On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky\(^1\) and Andrew V. Goldberg\(^2\)

\(^1\) Central Institute for Economics and Mathematics, Krasikova St. 32, 117418, Moscow, Russia
cher@ceemi.msk.su

\(^2\) Computer Science Department, Stanford University
Stanford, CA 94305, USA
goldberg@cs.stanford.edu

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 implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.

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 \([5]\) 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 \(O\left(nm \log\left(\frac{n}{m\sqrt{l}} + 2\right)\right)\) 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,\])

\* 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.
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 global relabeling and gap relabeling [5, 8] heuristics. As a result, one of our implementations is faster — on some problem families, asymptotically faster — than the previous implementations.

We study two implementations of the highest-level (HL) selection strategy, H_PRF and M_PRF; the only difference between these implementations is that the former uses both global and gap relabelings, while the latter uses only global relabeling. We also study two implementations of the first-in, first-out (FIFO) selection strategy, Q_PRF and F_PRF; the former uses both global and gap relabelings and the latter uses only global relabeling.

Our results suggest that, under HL selection, gap relabeling is a very useful addition to global relabeling: the H_PRF code is sometimes much faster than the M_PRF code and never significantly slower. Under FIFO selection, gap relabeling does not seem very useful: Q_PRF and F_PRF perform very closely on all problem families we consider. We give an informal explanation of these experimental observations in Section 6.

The H_PRF implementation is faster than the other codes on all problem classes we studied. This is in contrast with the work of [2, 22], where on some problem classes the FIFO implementation is faster. In particular, the FIFO implementation of Anderson and Setubal [2] takes 41.6 seconds on Washington-RLG-Wide problems with 65538 nodes compared to 1081.3 seconds for their HL implementation. Performance of our implementations on such problems is as follows: F_PRF, 23.97 seconds; Q_PRF, 26.35 seconds; M_PRF, 328.72 seconds; H_PRF, 13.47 seconds. (See Section 5 for details.) This is a good example of how much gap relabeling can help under the HL selection strategy.

We also exhibit a problem instance generator on which the running time of Dinitz' and push-relabel implementations seem to grow quadratically. On DIMACS problem families we used in the other tests, the growth rate is smaller.

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

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.

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 \textit{excess} \( e_f(v) \), defined as the difference between the incoming and the outgoing flows, is equal to zero. A \textit{preflow} satisfies the capacity constraints and the relaxed version of conservation constraints that requires the excesses to be nonnegative.

An arc is \textit{residual} if the flow on it can be increased without violating the capacity constraints, and \textit{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 \textit{distance labeling} \( d : V \rightarrow N \) 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 \textit{admissible} if \( d(v) = d(w) + 1 \).

We say that a node \( v \) is \textit{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 distance from \( v \) to \( t \) in \( G_f \). The method repeatedly performs the \textit{update operations}, \textit{push} and \textit{relabel}. 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 \textit{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. A \textit{relabeling} of \( v \) sets the label of \( v \) equal to the largest value allowed by the valid labeling constraints.

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 \textit{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 \textit{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 \textit{discharge} operation until there are no active nodes. (We shall discuss the maintenance of active nodes later.) The \textit{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 \textit{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 \textit{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 \cite{16, 17} and the highest-label algorithm runs in $O(n^2 \sqrt{m})$ time \cite{4}.

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 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 essentially by computing the decomposition of $f$ in the standard way (see e.g. \cite{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 \cite{5} and by Derigs and Meier \cite{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 small.

The overhead of maintaining the lists can be charged to relabel operations which change the distance labels. Other work done by the gap relabeling heuristic is "useful": it involves processing the nodes determined to be disconnected from the sink. Therefore a code that uses gap relabeling cannot be much slower than the same code without gap relabeling.
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 -0 optimization option.

4.2 Problem Families

We used seven problem families in our experimental evaluation. Six of these were 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 $a = 2x/4$ and $b = 2x/2$.
- **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 $b = 2x/5$.
- **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 $= 64$, arg2 $= 6$, 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 four codes, H\_PRF, M\_PRF, Q\_PRF, and F\_PRF. All these codes use the global update heuristic, with a global update performed after every \( n \) relabelings. The first two codes use HL selection with and without gap relabeling, respectively. The last two codes use FIFO selection with and without gap relabeling, respectively. 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 H\_PRF code. 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. For example, 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.

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.\(^3\) The data we tabulate is the average over the five runs. The programs exceeding the 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. To improve readability of the plots, we do not plot Q\_PRF data because for all problem families it is within 30\% of the F\_PRF data. We also do not plot M\_PRF data for families where it is within 30\% of the H\_PRF data.

5 Experimental Results

Our experiments show the H\_PRF code to be the fastest on all problem instances we report on. The FIFO implementations F\_PRF and Q\_PRF exhibit similar performance and are the second and the third fastest overall. The M\_PRF code (which is the same as H\_PRF but does not use gap relabeling) exhibits a wide variation in performance: it is about as fast as H\_PRF on some problem families, \(^3\) Except for the AK generator, which does not use randomness.
somewhat slower on others, and on some families M\_PRF is the slowest among all the codes we tested. These results show that, for the problem families we study, gap relabeling is a useful addition to global relabeling for the HL algorithm and not very useful but relatively harmless addition for the FIFO algorithm.

