Two-Level Push-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. We describe a two-level push-relabel algorithm for the maximum flow problem and compare it to the competing codes. The algorithm generalizes a practical algorithm for bipartite flows. Experiments show that the algorithm performs well on several problem families.

1 Introduction

The maximum flow problem is classical combinatorial optimization problem 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. We denote the number of vertices and arcs in the input network by n and m, respectively. Some time bounds also depend on the maximum arc capacity U. When U appears in the bound, we assume that the capacities are integral.

A theoretical line of research led to development of augmenting path, network simplex, blocking flow, and push-relabel methods and to a sequence of improved bounds; see [17] for a survey. Under the assumption $\log U = O(\log n)$ [12], the bound of $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U)$ is achieved by the binary blocking flow algorithm [15]. The best strongly polynomial bound of $O(nm \log_{m/(n \log n)} n)$ is achieved by the algorithm of [21].

From the practical point of view, good implementations of Dinitz blocking flow method [6,19] 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 [16]. For a long time, the HI-PR implementation [7] of the highest-level push-relabel method served as a benchmark for maximum flow algorithms. This implementation uses both the global update (see e.g. [13]) and gap [9] heuristics.

Mazzoni et al. [22] introduced a number of variations of the push-relabel method, including the partial augment-relabel method, which on the basis of limited experiments they claimed to be computationally superior. A variant of the method has been further studied in [24]. Recently, the author [18] developed an efficient implementation, PAR, of the algorithm and conducted more extensive experiments confirming that it outperforms HI-PR.

A. Goldberg and Y. Zhou (Eds.): AAIM 2009, LNCS 5564, pp. 212–225, 2009.

The partial augment-relabel algorithm moves flow along paths of k arcs. In the efficient implementations the best value of k is less then ten but greater than two. However, the idea of looking ahead or moving flow two steps has been used for several problems closely related to the maximum flow problem. In the context of the minimum-cost flow, [13] suggests an algorithm that pushes flow into a vertex that can push flow further without being relabeled. In the context of the assignment problem, an operation that moves a flow excess two levels at a time appears in [14]. For bipartite graphs, maximum flow algorithms that move flow excess two steps at a time and leave no excess on one side of the graph achieve better theoretical bounds [1] and practical performance [23].

The main contribution of this paper is a maximum flow algorithm that pushes flow on two adjacent levels of the network. The resulting algorithm is different from the partial augment-relabel algorithm with $k = 2$ and can be viewed as a generalization of the bipartite graph algorithm of [1] to general graphs. We present an efficient implementation, P2R, of this algorithm. Another contribution is an experimental evaluation of the algorithms, which compares P2R with several other codes, including a version of PAR that is an improved implementation of the algorithm used in [18]. The experiments show that P2R is comparable to PAR.

Our work makes progress towards unifying previous work on practical maximum flow algorithms and getting a better understanding of what works best in practice. The push-relabel method leads to practical algorithms for related problems, such as minimum-cost flows [13], assignment problem [14], bipartite matching [8] and parametric flows [2]. It is possible that some of the techniques discussed in this paper will lead to improved algorithms for some of the related problems as well.

2 Definitions and Notation

The 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 $A' = A \cup A^R$. 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 $s \in S, t \in T$. The capacity of a cut is defined by $u(S, T) = \sum_{v\in S, w\in T, (v,w)\in A} u(S, T)$. The max-flow, min-cut theorem [11] says that the maximum flow value is equal to the minimum cut capacity.

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 the 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 $v \in V \setminus \{s,t\}$.

The *residual capacity* of an arc $a \in A'$ is defined by $u_f(a) = u(a) - f(a)$. Note that if f satisfies capacity constraints, then u_f is non-negative. The *residual graph* $G_f = (V, A_f)$ is the graph 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 $(v,w) \in E_f$ 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 $d(w) < d(v)$, and denote the set of admissible arcs by A_d.

3 The Push-Relabel Method

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

Push(v,w) applies if $e_f(v) > 0$ and $(v,w) \in A_d$. It 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: $1 + \min_{(v,w)\in A_f} d(w)$ 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 is applicable.

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 with respect to this acyclic graph. See [25]. Both in theory and in practice, the first stage of the algorithm dominates the running time.

The *current arc* data structure is important for algorithm efficiency. 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.

