Algorithmica (1997) 19: 390–410

Algorithmica

On Implementing the Push–Relabel Method for the Maximum Flow Problem[1]

B. V. Cherkassky[2] and A. V. Goldberg[3]

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 show that the highest-level selection strategy gives better results when combined with both global and gap relabeling heuristics. We also exhibit a family of problems for which the running time of all implementations we consider is quadratic.

Key Words. Algorithms, Network optimization, Maximum flows, Experimental evaluation.

1. Introduction. The maximum flow problem is a classical combinatorial problem that arises in a wide variety of applications. In this paper we study implementations of the *push–relabel* [15], [18] method for the problem.

The basic methods for the maximum flow problem include the network simplex method of Dantzig [7], [8], the augmenting path method of Ford and Fulkerson [13], the blocking flow method of Dinitz [11], and the push–relabel method of Goldberg and Tarjan [15], [18]. (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 [18] runs in $O(nm \log(n^2/m))$ time, an algorithm of King *et al.* [22] runs in $O(nm + n^{2+\varepsilon})$ time for any constant $\varepsilon > 0$, algorithms of Cheriyan *et al.* [3] run in $O(n^3/\log n)$ time and $O(nm + (n \log n)^2)$ time with high probability, and an algorithm of Ahuja *et al.* [1] runs in $O(nm \log(n/(m\sqrt{U}) + 2))$ time.

Prior to the push–relabel method, several studies have shown that Dinitz's algorithm [11] is in practice superior to other methods, including the network simplex method [7], [8], Ford–Fulkerson algorithm [12], [13], Karzanov's algorithm [21], and Tarjan's algorithm [24]. See, e.g., [19]. Several recent studies (e.g., [2], [9], [10], and [23]) show that the push–relabel method is superior to Dinitz's 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 implementa-

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

[2] Central Institute for Economics and Mathematics, Krasikova St. 32, 117418 Moscow, Russia. cher@cemi.msk.su.

[3] Computer Science Department, Stanford University, Stanford, CA 94305, USA. goldberg@cs.stanford.edu. Current address: NEC Research Institute Inc., 4 Independence Way, Princeton, NJ 08540, USA. avg@research.nj.nec.com.

Received October 24, 1994; revised April 25, 1995. Communicated by D. S. Johnson.

tions, we use both global relabeling and gap relabeling [5], [9] heuristics. As a result, one of our implementations is faster—on some problem families, asymptotically faster—than the previous implementations.[4]

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] and [23], 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 65,538 nodes compared with 1,081.3 seconds for their HL implementation. Performance of our implementations on such problems is as follows: F_PRF, 24.66 seconds; Q_PRF, 27.27 seconds; M_PRF, 335.13 seconds; H_PRF, 13.88 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's and push–relabel implementations grow quadratically. On DIMACS problem families we used in the other tests, the growth rate is smaller.

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. In Section 6 we discuss the behavior of gap relabeling under HL and FIFO selection rules. We present our conclusions in Section 7. The Appendix describes our hard problem generator.

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 [18]. (See also [16].) We present the two-phase variant of the method [17], 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 nonnegative 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,

[4] When we say that code A is asymptotically faster than code B on a certain problem family, we mean that the ratio of the B to A running times increases with the problem size.

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

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

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

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: $V \to \mathbf{N}$ 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 exceeds the potential flow value (e.g., sum of capacities of all arcs out of the source plus one). Initially $d(v)$ is the smaller of n and the distance from v to t in G_f. The method 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. A *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 *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 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;

```
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);
```

Fig. 2. The discharge operation.

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.

Note that when the first discharge operation is applied to s, all arcs out of s become saturated and the distance label of s is set to n.

The remaining issue is the order in which active nodes are processed. Two natural orders were suggested in [17] and [18]. 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 always to select for discharging a node with the highest label. In the worst case, the FIFO algorithm runs in $O(n^3)$ time [17], [18] and the highest-label algorithm runs in $O(n^2\sqrt{m})$ time [4].

The HL algorithm implementation maintains an array of sets B_i, $0 \le i \le n-1$, 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.

The WAVE implementation is "in-between" the FIFO and the HL implementations. Like the HL implementation, WAVE maintains the array of sets B_i, and makes passes through the array as follows. The pass starts with b equal to the highest i such that B_i is not empty. Nodes in B_b are discharged until B_i is empty, and then b is decreased. When i becomes negative, the pass terminates. The WAVE implementation runs in $O(n^3)$ time [5], [18].

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. We experimented with several ways of implementing the second stage.

Our earlier implementation [6] is based on flow decomposition (see e.g., [16]) and works as follows. Define the *flow graph* $G^f = (V, e^f)$ where $E^f = \{a \in E : f(a) > 0\}$. While there are nodes in $V - \{s, t\}$ with a positive excess, we pick a node v from this set and search the reversal of the flow graph in a depth-first search manner starting from

v. The search discovers either a cycle Γ or a simple path P from v to s. In the former case we decrease flow on the reversal of Γ by $\delta = \min\{f(a)|a \in \Gamma'\}$. In the latter case we decrease flow on the reversal of P and by $\delta = \min\{f(a)|a \in P'\}$.

Although under this implementation the second stage usually takes significantly less time than the first stage, on some problems (e.g., Washington-RLG-Long problems) the second stage exhibits large variations in performance and sometimes takes several times more than the first stage.

