The Partial Augment–Relabel Algorithm for the Maximum Flow Problem

Andrew V. Goldberg

Microsoft Research – Silicon Valley, 1065 La Avenida, Mountain View, CA 94062
goldberg@microsoft.com

Abstract. The maximum flow problem is a classical optimization problem with many applications. For a long time, HI-PR, an efficient implementation of the highest-label push-relabel algorithm, has been a benchmark due to its robust performance. We propose another variant of the push-relabel method, the partial augment-relabel (PAR) algorithm. Our experiments show that PAR is very robust. It outperforms HI-PR on all problem families tested, asymptotically in some cases.

1 Introduction

The maximum flow problem and its dual, the minimum cut problem, are classical combinatorial optimization problems with applications in many areas of science and engineering. For this reason, the problem has been studied both from theoretical and practical viewpoints for over half a century. The problem is to find a maximum flow from the source to the sink (a minimum cut between the source and the sink) given a network with arc capacities, the source, and the sink. Below we denote the number of vertices and arcs in the input network by n and m, respectively. Theoretical line of research led to the development of augmenting path [16], network simplex [12], blocking flow [14], and push-relabel [20] methods. The best currently known time bounds appear in [25,19].

From the practical point of view, good implementations of Dinic's blocking flow method [10,21] proved superior to the network simplex and the augmenting path algorithms. The blocking flow method remained the method of choice until the development of the push-relabel method [20], which was quickly recognized as practical. Within a year of its invention, the method had been used in a physics application [28]. A parallel implementation of the method, including some speed-up heuristics applicable in the sequential context as well, has been studied in [17]. Implementations of variants of the method has been studied during the First DIMACS Implementation Challenge [23,2,27]. This work showed that using the global update and gap heuristics (discussed in detail in Section 3), one gets an implementation superior to the efficient implementations of Dinic's algorithm. Another conclusion was that, except in degenerate cases, the dynamic tree data structure, used in the most theoretically efficient algorithms, does not help in practice, where its overhead exceeds the corresponding performance gains [4]. The Challenge also led to the development of the DIMACS problem families.

Subsequent work of Cherkassky and Goldberg [11] showed that, with a proper choice of data structures, the combination of global update and gap heuristics is more robust than the individual heuristics, and the high-level selection rule outperforms other popular selection rules. This implementation, PRF, and its later version, HI-PR, has been used for over a decade in many applications. Several subsequent implementations, such as [26], use the combination of gap and global update and differ by low-level data structures or initialization strategies.

A number of attempts have been made to develop an implementation that is more robust than HI-PR. The ideas behind the binary blocking flow algorithm in particular appear practical, and Hagerup et al. [22] show that this algorithm outperforms Dinic' blocking flow algorithm. However, although the algorithm has a better worst-case time bound and performs better on specific bad instances, all attempts so far to produce an implementation of this algorithm that is more robust than HI-PR failed.

Some maximum flow algorithms developed for specific applications outperform HI-PR in these applications. For example, minimum cuts are being used extensively in vision applications. Boykov and Kolmogorov [5] developed an algorithm that is superior to the highest level and FIFO push-relabel implementations on many vision problems. Their implementation is extensively used by the vision community. See [5,6] for surveys of the vision applications.

In this paper we describe a new *partial augment-relabel (PAR)* variant of the push-relabel method. In our comparison of PAR to HI-PR, the former code is consistently faster, in some cases asymptotically so. The push-relabel algorithm implementations had significant impact on applications and experimental work in the area for over a decade. In this context, the superior performance of PAR is significant. Furthermore, the push-relabel algorithm is the basis for efficient implementations of algorithms for the minimum-cost flow [18], global minimum cut [8], and parametric flow [3] problems. The ideas presented in this paper apply to these problems, and may improve performance of the corresponding implementations.

2   Definitions and Notation

Input to the maximum flow problem is (G, s, t, u), where $G = (V, A)$ is a directed graph, $s, t \in V$, $s \neq t$ are the *source* and the *sink*, respectively, $u : A \Rightarrow [1, \ldots, U]$ is the *capacity function*, and U is the *maximum capacity*.

Let a^R denote the *reverse* of an arc a. let A^R be the set of all reverse arcs, and let $A' = A \cup A^R$. Note that we add a reverse arc for every arc of A; in particular, if A contains both $a = (v, w)$ and $b = (w, v)$, then a^R is an arc parallel to but different from b. A function g on A' is *anti-symmetric* if $g(a) = -g(a^R)$. Extend u to be an anti-symmetric function on A', i.e., $u(a^R) = -u(a)$.