3.1 HI-PR Implementation

Next we review the HI-PR implementation [7] of the push-relabel algorithm. It uses highest-label selection rule, and global update and gap heuristics. We say that a vertex $v \neq s, t$ is *active* if $d(v) < n$ and $e_f > 0$.

The method uses a *layers of buckets* data structure. Layers correspond to distance labels. 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 a singly linked list 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. We refer to the active and inactive buckets of layer i by α_i and β_i, respectively.

To implement the highest-label selection, we maintain the index μ of the highest layer with non-empty active bucket. The index increases if an active vertex is inserted into a layer higher than the current value of μ, which can happen during a relabel operation.

At each step of the algorithm, we examine α_μ. If it is empty, we decrease μ or terminate if $\mu = 0$. Otherwise, let v be the first active vertex of α_μ. We look for an admissible arc out of v. If such an arc (v, w) is found, we push flow along this arc. In addition to changing f, the push can have two side-effects. First, $e_f(w)$ may change from zero to a positive value, making w active. We delete w from $\beta_{d(w)}$ and insert it into $\alpha_{d(w)}$. Second, $e_f(v)$ can decrease to zero, making it inactive. We extract v from the head of $\alpha_{d(v)}$ and insert it into $\beta_{d(v)}$. If no admissible arc out of v exists, we relabel v. This increases $d(v)$. We extract v from the head of $\alpha_{d'(v)}$, where $d'(v)$ is the old distance label of v, and insert it into $\alpha_{d(v)}$. In this case we also increase μ to $d(v)$. Then we proceed to the next step.

The gap heuristic [10] 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 the 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. 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, and set $\mu = d(v) - 1$. 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 $O(m)$ work has been done by the algorithm; this allows amortization of global updates.

The following improvements to the implementation of global relabeling have been proposed in [18]: (i) incremental restart, (ii) early termination, and (iii) adaptive amortization. Suppose flows on arcs at distance D or less have not change. The incremental restart takes advantage of this fact: 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. 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.

With incremental restart and early termination, global updates sometimes cost substantially less than the time to do breadth-first search of the whole graph, and an amortization strategy can be used to trigger a global update. Our new implementation of PAR uses a threshold that is different from that used in [18]. Every time we do a global update, we set the threshold T to $T = S + C$ where S is the number of vertices scanned during the global update and C is a constant that represents the cost of calling the global update routine. The next global update is performed when $WF > T$, where W is the number of vertex scans since the last global update and F is the global update frequency parameter. In our experiments we use $C = 500$ and $F = 0.2$.

Note that buckets have constant but non-trivial overhead. For instances where heuristics do not help, the buckets slow the code down by a constant factor. When the heuristics help, however, the improvement can be asymptotic.

3.2 PAR Implementation

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 vertices 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 depth-first manner. 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 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. Our PAR implementation uses layers and highest-level 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. Results for $2 \leq k \leq 6$ would have been similar.

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.

Our new implementation of PAR includes two optimizations. The first optimization is to use regular queue-based breadth-first search (instead of the incremental breadth-first search described in Section-3.1) for the first global update. This is because the first update usually looks at the majority of the vertices, and the queue-based implementation is faster. The second optimization is a more careful implementation of the augment operation that, when flow is pushed into and then out of a vertex, moves the vertex between active and inactive lists only if the vertex status changed from the time before the first push to the time after the second one. The effects of these optimizations are relatively minor, but noticeable on easy problems, which includes some practical vision instances and some of the DIMACS problems.

4 The P2R Algorithm

The *two-level push-relabel algorithm (P2R)* at each step picks a vertex u to process and applies the *two-level push* operation to it. Although u can be any active vertex, the implementation discussed in this paper uses the highest label selection. It uses the same data structures and heuristics as PAR.

The two-level push operation works as follows. Using the current arc data structure, we examine admissible arcs out of u. If (u, v) is such an arc, and v is the sink, we push flow on (u, v). If v is not a sink, we make sure v has an outgoing admissible arc, and relabel v if it does not.

When we find an admissible arc (u, v) such that v has an outgoing admissible arc, we consider two cases. If v is active, push the maximum possible amount from u to v, and then push flow on admissible arcs out of v until either v is no longer active or v has no outgoing admissible arcs. In the latter case, relabel v.