An alternative implementation of the second stage is to run the first stage "backward" [14]. Again, under this implementation the second stage usually takes significantly less time than the first stage, but on some problems (e.g., Acyclic-Dense), the second stage takes several times more than the first stage.

Our current implementation of the second stage is similar to the flow-decomposition-based implementation described above. We run a depth-first search in the reversal of the flow graph from the set of nodes with positive excess other than the source and the sink. If the depth-first search discovers a cycle, the flow on the cycle is reduced until one of the cycle arcs has a zero flow, and the depth-first search is restarted. The depth-first search produces a topological ordering of the nodes reachable from the nodes with positive excess. (Note that because we eliminate flow cycles during the search, the flow graph induced by these reachable nodes is acyclic.) We process these nodes in topological order. When a node is processed, flow into it is reduced until the excess at the node becomes zero.

Under this implementation, the second stage takes at most twice the time of the first stage on all problem instances in our experiments. The second stage takes longer than the first stage only on some problems in Washington-Line-Moderate family. On the vast majority of instances we tried, the running time of the second stage is a small fraction of the running time of 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 backward breadth-first search, which is computationally expensive compared with 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 [5] and by Derigs and Meier [9], and based on the following observation. Let g be an integer and $0 < g < n$. Suppose at a 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 40 MHZ 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 [20]. 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 were used at the First DIMACS Challenge [20]. These families are produced by three generators available from DIMACS. The first generator is RMFGEN of Goldfarb and Grigoriadis [19], 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 (described in the Appendix). This generator produces problem instances that are hard for the push–relabel and Dinitz's 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 a 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 $\mathbf{a} = 2^{x/4}$ and $\mathbf{b} = 2^{x/2}$.
- **Genrmf-Wide.** A network with $n = 2^x$ nodes in this family is generated by the `genrmf.c` program with parameters $\mathbf{a} = 2^{2x/5}$ and $\mathbf{b} = 2^{x/5}$.

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

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

- **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 which takes only one parameter, k.

4.3. *Implementations Evaluated.* We experimented with several variants of the push–relabel method, but we report on 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 WAVE, highest excess selection, last-in, first-out selection, and various hybrid strategies. In particular, the WAVE implementation showed reasonable performance similar to that of the FIFO implementation. Overall performance of these strategies was worse than that of the H_PRF code, however, and we do not report the results. 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 compared them with a previous implementation of the push–relabel method and with an implementation of Dinitz's algorithm [11].

The former implementation, developed by Anderson and Setubal [2] (ASF), implements the FIFO push–relabel algorithm using the global relabeling heuristic only; global relabelings are performed after every $m/2$ relabelings. The ASF implements the same algorithm as our F_PRF, except the global update frequency is different. We use this code as a "sanity check" for our implementation and to facilitate the comparison of our data to the data reported in [2]. (As observed in [23] and confirmed by our data, the global update frequency used in ASF is too low for dense graphs.)

We developed our own implementation of Dinitz's algorithm (DF). This implementation is written in the same programming style as our PRF implementations. Our implementation of Dinitz's algorithm seems to perform better than that of [2] on the basis of indirect comparison. We also compared our implementation of Dinitz's algorithm with

that of Goldfarb and Grigoriadis [19] (compiled with the `f77` compiler using the `-O` optimization option). Our implementation was faster by a factor of 1.5 or more on a subset of problem instances we tried.

When tabulating results of our experiments, we give the running times, the number of relabelings, and the number of pushes. To obtain a data point for a code, we make five runs of the code on problems produced with the same generator parameters but different pseudorandom generator seeds.[5] 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.

The running time is the user CPU time in seconds and excludes the input and output times. The number of relabelings is in 100's, rounded to the nearest integer. Similarly, the number of pushes is in 100's, rounded to the nearest integer.

We plot the data in addition to tabulating it. Our plots use logarithmic scales. To improve the 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 the 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, 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 a not very useful but relatively harmless addition for the FIFO algorithm.

The theoretical motivation of the HL selection strategy is to reduce the number of pushes. Operation counts for H_PRF and Q_PRF show that the former code usually makes significantly fewer pushes, and this often seems to be the main reason why H_PRF is faster than Q_PRF.

The ASF code implements the same FIFO algorithm as F_PRF but applies global relabeling after every $m/2$ relabelings (versus n for F_PRF). This and the low level implementation details account for the fact that 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. On dense networks, ASF makes too few global relabelings and performs asymptotically worse than F_PRF.

Our implementation DF of Dinitz's 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 [23] on

[5] Except for the AK generator, which is deterministic.

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 4 gives data for the Genrmf-Wide problem family. On this family, M_PRF and H_PRF performance is very close.

5.2. *Genrmf-Long Family.* Figure 5 gives data for the Genrmf-Long problem family. On this family H_PRF is somewhat faster than M_PRF. DF is asymptotically slower than the other codes.

5.3. *Washington-RLG-Wide Family.* Figure 6 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.