A flow f is an anti-symmetric function on A' that satisfies *capacity constraints* on all arcs and *conservation constraints* at all vertices except s and t. The capacity constraint for $a \in A$ is $0 \leq f(a) \leq u(a)$ and for $a \in A^R$ it is $-u(a^R) \leq f(a) \leq 0$. The conservation constraint for v is $\sum_{(u,v) \in A} f(u, v) = \sum_{(v,w) \in A} f(v, w)$. The *flow value* is the total flow into the sink: $|f| = \sum_{(v,t) \in A} f(v, t)$.

A *cut* is a partitioning of vertices $S \cup T = V$ with $s \in S, t \in T$. Capacity of a cut is defined by $u(S,T) = \sum_{v \in S, w \in T, (v,w) \in A} u(v,w)$.

A *preflow* is a relaxation of a flow that satisfies capacity constraints and a relaxed version of conservation constraints $\sum_{(u,v) \in A} f(u,v) \geq \sum_{(v,w) \in A} f(v,w)$. We define flow *excess* of v by $e_f(v) = \sum_{(u,v) \in A} f(u,v) - \sum_{(v,w) \in A} f(v,w)$. For a preflow f, $e_f(v) \geq 0$ for all $v \in V - \{s,t\}$.

Residual capacity is defined by $u_f(a) = u(a) - f(a)$ for $a \in A$ and $u_f(a) = f(a^R)$ for $a \in A^R$. Note that if f satisfies capacity constraints, then u_f is non-negative. The *residual graph* $G_f = (V, A_f)$ is induced by the arcs in A' with strictly positive residual capacity. An *augmenting path* is an s–t path in G_f.

A distance labeling is an integral function d on V that satisfies $d(t) = 0$. Given a preflow f, we say that d is valid if for all $(v,w) \in A_f$ we have $d(v) \leq d(w) + 1$. Unless mentioned otherwise, we assume that a distance labeling is valid with respect to the current preflow in the graph. We say that an arc (v,w) is *admissible* if $(v,w) \in A_f$ and $d(v) = d(w) + 1$, and denote the set of admissible arcs by A_d.

3   The Push-Relabel Method

The push-relabel method [20][1] maintains a preflow and a distance labeling, which are modified using two *basic operations*:

Push(v,w) applies if $e_f(v) > 0$ and $(v,w) \in A_d$. The operation chooses $\delta : 0 < \delta \leq \min(u_f(v,w), e_f(v))$, increases $f(v,w)$ and $e_f(w)$ by δ and decreases $e_f(v)$ and $f((v,w)^R)$ by δ. A push is *saturating* if after the push $u_f(v,w) = 0$ and *non-saturating* otherwise.

Relabel(v) applies if $d(v) < n$ and v has no outgoing admissible arcs. A relabel operation increases $d(v)$ to the maximum value allowed by the distance labeling constraints: $1 + \min_{(v,w) \in A_f} d(w)$ or to n if v has no outgoing residual arcs.

The method can start with any feasible preflow and distance labeling. Unless mentioned otherwise, we assume the following simple initialization: f is zero on all arcs except for arcs out of s, for which the flow is equal to the capacity; $d(s) = n$, $d(t) = 0$, $d(v) = 1$ for all $v \neq s, t$. For a particular application, one may be able to improve algorithm performance using an application-specific initialization. After initialization, the method applies push and relabel operations until no operation applies.

When no operation applies, the set of all vertices v such that t is reachable from v in G_f defines a minimum cut, and the excess at the sink is equal to the maximum flow value. For applications that need only the cut, the algorithm can terminate at this point. For applications that need the maximum flow, we run the second stage of the algorithm. One way to implement the second stage is to first reduce flow around flow cycles to make the flow acyclic, and then to return flow excesses to the source by reducing arc flows in the reverse topological order

---

[1] Sometimes it is referred to as preflow-push, which is misleading: e.g., Karzanov's implementation [24] of the blocking flow method uses preflows and the push operation.

with respect to this acyclic graph. See [29]. Both in theory and in practice, the first stage of the algorithm dominates the running time.

