was always faster than our implementation of Dinitz' algorithm DF; on many problem families H_PRF was asymptotically faster and on large problems the speedup was sometimes one or two orders of magnitude. (Our implementation of Dinitz' algorithm seems to perform better than that of [2] on the basis of indirect comparison.) We believe that the highest label variant of the push-relabel method

with global and gap relabeling heuristics is the best currently available method for solving maximum flow problems.

Our experiments show that the gap relabeling heuristic should be used together with the global relabeling heuristic in implementations of the push-relabel method, especially in its highest label selection variant.

One can design problem families that are bad for the H_PRF code and not so bad for the Q_PRF code. This fact, combined with the reasonable performance of the Q_PRF code, makes the code a natural candidate to consider when H_PRF does not perform well.

The push-relabel method is superior to Dinitz' method in practice, often by a wide margin when the global and gap relabeling heuristics are used. However, experiments with the AK problem family show that even with the heuristics, the push-relabel implementations can take quadratic time on certain problems. However, the growth rate was significantly smaller for the other six problem families.

## Code Availability

The codes of our implementations and the AK generator are available via a mail server, as are several other codes. For a list of available software and instructions for obtaining the software, send mail to `ftp-request@theory.stanford.edu` and put `send opt-code-info` as the subject line. The reply will contain the desired information.

## Acknowledgments

We would like to thank Robert Kennedy for his help in preparation of this paper, and to Richard Anderson for providing his maximum flow code.

## References

1. R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved Time Bounds for the Maximum Flow Problem. *SIAM J. Comput.*, 18:939–954, 1989.
2. R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.
3. J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a Maximum Flow be Computed in $o(nm)$ Time? In *Proc. ICALP*, 1990.
4. B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. In A. V. Karzanov, editor, *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158, pp. 23–30, 1994.
5. E. Cohen and N. Megiddo. Strongly Polynomial and NC Algorithms for Detecting Cycles in Dynamic Graphs. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 523–534, 1989.
6. G. B. Dantzig. Application of the Simplex Method to a Transportation Problem. In T. C. Koopmans, editor, *Activity Analysis and Production and Allocation*, pages 359–373. Wiley, New York, 1951.
7. G. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1962.
8. U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

9. U. Derigs and W. Meier. An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow. In *ASI Series on Computer and System Sciences*, volume 8, pages 209–223. NATO, 1992.
10. E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
11. J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.
12. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.
13. A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.
14. A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).
15. A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.
16. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.
17. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.
18. D. Goldfarb and M. D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. *Annals of Oper. Res.*, 13:83–123, 1988.
19. D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: F1rst DIMACS Implementation Challenge*. AMS, 1993.
20. A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dok.*, 15:434–437, 1974.
21. V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 157–164, 1992.
22. Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.
23. R. E. Tarjan. A Simple Version of Karzanov's Blocking Flow Algorithm. *Operations Research Letters*, 2:265–268, 1984.