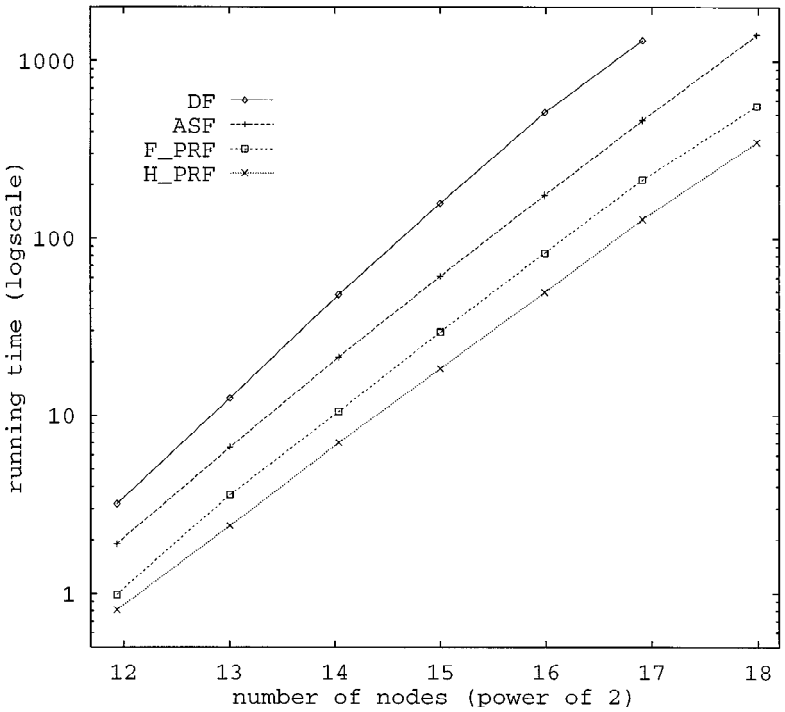
5.4. *Washington-RLG-Long Family.* Figure 7 gives data for the Washington-RLG-Long problem family. Here H_PRF performs better than M_PRF. M_PRF is slower than the FIFO codes. On this family the HL codes have better asymptotic performance than the FIFO codes. DF is asymptotically slower than the other codes.

5.5. *Washington-Line-Moderate Family.* Figure 8 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 9 gives data for the Acyclic-Dense problem family. On this family, H_PRF is somewhat faster than M_PRF. DF performs about as well as F_PRF on this family. ASF is asymptotically slower than the other codes.

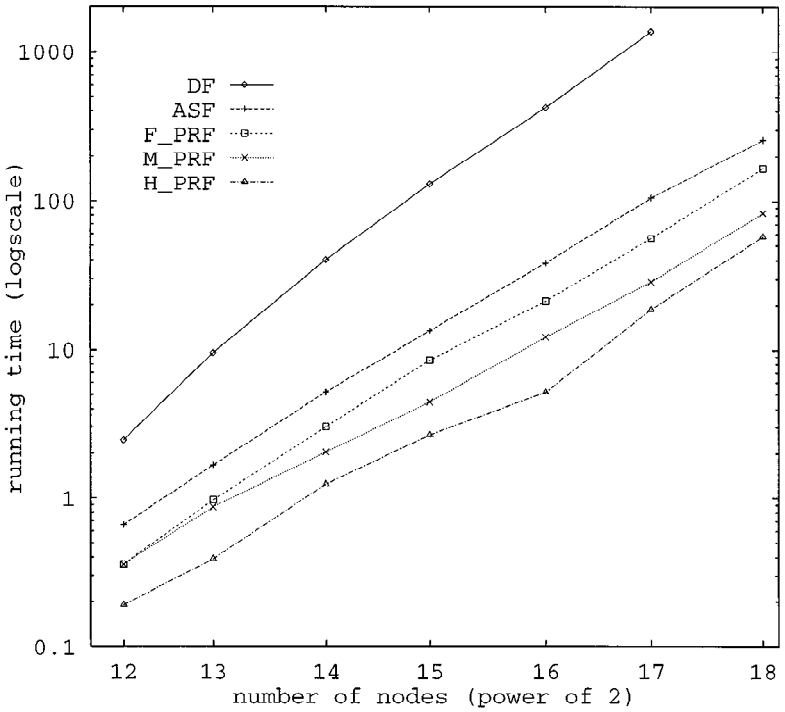
5.7. *AK Family.* Figure 10 gives data for the AK problem family. On this family all codes exhibit a roughly quadratic growth rate. However, the fastest code, H_PRF, is an order of magnitude faster than the slowest code, DF. This problem family is designed so that gap relabeling does not help. M_PRF is almost as fast as H_PRF. Our FIFO codes do the same number of relabelings as our HL codes. The FIFO codes, however, do almost twice the number of pushes the HL codes do, and as a result the FIFO codes are somewhat slower.

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.