The *current arc* data structure [20] is important for algorithm efficiency; it works as follows. Each vertex maintains a current arc $a(v)$. Initially, and after each relabeling of v, the arc is the first arc on $v's$ arc list. When we examine $a(v)$, we check if it is admissible. If not, we advance $a(v)$ to the next arc on the list. The definition of basic operations implies that only relabeling v can create new admissible arcs out of v. Thus as $a(v)$ advances, arcs behind it on the list are not admissible. When the arc advances past the end of the list, v has no outgoing admissible arcs and therefore can be relabeled. Thus the current arc data structure allows us to charge to the next relabel operation the searches for admissible arcs to apply push operations to. We say that a vertex $v \neq s, t$ is *active* if $d(v) < n$ and $e_f > 0$. The highest-label variant of the method, which at each step selects an active vertex with the highest distance label, runs in $O(n^2\sqrt{m})$ time [9,30].

HI-PR Implementation. Next we review the HI-PR implementation [11] of the push-relabel algorithm. It uses the highest-label selection rule, and global update and gap heuristics.

To facilitate implementation, the method uses a *layers of buckets* data structure. The layers correspond to distance labels i. Each layer i contains two buckets, *active* and *inactive*. A vertex v with $d(v) = i$ is in one of these buckets: in the former if $e_f(v) > 0$ and in the latter otherwise. Active buckets are maintained as singly linked lists and support insert and extract-first operations. Inactive buckets are maintained as doubly linked lists and support insert and delete operations. The layer data structure is an array of records, each containing two pointers – to the active and the inactive buckets of the layer. A pointer is `null` if the corresponding bucket is empty. To implement the highest-level selection, we maintain the index of the highest layer with non-empty active bucket.

The gap heuristic [13] is based on the following observation. Suppose for $0 < i < n$, no vertex has a distance label of i but some vertices w have distance labels $j : i < j < n$. The validity of d implies that such w's cannot reach t in G_f and can therefore be deleted from the graph until the end of the first phase of the algorithm.

Layers facilitate implementation of the gap heuristic. We maintain the invariant that the sequence of non-empty layers (which must start with layer zero containing t) has no gaps. Note that only the relabeling operation moves vertices between layers. During relabeling, we check if a gap is created, i.e., if increasing distance label of v from its current value $d(v)$ makes both buckets in layer $d(v)$ empty. If this is the case, we delete v and all vertices at the higher layers from the graph, restoring the invariant. Note that deleting a vertex takes constant time, and the total cost of the gap heuristic can be amortized over the relabel operations.

Push and relabel operations are local. On some problem classes, the algorithm substantially benefits from the *global relabeling* operation. This operation performs backwards breadth-first search from the sink in G_f, computing exact

distances to the sink and placing vertices into appropriate layer buckets. Vertices that cannot reach the sink are deleted from the graph until the end of the first phase. Global update places the remaining vertices in the appropriate buckets and resets their current arcs to the corresponding first arcs. HI-PR performs global updates after $\Omega(m)$ work has been done by the algorithm; this allows amortization of global updates.

HI-PR implements the highest-label algorithm, which runs in $O(n^2\sqrt{m})$ time. As the work of heuristics is amortized, HI-PR runs in the same time bound.

4   PAR Algorithm

First we describe a well-known *augment-relabel (AR)* variation of the push-relabel method.[2] The AR algorithm maintains a flow (not a preflow), and augments along the shortest augmenting path. However, instead of breadth-first search, it uses the relabel operation to find the paths. The method maintains a valid distance labeling. Any initial labeling can be used. To find the next augmenting path, the method starts at s and searches the admissible graph in the depth-first search manner. At a general step, the algorithm has an admissible path from s to the current vertex v and tries to extend it. If v has an admissible arc (v, w), the path is extended to w. (The admissible arcs can be efficiently found using the current arc data structure.) Otherwise the method shrinks the path, making the predecessor of v on the path the current vertex, and relabels v. An augmenting path is found if t becomes the current vertex. The algorithm augments the flow along the path and starts a new search. Note that the new search can start at s or at the last vertex of the augmenting path reachable from s in G_f after the augmentation.

We experimented with the AR algorithm and its preflow variation (which picks a vertex with excess, finds an admissible path to t, and pushes as much flow as possible on arcs of the path, possibly creating flow excesses at intermediate vertices). These algorithms performed very poorly. Next we discuss a related algorithm that performs well.