The ASF code implements the same FIFO algorithm as F\_PRF but applies global relabeling after every $m/2$ relabelings (vs. $n$ for F\_PRF). This and the low level implementation details account for the fact that, with one exception, ASF is slower than F\_PRF. On sparse networks, the relabeling frequency for the two codes is similar, and so is the code performance. On such networks F\_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 F\_PRF, works better. On dense networks, ASF makes too few global relabelings and performs asymptotically worse than F\_PRF.

Our implementation DF of Dinitz' algorithm is the slowest overall, and often asymptotically slower than the other codes. However, it is faster that M\_PRF
on the Washington-RLG-Wide family (by a wide margin) and on Acyclic-Dense family (by a small margin). On the latter family, DF is faster than ASF (by a wide margin).

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

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

### 5.1 Genrmf-Wide Family

Figure 1 gives data for the genrmf-wide problem family. On this family, M.PRIF and H.PRIF performance is very close.

### 5.2 Genrmf-Long Family

Figure 2 gives data for the genrmf-long problem family. On this family H.PRIF is somewhat faster than M.PRIF. The HL codes exhibits a larger performance variation than the FIFO codes. DF is asymptotically slower than the other codes.
Fig. 3. Washington-RLG-Wide family data.

5.3 Washington-RLG-Wide Family

Figure 3 gives data for the Washington-RLG-Wide problem family. On this family, H_PRF greatly benefits from gap relabeling; it is faster than M_PRF by a wide margin. M_PRF is asymptotically slower than the other codes.

5.4 Washington-RLG-Long Family

Figure 4 gives data for the Washington-RLG-Long problem family. Here H_PRF performs better than M_PRF. The former code exhibits a large variation in performance. Although M_PRF is slower than then FIFO codes, it seems to have a slightly better asymptotic behavior. DF is asymptotically slower than the other codes.
5.5 Washington-Line-Moderate Family

Figure 5 gives data for the Washington-Line-Moderate problem family. On this family, all our push-relabel codes have similar performance. The other two codes are significantly slower; DF is the slowest code.

5.6 Acyclic-Dense Family

Figure 6 gives data for the Acyclic-Dense problem family. On this family, H_PRF is somewhat faster than M_PRF. DF performs reasonably well on this family. ASF is asymptotically slower than the other codes.

5.7 AK Family

Figure 7 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. M_PRF is almost as fast as H_PRF.
Fig. 5. Washington-Line-Moderate family data. The number of arcs is approximate, since the exact number depends on the seed.

6 Discussion of Gap Relabeling

Our experimental results show that when added to the HL algorithm with global relabeling, gap relabeling sometimes drastically improves performance and never significantly decreases it. When added to the FIFO algorithm with global relabeling, gap relabeling does not have much effect on performance, at least on the problem classes we studied. Below we give an informal explanation of these observations. Our explanation is not a formal proof, and one might be able to construct graphs for which the behavior is different. However, the explanation seems to fit our experimental results.

Suppose a gap arises during an execution of the M_PRF implementation (which does not use gap relabeling). Then the implementation wastes time processing active nodes which would have been discarded by the gap heuristic until distance labels of these nodes increase to \( n \) or a global relabeling is performed. As a result, under HL selection, nodes on the source side of a gap are more likely to be processed than the other nodes.

Thus gap relabeling can save a lot of work when combined with HL selection and global relabeling. Because of its small overhead (see Section 3), gap relabeling does not waste much work.
Fig. 6. Acyclic-Dense family data.

Now suppose a gap arises during an execution of the F_PRF implementation (which does not use gap relabeling). Compared to the Q_PRF implementation, the "wasted" work is in processing nodes with distance labels above the gap. We say that an interval between global updates as bad if at least a quarter of the work during this time interval is "wasted" and good otherwise. Therefore the total time of the good intervals is likely to be at most 4/3 of the total time of the F_PRF implementation. After a bad interval, it is likely that a constant fraction of the remaining nodes will be discarded by the global update at the end of the interval, because active nodes are processed uniformly and the fraction of active nodes behind the gap is likely to be proportional to the fraction of the total number of nondiscarded nodes behind the gap. Thus the number of bad time intervals is likely to be $O(\log n)$. Since the total work done during an interval between global updates (which occur after every $n$ relabelings) is likely to be $O(m)$, the total time of bad intervals is $O(m \log n)$. If the running time of Q_PRF is $\omega(m \log n)$, which is usually the case, then the running time of F_PRF is unlikely to exceed the running time of Q_PRF by a factor much more than 4/3.

Thus gap relabeling is unlikely to save much work when combined with FIFO selection and global relabeling. On the other hand, since the extra overhead of gap relabeling in this case is small, gap relabeling does not waste much work.
7 Concluding Remarks

Our best implementation of the push-relabel method, H.PR, was always faster than our implementation of Dinitz' algorithm DF; on many problem families H.PR 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 HL variant of the push-relabel method with global and gap relabeling heuristics is the best currently available method for solving maximum flow problems.

One can design problem families that are bad for the H.PR code and not as bad for the F.PR code. This fact, combined with the reasonable performance of the F.PR code in our study, makes the code a natural candidate to consider when H.PR does not perform well. F.PR is also better suited for parallel and distributed implementation, and it is simpler than H.PR.

M.PR is much less robust than H.PR and never performs significantly better. Thus gap relabeling should be used in implementations for the HL algorithm.

Q.PR performance is similar to (but overall slightly worse than) F.PR performance, and in this case gap relabeling does not seem to be worth implementing.
With the appropriate heuristics added, 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, push-relabel implementations can take quadratic time on certain problems. On the positive side, 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 Richard Anderson for providing his maximum flow code.

References