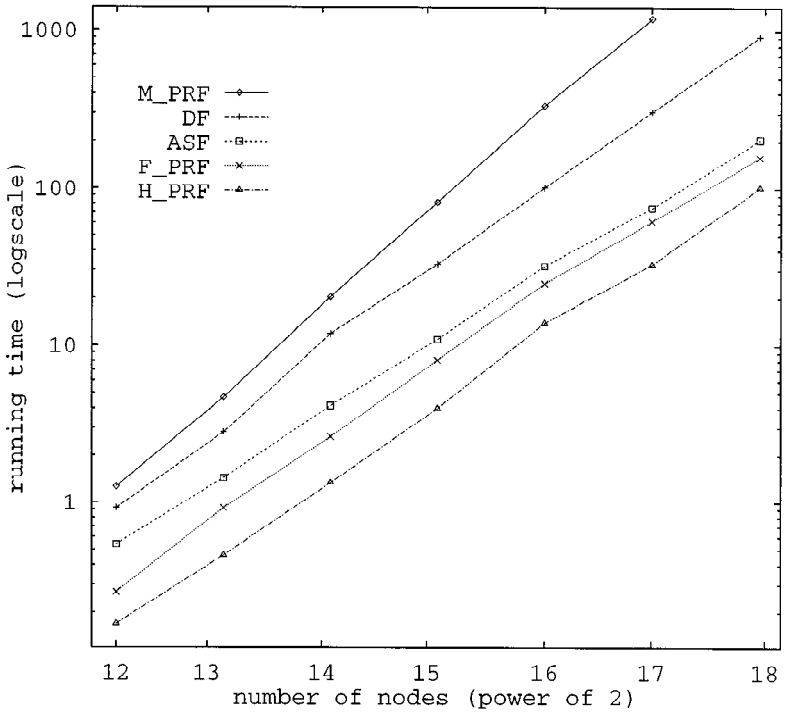
Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
3,920	18,256	**3.19**	**1.90**	**0.99**	**0.97**	**0.81**	**0.81**
				482	465	437	437
				1,179	1,151	903	903
8,214	38,813	**12.48**	**6.62**	**3.58**	**3.98**	**2.35**	**2.39**
				1,373	1,344	1,137	1,127
				3,461	3,393	2,349	2,329
16,807	80,262	**48.06**	**21.31**	**10.53**	**11.09**	**7.09**	**7.03**
				3,353	3,293	2,987	2,987
				8,257	8,146	6,281	6,281
32,768	157,696	**157.86**	**61.18**	**29.52**	**32.01**	**18.96**	**18.35**
				8,133	7,978	6,497	6,432
				19,059	18,812	12,124	11,993
65,025	314,840	**511.72**	**175.63**	**82.79**	**87.51**	**50.67**	**49.50**
				20,817	20,691	17,071	16,943
				50,700	41,595	33,955	33,716
123,210	599,289	**1,310.17**	**464.10**	**214.83**	**243.65**	**129.53**	**128.12**
				47,930	47,857	39,957	39,957
				110,755	110,642	73,878	73,878
259,308	1,267,875		**1,406.00**	**558.10**	**617.96**	**353.69**	**349.76**
				123,754	124,076	109,633	109,633
				289,760	290,711	196,742	196,742

Fig. 4. Genrmf-Wide family data.



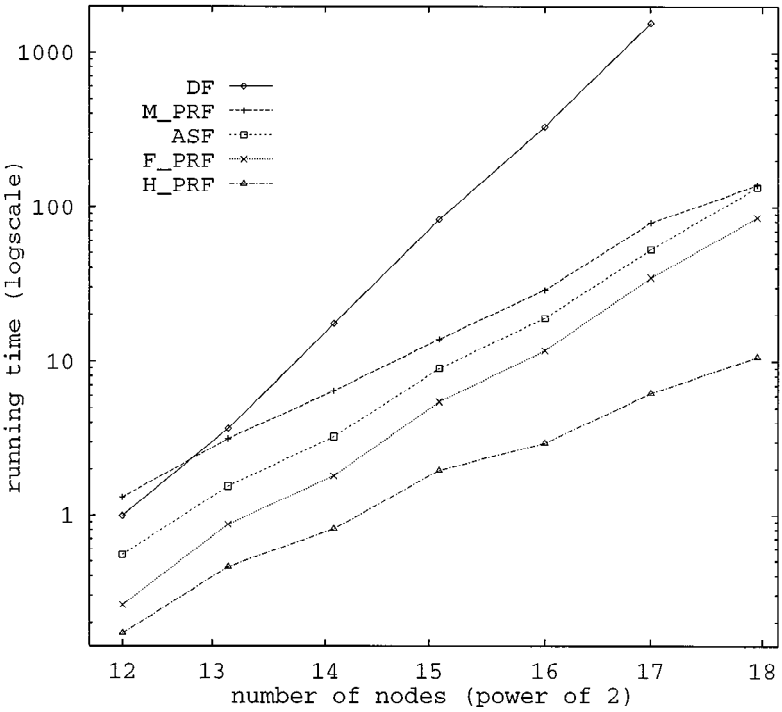
Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
4,096	18,368	**2.47**	**0.66**	**0.36**	**0.34**	**0.36**	**0.19**
				172	161	240	106
				476	458	394	204
7,371	33,498	**9.54**	**1.67**	**0.97**	**0.94**	**0.87**	**0.39**
				376	356	551	209
				1,103	1,072	894	395
15,448	71,687	**40.20**	**5.18**	**3.02**	**3.09**	**2.04**	**1.24**
				997	994	1,134	607
				3,022	3,013	1,893	1,062
30,589	143,364	**129.83**	**13.41**	**8.50**	**8.79**	**4.43**	**2.67**
				2,400	2,278	2,314	1,229
				7,507	7,329	3,723	2,108
65,536	311,040	**422.86**	**38.28**	**21.26**	**22.76**	**12.18**	**5.17**
				5,218	4,986	6,303	2,201
				17,694	17,234	9,391	3,863
130,682	625,537	**1,360.41**	**104.74**	**56.08**	**61.93**	**28.61**	**18.68**
				12,831	12,375	14,975	9,315
				41,661	41,022	21,294	13,512
270,848	1,306,607		**258.01**	**165.92**	**173.30**	**82.80**	**57.62**
				34,076	33,420	42,838	29,107
				118,067	116,994	61,021	40,787

Fig. 5. Genrmf-Long family data.