The partial augment-relabel (PAR) algorithm is a push-relabel algorithm that maintains a preflow and a distance labeling. The algorithm has a parameter k. At each step, the algorithm picks an active vertex v and attempts to find an admissible path of k arcs starting at v. If successful, the algorithm executes k push operations along the path, pushing as much flow as possible. Otherwise, the algorithm relabels v.

PAR looks for augmenting paths in the same way as AR. It maintains a current vertex x (initially v) with an admissible path from v to x. To extend the path, the algorithm uses the current arc data structure to find an admissible arc (x, y). If such an arc exists, the algorithm extends the path and makes y the current vertex. Otherwise the algorithm shrinks the path and relabels x. The

---

[2] [1] refers to this algorithm as the shortest augmenting path algorithm. However, this is only one of the methods that augment along the shortest paths. Methods of [14,15] are also shortest augmenting path algorithms.

search terminates if $x = t$, or the length of the path reaches k, or v is the current vertex and v has no outgoing admissible arcs.

As in the push-relabel method, we have the freedom to choose the next active vertex to process. We use the highest-label selection rule. One can show that, for $k = O(\sqrt{m})$, PAR has the same $O(n^2\sqrt{m})$ bound as HI-PR.

Implementation details. Our PAR implementation is similar to that of HI-PR. In particular, we use layers and highest-label selection. The gap heuristic is identical to that used in HI-PR. After experimenting with different values of k we used $k = 4$ in all of our experiments. The best value of k is problem-dependent, but we have seen only modest improvements compared to $k = 4$. Results for $3 \leq k \leq 6$ would have been similar. We concentrate on the differences in addition to the most obvious one, the use of partial augment strategy.

Note that HI-PR relabels only active vertices currently being processed, and as a side-effect we can maintain active vertices in a singly-linked list. PAR can relabel other vertices as well, and we may have to move an active vertex in the middle of a list into a higher-level list. Therefore PAR uses doubly-linked lists for active as well as inactive vertices. List manipulation becomes slower, but the overall effect is very minor. No additional space is required as the inactive list is doubly-linked in both implementations and a vertex is in at most one list at any time.

We also make improvements to global relabeling, which could also be applied to HI-PR. These include (i) incremental restart, (ii) early termination, and (iii) adaptive amortization.

Incremental restart takes advantage of the fact that if, since the previous global update, flows from vertices at distance D or less have not changed, we can start the update from layer D as lower layers are already in breadth-first order. This change can be implemented very efficiently as the only additional information we need is the value of D, which starts at n after each global update, and is updated to $min(d(w), D)$ each time we push flow to a vertex w. Incremental restart has little cost but can save substantial amount of work, especially in combination with the highest-label vertex selection.

The early termination heuristic stops breadth-first search when all vertices active immediately before the global update have been placed in their respective layers by the search. More precisely, let L be the highest distance label of an active vertex at this point. We stop breadth-first search after scanning all vertices in the new layer L. For vertices with distance labels of L and below that have not been scanned by breadth-first search, we set their distance labels to $L + 1$ and insert them into layer $L + 1$. Vertices with distance labels above $L + 1$ remain in their previous layers. In general, the early termination heuristic may stop before the breadth-first search is completed and make global updates less effective. However, in our experiments the work saved by early termination seem to outweigh the potential increase in other operations.

Note that global update with an incremental restart and early termination produces a valid distance labeling and the distance labels cannot decrease during an update. Thus the algorithm remains correct, and the running time bound does not get any worse.

With incremental restart and early termination, global updates sometimes cost substantially less than the time to do breadth-first search of the whole graph. We experimented with various amortization strategies based on a threshold, which is set based on the work done by the previous global update and the number n' of vertices left in the graph. When the work since the last global update exceeds the threshold, we do the next update. The code used in our experiments uses the number of relabel operation to measure the work and sets the threshold T as follows: $T = F\left(\frac{n}{100} + n'4^{S/n'}\right)$, where S is the number of vertices scanned during the last global update and F is the global update frequency parameter. The intuition is to do the next update earlier if the last one was cheap, but to limit the variation in the threshold value by a factor of 4.

Our experiments use a fixed value $F = 1.0$ and $k = 4$. One can improve performance by experimentally tuning these parameters to a a specific applications.

5   Experimental Results

We test code performance on DIMACS problem families [23] (see also [21]) and on problems from vision applications.[3] As we will see, many DIMACS problems are very simple and algorithm performance is sometimes sensitive to low-level details. To make sure that the performance is not affected by the order in which the input arcs are listed, we do the following. First, we re-number vertex IDs at random. Next, we sort arcs by the vertex IDs. The vision problems have been made available at `http://vision.csd.uwo.ca/maxflow-data/` by the vision group at the University of Western Ontario, and include instances from stereo vision, image segmentation, and multiview reconstruction.

The main goal of our experiments is to compare PAR with HI-PR. We use the latest version, 3.6, of HI-PR and the current version, 0.23, of PAR. We also make a comparison to an implementation of Chandran and Hochbaum [7]. A paper describing this implementation is listed on authors' web sites as "submitted for publication" and no preprint is publicly available. The authors do make their code available, and gave several talks claiming that the code performs extremely well. These claims were about version 3.1 of their code, which we refer to as CH. Recently, an improved version, 3.21, replaced the old version on the web site. We compare to this version, denoted as CHn (n for new), as well. We are not aware of any public document that describes either of these codes.

Our experiments were conducted on an HP Evo D530 machine with 3.6 MGz Pentium 4 processor, 28 KB level 1 and 2 MB level 2 cache, and 2GB of RAM. The machine was running Fedora 7 Linux. C codes HI-PR, PAR, and CHn were compiled with the gcc version 4.1.2 compiler that came with the Fedora distribution using "-O4" optimization option. C++ code CH was compiled with the g++ version 4.1.2 compiler using "-O4" optimization.

For synthetic problems, we report averages over ten instances for each problem size. We give running time in seconds. In some tables we also give scan count

---

[3]   Due to space restrictions we omit problem descriptions. See the references.

Table 1. RMF-Long (left) and RMF-Wide (right) problem families

n	0.3 M	0.5 M	1.1 M	2.0 M	4.1 M	0.3 M	0.5 M	1.0 M	2.1 M	4.2 M
m	1.3 M	2.6 M	5.1 M	10.2 M	20.3 M	1.3 M	2.6 M	5.2 M	10.3 M	20.7 M
HI-PR time	0.67	1.75	4.26	12.63	41.88	6.45	15.05	41.55	100.12	266.99
sd%	8.04	15.50	10.71	11.41	18.35	2.99	4.48	3.55	3.71	12.06
PAR time	0.32	0.65	1.35	2.79	5.89	3.76	9.29	21.67	50.90	128.61
sd%	5.88	8.69	7.18	7.12	8.78	5.88	8.69	7.18	7.13	8.78
HI-PR sc/n	8.16	11.25	13.25	20.04	33.97	52.93	60.90	76.55	88.64	102.43
PAR sc/n	4.99	5.02	5.05	5.16	5.26	47.16	54.64	62.06	70.24	80.63
CH time	0.74	1.79	4.68	9.21	24.67	8.53	17.30	72.61	476.50	1200.07
sd%	11.90	10.32	4.68	7.57	10.80	76.84	71.17	87.88	47.10	36.83
CHn time	0.59	1.36	3.28	7.43	16.73	2.49	6.31	13.56	85.05	430.42
sd%	18.92	15.34	12.51	14.17	8.69	9.56	18.65	17.88	113.28	90.64

Table 2. Wash-Wide problem family

n	0.13 M	0.26 M	0.52 M	1.0 M	2.1 M	4.2 M	8.4 M
m	0.4 M	0.8 M	1.6 M	3.1 M	6.3 M	12.5 M	25.0 M
HI-PR time	0.80	2.36	7.61	22.10	51.81	133.37	273.51
PAR time	0.47	1.53	4.33	12.71	29.48	77.40	160.17
HI-PR sc/n	16.20	19.99	25.51	30.46	32.32	39.92	39.22
PAR sc/n	9.01	12.26	14.21	18.55	19.66	25.86	25.27
CH time	0.39	1.29	3.23	8.20	18.91	47.05	118.41
CHn time	0.35	1.09	2.78	6.85	15.43	36.32	74.18

per vertex (sc/n), where the scan count (sc) is the sum of the number of relabel operations and the number of vertices scanned by the global update operations. This gives a machine-independent measure of performance. In the tables, $k = 10^3$ and $M = 10^6$. In Table 1, we also give standard deviation in percent ($sd\%$).

Another experiment we ran, but omit due to the lack of space, measured how much improvement is due to the improved global updates. These usually improve performance, but by less than 20%. Most of the improvement compared to HI-PR is due to the PAR basic operation ordering.