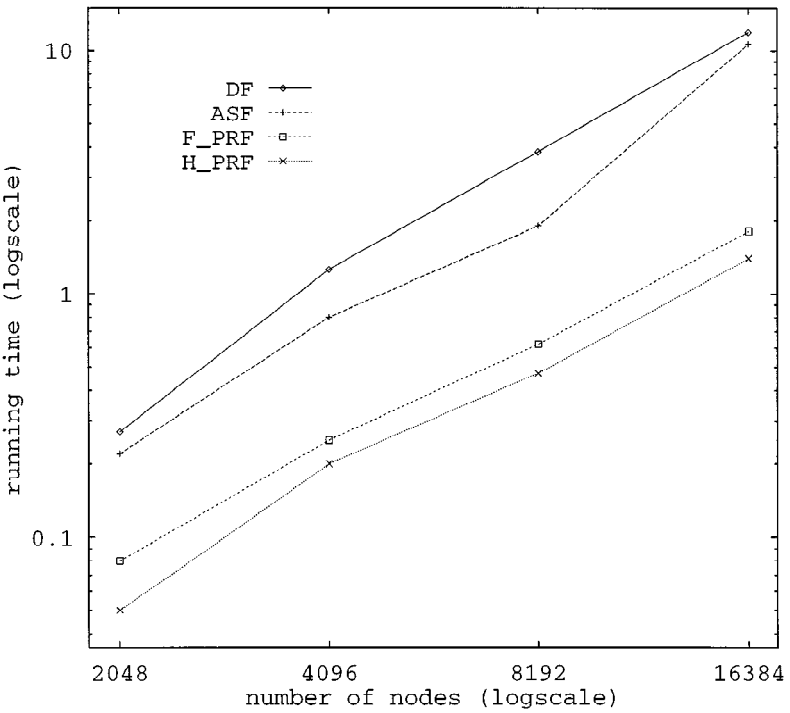
Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
4,098	12,224	**0.92**	**0.54**	**0.27**	**0.26**	**1.26**	**0.17**
				140	121	1,185	92
				572	537	1,972	347
8,194	24,418	**2.80**	**1.43**	**0.92**	**0.73**	**4.69**	**0.46**
				342	310	4,197	238
				1,317	1,261	6,442	830
16,386	48,896	**11.88**	**4.16**	**2.62**	**2.82**	**20.33**	**1.33**
				943	883	15,708	665
				3,348	3,231	23,752	2,138
32,770	97,772	**32.77**	**10.97**	**8.11**	**8.68**	**81.78**	**3.99**
				2,400	2,278	2,314	1,229
				7,507	7,329	3,723	2,108
65,538	195,584	**101.38**	**31.86**	**24.66**	**27.27**	**335.13**	**13.88**
				5,900	5,769	239,246	5,962
				18,920	18,708	344,717	16,089
131,074	391,168	**306.73**	**75.33**	**61.45**	**67.04**	**1,185.51**	**32.61**
				12,584	12,027	762,145	10,619
				40,716	39,501	1,068,044	29,560
262,146	782,336	**916.51**	**205.23**	**158.55**	**176.96**		**101.82**
				29,188	29,316		31,121
				91,972	92,289		81,046

Fig. 6. Washington-RLG-Wide family data.



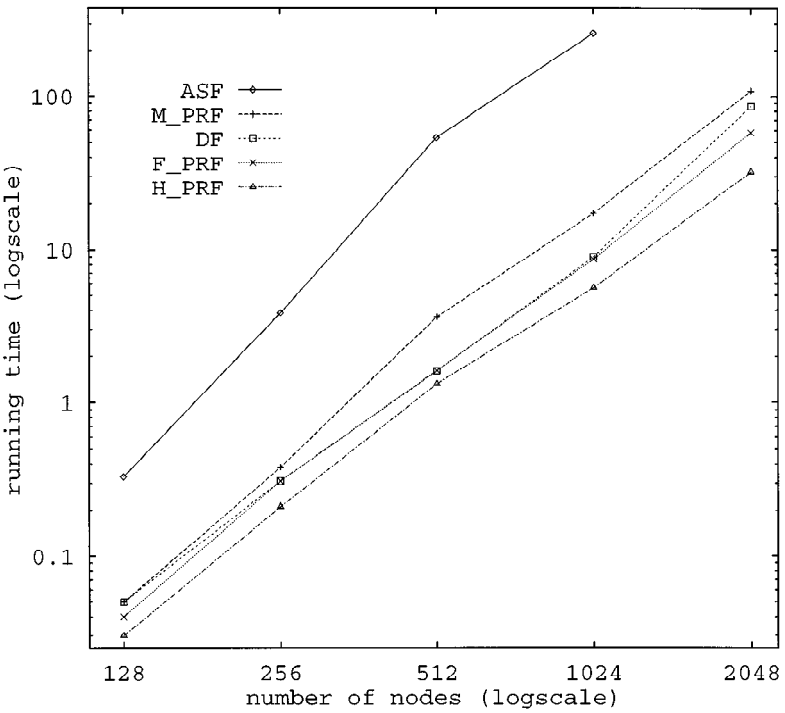
Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
4,098	12,224	**0.99**	**0.55**	**0.26**	**0.27**	**1.31**	**0.17**
				140	121	1,185	92
				572	537	1,972	347
8,194	24,512	**3.66**	**1.54**	**0.87**	**0.74**	**3.14**	**0.46**
				312	285	2,697	256
				1,348	1,305	4,363	847
16,386	49,088	**17.40**	**3.24**	**1.80**	**1.85**	**6.44**	**0.81**
				568	474	5,085	375
				2,844	2,669	8,126	1,342
32,770	98,240	**82.52**	**8.97**	**5.47**	**5.91**	**13.84**	**1.96**
				1,454	1,301	10,443	830
				7,605	7,329	17,248	3,005
65,538	196,544	**330.62**	**18.92**	**11.68**	**12.69**	**28.84**	**2.94**
				2,446	2,236	21,543	1,121
				15,912	15,564	33,419	4,355
131,074	391,168	**1,562.85**	**52.80**	**34.59**	**38.37**	**78.54**	**6.21**
				6,312	5,970	57,764	2,451
				42,584	41,942	90,098	9,139
262,146	786,368		**134.30**	**84.95**	**93.19**	**139.25**	**10.67**
				12,373	11,428	101,468	3,640
				98,419	96,698	159,696	15,250