5.1 Experiments with DIMACS Families

PAR vs. HI-PR. First we compare PAR to HI-PR on DIMACS families. On RMF-Long family (Table 1 left), the new code gives an asymptotic improvement, and its running time is almost linear: As the problem size increases by a factor of 16, the number of scans per vertex shows a minor increase from 4.99 to 5.26. On RMF-Wide family (Table 1 right), PAR is faster, by about a factor of two for larger problems.

On Washington problem families (Tables 2–3), PAR outperforms HI-PR, usually by a little less than a factor of two. Note that Wash-Long and Wash-Line problems are very easy, with larger problems requiring less then two scans per vertex.

Table 3. Wash-Long (left) and Wash-Line (right) problem families

n	0.5 M	1.0 M	2.1 M	4.2 M	8.4 M	41 K	66 K	104 K	165 K	262 K
m	1.6 M	3.1 M	6.3 M	12.6 M	25.2 M	2.1 M	4.2 M	8.4 M	16.8 M	33.5 M
HI-PR time	0.82	1.60	3.05	5.59	10.82	0.24	0.46	1.06	2.26	4.50
PAR time	0.51	0.98	1.81	3.46	6.66	0.15	0.28	0.64	1.40	3.02
HI-PR sc/n	3.98	3.62	3.26	2.68	2.55	2.07	2.06	2.05	2.05	2.04
PAR sc/n	2.49	2.19	1.83	1.66	1.56	1.13	1.11	1.10	1.09	1.07

Table 4. Acyclic-Dense problem family

n	2.0 K	2.9 K	4.1 K	5.8 K	8.2 K
m	2.1 M	4.2 M	8.4	16.8 M	33.6 M
HI-PR time	0.93	1.93	4.60	10.63	27.08
PAR time	0.13	0.25	0.51	1.02	2.05
HI-PR sc/n	8.81	9.12	10.50	11.25	12.39
PAR sc/n	1.87	1.81	1.93	1.91	1.93

The Acyclic-Dense problem family (Table 4) is also very easy for PAR, which performs asymptotically better than HI-PR. This problem family is very sensitive to initialization. A slightly different initial labeling can cause significant performance degradation.

The above experiments show that on DIMACS problem families, PAR outperforms HI-PR, asymptotically on RMF-Long and Acyclic-Dense families.

Comparison with CH. Next we compare HI-PR and PAR with CH and CHn. We exclude easy problem families where for large problem sizes PAR performs less than three scans per vertex. The data appear in Tables 1 and 2.

On RMF-Long and RMF-Wide families, PAR is asymptotically the fastest code. On the RMF-Wide problems, CH and CHn exhibit very high variance in the running time: Table 1 shows this for all sizes for CH and for the larger sizes for CHn. For the largest size, the ratio between the fastest and the slowest runs of CHn is 14.3, compared to 1.35 for PAR. This is an indication that CH and CHn are not robust. On Wash-Wide problems, CHn is the fastest code. PAR is slower by about a factor of two.

5.2   Vision Instances

Stereo vision problems (see Table 5) have several independent subproblems: tsukuba has 16, sawtooth – 20, venus – 22. The BVZ and KZ2 prefixes refer to different ways of defining the problem. As suggested in [5], we report the total time for each problem. Here CHn is the fastest code. PAR is the second fastest, losing by less than a factor of two. HI-PR loses to PAR by about a factor of two. CH performs poorly, losing by three orders of magnitude in some cases.

On multiview instances, (Table 6[4]) PAR is about a factor of two faster than HI-PR. CH crashes on these problems. CHn fails feasibility and optimality selfchecks on the smaller problems and fails to allocate the memory it needs for the larger problems.

---

[4]   *dnf* stands for "did not finish."