Fig. 7. Washington-RLG-Long family data.



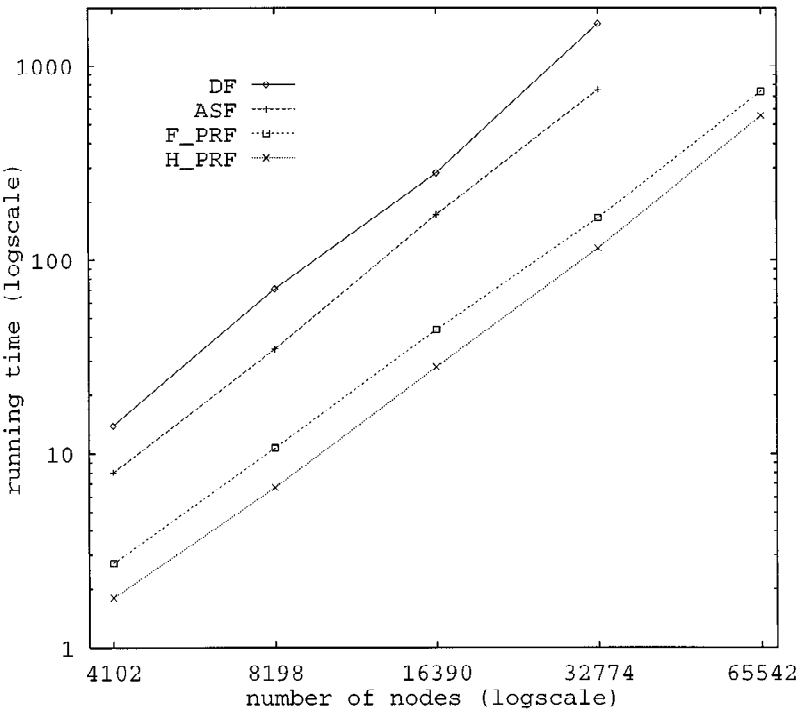
Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
2,050	22,300	**0.27**	**0.22**	**0.08**	**0.06**	**0.04**	**0.05**
				11	9	7	5
				66	53	48	44
4,098	65,000	**1.26**	**0.80**	**0.25**	**0.24**	**0.22**	**0.20**
				26	16	20	7
				142	130	109	90
8,194	187,400	**3.84**	**1.90**	**0.62**	**0.59**	**0.52**	**0.47**
				38	27	24	10
				285	244	181	163
16,386	522,200	**11.91**	**10.63**	**1.81**	**1.66**	**1.60**	**1.40**
				97	52	76	18
				557	501	404	325

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



Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
128	8,128	**0.05**	**0.33**	**0.04**	**0.04**	**0.05**	**0.03**
				4	3	6	2
				9	9	13	8
256	32,640	**0.31**	**3.83**	**0.31**	**0.24**	**0.38**	**0.21**
				9	8	24	7
				27	26	36	21
512	130,816	**1.60**	**53.71**	**1.61**	**1.52**	**3.61**	**1.32**
				19	17	47	17
				60	56	119	49
1,024	523,776	**8.95**	**258.79**	**8.65**	**8.31**	**17.24**	**5.60**
				44	41	95	33
				139	134	271	102
2,048	2,096,128	**86.13**		**58.04**	**53.83**	**108.61**	**32.06**
				140	129	412	88
				386	370	723	261

Fig. 9. Acyclic-Dense family data.



Nodes	Arcs	DF	ASF	F_PRF	Q_PRF	M_PRF	H_PRF
4,102	6,151	**13.90**	**7.97**	**2.72**	**2.77**	**1.85**	**1.80**
				657	657	657	657
				11,838	11,838	6,585	6,585
8,198	12,265	**71.00**	**34.50**	**10.70**	**10.73**	**6.68**	**6.70**
				1,947	1,947	1,947	1,947
				45,919	45,919	24,926	24,926
16,390	24,583	**281.98**	**172.15**	**43.47**	**42.62**	**29.03**	**27.88**
				5,385	5,385	5,385	5,385
				178,710	178,710	94,783	94,783
32,774	49,159	**1,651.90**	**753.52**	**165.87**	**164.87**	**122.70**	**115.35**
				15,098	15,098	15,098	15,098
				701,606	701,606	365,979	365,979
65,542	98,311			**740.08**	**758.78**	**558.48**	**555.77**
				43,965	43,965	43,965	43,965
				2,772,947	2,772,947	1,430,605	1,430,605

Fig. 10. AK family data.