Table 5. Stereo vision data

name	HI-PR	PAR	CH	CHn
BVZ–				
tsukuba	8.07	4.01	643.62	2.80
sawtooth	12.45	7.70	3,127.23	5.45
venus	23.65	11.79	2,707.32	7.23
KZ2–				
tsukuba	30.39	13.31	4,020.49	6.85
sawtooth	31.88	16.35	13,472.85	11.70
venus	61.64	24.68	12,898.89	15.84

Table 7. Segmentation data

name	HI-PR	PAR	CH	CHn
bone-xyzx-6-10	1.06	0.37	dnf	0.20
bone-xyzx-6-100	1.08	0.39	dnf	0.25
bone-xyz-6-10	2.41	0.86	dnf	0.50
bone-xyz-6-100	2.48	0.93	dnf	0.58
bone-xyzx-26-10	2.89	1.04	dnf	0.60
bone-xyzx-26-100	3.17	1.07	dnf	0.58
bone-xy-6-10	6.65	2.04	dnf	1.19
bone-xy-6-100	6.92	2.16	dnf	1.36
bone-xyz-26-10	6.38	2.25	dnf	1.20
bone-xyz-26-100	6.72	2.44	dnf	1.39
liver-6-10	34.59	16.90	dnf	14.18
liver-6-100	46.69	18.76	dnf	17.59
babyface-6-10	52.75	26.42	dnf	22.77
babyface-6-100	71.55	34.36	dnf	37.85

Table 6. Multiview data

name	HI-PR	PAR	CH	CHn
gargoyle-sml	4.37	2.68	dnf	dnf
camel-sml	8.54	4.56	dnf	dnf
gargoyle-med	125.20	53.04	dnf	dnf
camel-med	160.27	76.77	dnf	dnf

Data in Table 7 shows that on segmentation instances, CHn is the fastest for smaller problem sizes, but PAR loses by less than a factor of two. On larger problems, CHn and PAR perform similarly. HI-PR is two to three times slower than PAR. CH crashes on these problems.

6   Concluding Remarks

The push-relabel method allows endless variations. Over the years we tried many potentially promising ideas, but until now our attempts failed to produce an implementation more robust than HI-PR. Robustness is the main reason why HI-PR remained the benchmark for a long time. Our experiments give a strong evidence that PAR is a more robust code. It outperforms HI-PR on all problem families we used, and in some cases the improvement in performance is asymptotic. Simplicity is another reason for the popularity of HI-PR, and PAR is not much more complicated.

CH is not a very robust code. It is an order of magnitude slower that HI-PR on large RMF-Wide problems and over two orders of magnitude slower on stereo vision problems (the only vision problems it has not crashed on). Compared to PAR, CH is never significantly faster, and often significantly slower – by over three orders of magnitude on the KZ2-sawtooth instance, for example. Our experience with this code contradicts the claims made by its authors. CHn is more robust than CH. However it is noticeably slower on large RMF-Wide instances. Furthermore, it failed to run on the multiview instances due to higher memory overhead and insufficient precision. Fixing these problems will probably make CHn slower as it may require higher precision.

Graphs in vision problems are very regular and can be represented more compactly, as done in [5]. It would be interesting to implement PAR with such a representation of a graph. This would make PAR more efficient on the vision problems and would enable a direct comparison to the algorithm of [5]. The results of [5] also suggest that FIFO outperforms the highest-label selection on vision problems, so it would be interesting to study a FIFO version of PAR.

Our experiments indicate that the DIMACS data set is showing its age. Most problems are easy for PAR, requiring less than ten scans per vertex. Wash-Wide problems are somewhat harder. Only RMF-Wide problems show clear asymptotic increase in the number of scans per vertex, but even for these problems the rate of increase is small. The largest problems with millions of vertices are solved using under a hundred scans per vertex. Although some DIMACS problems are still useful, there is a need for other synthetic and real-life data sets. The vision problems made available by the UWO group at our request is a step in this direction.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Englewood Cliffs (1993)
2. Anderson, R.J., Setubal, J.C.: Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In: Johnson, D.S., McGeoch, C.C. (eds.) Network Flows and Matching: First DIMACS Implementation Challenge, pp. 1–18. AMS (1993)
3. Babenko, M.A., Goldberg, A.V.: Experimental Evaluation of a Parametric Flow Algorithm. Technical Report MSR-TR-2006-77, Microsoft Research (2006)
4. Badics, T., Boros, E.: Implementing a Maximum Flow Algorithm: Experiments with Dynamic Trees. In: Johnson, D.S., McGeoch, C.C. (eds.) Network Flows and Matching: First DIMACS Implementation Challenge, pp. 65–96. AMS (1993)
5. Boykov, Y., Kolmogorov, V.: An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision. IEEE transactions on Pattern Analysis and Machine Intelligence 26(9), 1124–1137 (2004)
6. Boykov, Y., Veksler, O.: Graph Cuts in Vision and Graphics: Theories and Applications. In: Paragios, N., Chen, Y., Faugeras, O. (eds.) Handbook of Mathematical Models in Computer Vision, pp. 109–131. Springer, Heidelberg (2006)
7. Chandran, B., Hochbaum, D.: A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem (submitted, 2007)
8. Chekuri, C.S., Goldberg, A.V., Karger, D.R., Levine, M.S., Stein, C.: Experimental Study of Minimum Cut Algorithms. In: Proc. 8th ACM-SIAM Symposium on Discrete Algorithms, pp. 324–333 (1997)
9. Cheriyan, J., Maheshwari, S.N.: Analysis of Preflow Push Algorithms for Maximum Network Flow. SIAM J. Comput. 18, 1057–1086 (1989)