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.

Now suppose a gap arises during an execution of the F_PRF implementation (which does not use gap relabeling). Compared with 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 is *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 four-thirds 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 $\frac{4}{3}$.

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_PRF, was always faster than our implementation of Dinitz's 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 experimental results suggest 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.

Problem families that are bad for the H_PRF code and not as bad for the F_PRF code can be designed. This fact, combined with the reasonable performance of the F_PRF code in our study, makes the code a natural candidate to consider when H_PRF does not perform well. F_PRF is also better suited for parallel and distributed implementation, and it is simpler than H_PRF.

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

Q_PRF performance is similar to (but overall slightly worse than) F_PRF 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's 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.

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

Appendix. The literature contains several problem families which are hard for push–relabel algorithms [2], [4] in the sense that the algorithms' running time is close to their worst-case bounds. These families, however, are not hard if the global update heuristic is used. Below we describe the problem family generated by our generator AK. For this family, $m = O(n)$. The FIFO, HL, and WAVE versions of the push–relabel method take $\Omega(n^2)$ time on problems in this family even if global and gap relabeling operations are used, under the assumption that the initial distance labeling gives exact distances to the sink. This assumption holds for most implementations of the push–relabel method. Although for the push–relabel algorithms without update heuristics the AK networks are not as hard as those described in [2] and [4], the AK networks are harder if the update heuristics are used. Dinitz's algorithm also takes $\Omega(n^2)$ time on AK networks.

The $AK(k)$ network consists of two subnetworks, $N1(k)$ and $N2(k)$, connected in parallel. $N1(k)$ is hard for the HL and the FIFO implementations and $N2(k)$ is hard for the WAVE implementation. Both $N1(k)$ and $N2(k)$ are hard for Dinitz's algorithm.

Let k be the parameter that determines the network size. $N1(k)$ consists of two paths, upper and lower, containing k nodes each. (See Figure 11.) Let $u_1, \ldots, u_k$ and $\ell_1, \ldots, \ell_k$ be the upper and the lower path nodes, respectively. All arcs of the lower path have a capacity of $k + 1$. Capacities of the upper path arcs start at k and decrease by one at every step; thus the capacity of (u_i, u_{i+1}) is $k - i + 1$. Also, each node of the upper path is connected to the first node ℓ_1 of the lower path by a unit capacity arc. There are two more nodes, s_1 and t_1, in addition to the path nodes, which we call the source and the sink of $N1(k)$. There are arcs (s_1, u_1) of capacity $k + 1$, (s_1, ℓ_1) of capacity 1, (u_k, t_1) of capacity 1, and (ℓ_k, t_1) of capacity $k + 1$.

$N2(k)$ consists of a path with $2k+2$ nodes, $x_0, x_1, \ldots, x_{2k+1}$. (See Figure 12.) As one goes along the path, the capacities of the arcs first decrease by one at every step, reaching

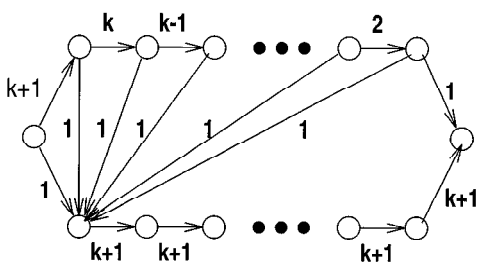


Fig. 11. Subnetwork $N1(k)$.

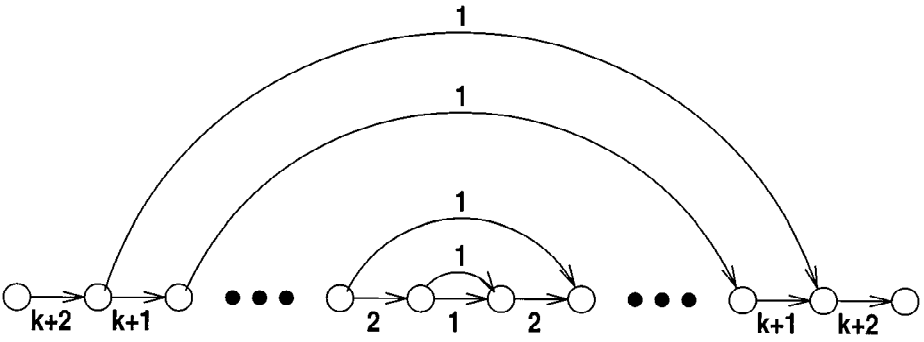


Fig. 12. Subnetwork $N2(k)$.

one at the middle arc (x_k, x_{k+1}), and then increase by one at every step. (Arcs (x_0, x_1) and (x_{2k}, x_{2k+1}) have a capacity of $k+2$.) In addition, for each $i = 1, 2, \ldots, k+1$, there is an arc (x_i, x_{2k+1-i}) with unit capacity. We call the first and the last node of the path the source and the sink of $N2(k)$, respectively.

$AK(k)$ contains the source and the sink in addition to the two subnetworks. The source node is connected to the source of $N1(k)$ and $N2(k)$ by arcs with very large capacities. The sinks of $N1(k)$ and $N2(k)$ are connected to the sink node by arcs with very large capacities.

Consider an execution of the HL algorithm on the first subnetwork. Recall that we assume that the initial distance labeling gives exact distances to the sink. A *phase i* is the period from the time the arc (u_i, u_{i+1}) is saturated for the first time to the time the arc (u_{i+1}, u_{i+2}) is first saturated. (Note that the upper path arcs are first saturated in the order determined by the path.) We show that the number of push operations during phase i is at least $i+1$, even with global and gap relabelings. This implies that the total work is $\Omega(k^2)$.

It can be shown by induction on i that the following sequence of events takes place. Just before the beginning of phase i, u_i has $k-i+1$ units of excess. Distance labels of nodes $u_1, \ldots, u_{i-1}$ are equal to $d(\ell_1)+1 = k+1$ and distance labels of other nodes are unchanged. The node u_i is discharged, saturating the arc (u_i, u_{i+1}) of capacity $k-i+1$ and starting the phase. The discharge also increases $d(u_i)$ to $d(\ell_1)+1$ and pushes a unit of flow to ℓ_1. This flow unit must move to ℓ_{i+1} before the next phase can start. The number of pushes required to move the unit of excess from u_i to ℓ_{i+1} is at least $i+1$. Note that until the arc (ℓ_1, ℓ_2) is saturated, distance labels of nodes in $N2(k)$ are exact. Thus the HL algorithm takes $\Omega(k^2)$ time with or without global and gap relabelings.

Next consider an execution of the FIFO algorithm on $N1(k)$ (starting from the time s_1 is first discharged). We consider the case when s_1 pushes first to u_1 and then to ℓ_1; the other case is similar. It can be shown by induction on p that after $2p$ passes, distance labels of nodes $u_1, \ldots, u_{2p}$ are equal to $d(\ell_1)+1 = k+1$ and distance labels of other nodes of $N1(k)$ are unchanged. There are $k-2p+1$ units of excess at u_{2p+1}, 1 unit of excess at ℓ_1, 2 units of excess at $\ell_3, \ell_5, \ldots, \ell_{2p+1}$, and u_{2p+1} appears on the queue before ℓ_1. The number of active nodes at pass p for $1 \le p \le k$ is $\Omega(p)$ and therefore

the algorithm takes $\Omega(k^2)$ time for these passes only. The arc (ℓ_1, ℓ_2) becomes saturated only at pass k, and until that the distance labels of nodes in the lower path are exact. Thus the FIFO algorithm on $N1(k)$ takes $\Omega(k^2)$ time with or without global and gap relabelings.

Dinitz's algorithm on $N1(k)$ goes through $k+1$ blocking flow phases. Phase zero saturates arcs (s_1, ℓ_1) and (u_k, t_1). For $1 \le i \le k$, phase i saturates the arc (u_i, ℓ_1). Thus Dinitz's algorithm also takes $\Omega(k^2)$ time. The WAVE algorithm, however, runs in linear time on $N1(k)$. We show that it takes $\Omega(k^2)$ time on $N2(k)$.

Consider an execution of the WAVE algorithm on $N2(k)$. At the first pass, the algorithm first discharges x_0, pushing $k+2$ flow units to x_i, then discharges x_1, saturating the arc (x_1, x_{2k}), relabeling x_1, and saturating (x_1, x_2). The rest of the pass moves the unit of flow from x_{2k} to the sink of $N2(k)$. For $i = 2, \ldots, k$, the ith pass first discharges x_{2i-1}, saturating $(x_{2i-1}, x_{2(k-i)+2})$, relabels x_{2i-1}, and pushes the remaining excess to x_{2i}, saturating (x_{2i-1}, x_{2i}). The rest of the pass moves the unit of flow just pushed to $x_{2(k-i)+2}$ to the sink of $N2(k)$. Note that distance labels of all nodes with excess considered during the execution are exact, so global and gap relabelings do not help. It is easy to see that pass i takes $\Omega(i)$ time, so the total time is $\Omega(k^2)$.

The above results imply that the HL, FIFO, and WAVE algorithms take $\Omega(k^2)$ time on the $AK(k)$ network with or without global and gap relabelings, and Dinitz's algorithm also takes $\Omega(k^2)$ time.

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, Providence, RI, 1993.

[3] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a Maximum Flow be Computed in $o(nm)$ Time? In *Proc. ICALP*, 1990.

[4] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Network Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[5] 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 Translations, Vol. 158, pp. 23–30. AMS, Providence, RI, 1994.

[6] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, Stanford, CA, 1994.

[7] 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.

[8] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1962.

[9] 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.

[10] U. Derigs and W. Meier. *An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow*. NATO ASI Series on Computer and System Sciences, vol. 8, pages 209–223. Nijhoff, The Hague, 1992.

[11] E. A. Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[12] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1972.

[13] L. R. Ford, Jr., and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[14] A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1985.

[15] A. V. Goldberg. Efficient Graph Algorithms for Sequential and Parallel Computers. PhD thesis, M.I.T., Cambridge, MA, January 1987. (Also available as Technical Report TR-374, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1987).

[16] 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, Berlin, 1990.

[17] 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.

[18] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[19] D. Goldfarb and M. D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. *Ann. of Oper. Res.*, 13:83–123, 1988.

[20] D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, Providence, RI, 1993.

[21] A. V. Karzanov. Determining the Maximal Flow in a Network by the Method of Preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.

[22] 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.

[23] 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, Providence, RI, 1993.

[24] R. E. Tarjan. A Simple Version of Karzanov's Blocking Flow Algorithm. *Oper. Res. Lett.*, 2:265–268, 1984.