10. Cherkassky, B.V.: A Fast Algorithm for Computing Maximum Flow in a Network. In: Karzanov, A.V. (ed.) Collected Papers. Combinatorial Methods for Flow Problems, vol. 3, pp. 90–96. The Institute for Systems Studies, Moscow (1979) (in russian); English translation appears in AMS Trans., vol.158, pp. 23–30 (1994)
11. Cherkassky, B.V., Goldberg, A.V.: On Implementing Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19, 390–410 (1997)
12. Dantzig, G.B.: Application of the Simplex Method to a Transportation Problem. In: Koopmans, T.C. (ed.) Activity Analysis and Production and Allocation, pp. 359–373. Wiley, New York (1951)
13. Derigs, U., Meier, W.: 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 vol. 8, pp. 209–223. NATO (1992)
14. Dinic, E.A.: Metod porazryadnogo sokrashcheniya nevyazok i transportnye zadachi. In: Issledovaniya po Diskretnoĭ Matematike, Nauka, Moskva (1973) (in russian); Title translation: Excess Scaling and Transportation Problems
15. Edmonds, J., Karp, R.M.: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. J. Assoc. Comput. Mach. 19, 248–264 (1972)
16. Ford, L.R., Fulkerson, D.R.: Maximal Flow Through a Network. Canadian Journal of Math. 8, 399–404 (1956)
17. Goldberg, A.V.: Efficient Graph Algorithms for Sequential and Parallel Computers. PhD thesis, M.I.T., January 1987. Technical Report TR-374, Lab. for Computer Science, M.I.T (1987)
18. Goldberg, A.V.: An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. J. Algorithms 22, 1–29 (1997)
19. Goldberg, A.V., Rao, S.: Beyond the Flow Decomposition Barrier. J. Assoc. Comput. Mach. 45, 753–782 (1998)
20. Goldberg, A.V., Tarjan, R.E.: A New Approach to the Maximum Flow Problem. J. Assoc. Comput. Mach. 35, 921–940 (1988)
21. Goldfarb, D., Grigoriadis, M.D.: A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. Annals of Oper. Res. 13, 83–123 (1988)
22. Hagerup, T., Sanders, P., Träff, J.L̃.: An implementation of the binary blocking flow algorithm. Algorithm Engineering, 143–154 (1998)
23. Johnson, D.S., McGeoch, C.C.: Network Flows and Matching: First DIMACS Implementation Challenge. In: AMS. Proceedings of the 1-st DIMACS Implementation Challenge (1993)
24. Karzanov, A.V.: Determining the Maximal Flow in a Network by the Method of Preflows. Soviet Math. Dok. 15, 434–437 (1974)
25. King, V., Rao, S., Tarjan, R.: A Faster Deterministic Maximum Flow Algorithm. J. Algorithms 17, 447–474 (1994)
26. Mehlhorn, K., Naher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge (1999)
27. Nguyen, Q.C., Venkateswaran, V.: Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In: Johnson, D.S., McGeoch, C.C. (eds.) Network Flows and Matching: First DIMACS Implementation Challenge, pp. 19–42. AMS (1993)
28. Ogielski, A.T.: Integer Optimization and Zero-Temperature Fixed Point in Ising Random-Field Systems. Phys. Rev. Lett. 57, 1251–1254 (1986)
29. Sleator, D.D., Tarjan, R.E.: A Data Structure for Dynamic Trees. J. Comput. System Sci. 26, 362–391 (1983)
30. Tuncel, L.: On the Complexity of Preflow-Push Algorithms for Maximum-Flow Problems. Algorithmica 11, 353–359 (1994)