If v has no excess, we proceed in a way that avoids activating v, i.e., we do not push to v more flow than v can push out along its admissible arcs. To keep the same asymptotic complexity, we do it in such a way that the work can be amortized over distance label increases of v. First, we set $\delta = \min(u_f(u, v), e_f(u))$ and set $\Delta = \delta + e_f(v)$. Then we compute the amount C that v can push along its admissible arcs as follows. We start with $C = 0$ and examine arcs of v starting from the current arc. Each time we see and admissible arc (v, w), we add $u_f(v, w)$ to C. We stop either when we reach the end of the arc list of v or when $C \geq \Delta$.

In the former case, we push $C - e_f(v)$ units of flow on (u, v), push C units of flow out of v, and relabel v. Note that the push on (u, v) may move less than δ units of flow and therefore neither get rid of the excess at u nor saturate (u, v). However, we can charge the work of this push to the relabeling of v.

In the latter case, we push δ units of flow on (u, v) and then push δ units of flow out of v on admissible arcs starting from the current arc of v and advancing the current arc to the last arc used for pushing flow. Note that while doing so, we examine the same arcs as we did when computing C. Therefore the work involved in computing C is amortized over the current arc advances.

P2R does something different from a generic push-relabel algorithm in two places. First, a push from u to v may move less flow. Second, extra work is involved in computing C. As has been mentioned above, non-standard work can be amortized over other work done by the algorithm. Therefore generic push-relabel bounds apply to P2R. We also believe that the $O(n^2\sqrt{m})$ bound for the highest-label push-relabel algorithm [5,26] can be matched but omit details due to the lack of space.

Note that if we apply P2R to a bipartite graph, the algorithm will maintain the invariant that except in the middle of the two-level push operation, all excess is on one side of the network. In this sense, the algorithm generalizes the bipartite flow algorithm of [1].

5 Experimental Results

We test code performance on DIMACS problem families [20] (see also [19]) and on problems from vision applications.[1] We use RMF-Long, RMF-Wide, Wash-Long, Wash-Wide, Wash-Line, and Acyc-Dense problem families. 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/` and include instances from stereo vision, image segmentation, and multiview reconstruction.

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

The main goal of our experiments is to compare P2R to PAR and HI-PR. We use the latest version, 3.6, of HI-PR, version 0.43, of PAR and version 0.45 of P2R. We also make a comparison to an implementation of Chandran and Hochbaum [4]. 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 talks 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 also compare to this version, denoted as CH-n. Finally, for vision problem we compare to the code BK of Boykov and Kolmogorov [3]. As code is intended for vision applications and does not work well on the DIMACS problems, we restrict the experiments to the vision problems.

Our experiments were conducted on an HP Evo D530 machine with 3.6 HGz 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, P2R, BK, and CH-n were compiled with the gcc compiler version 4.1.2 using "-O4" optimization option. C++ code CH was compiled with the g++ compiler using "-O4" optimization.

For synthetic problems, we report averages over 10 instances for each problem size. In all tables and plots, we give running time in seconds. For our algorithms, we also give scan count per vertex, where the scan count 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.

5.1 Experiments with DIMACS Families

Figures 1 – 6 give performance data for the DIMACS families. We discuss the results below.

P2R vs. PAR vs. HI-PR. First we note that P2R and PAR outperform HI-PR, in some cases asymptotically so (e.g., RMF and Acyc-Dense families).

Compared to each other, P2R and PAR performance is very similar. The latter code is faster more often, but by a small amount. The biggest difference is on the Acyc-Dense problem family, where P2R is faster by roughly a factor of 1.5.

Note that for P2R and PAR, many DIMACS problem families are easy. For the biggest Wash-Long, Wash-Line, and Acyc-Dense problems, these codes perform less than two scans per vertex, and for RMF-Long – around six. Note that Acyc-Dense are highly structured graphs, and the fact that P2R does about 50% fewer operations than PAR is probably due to the problem structure and is not very significant.

The "wide" instances are harder, but not too hard. For Wash-Wide problems, the number of scans per vertex grows slowly with the problem size, and stops growing between the two largest problem sizes. Even for the largest problems with over eight million vertices, the number of scans per vertex is around 25. RMF-Wide problems are the hardest and the number of scans per vertex slowly

grows with the problem size, but even for the largest problem with over four million vertices, the number of scans per vertex is in the 70's.

Comparison with CH. On RMF-Long problem families, the CH codes are asymptotically slower, and CH-n is somewhat faster than CH. On the RMF-Wide family, CH is significantly slower than CH-n. Performance of both codes is asymptotically worse than that of P2R and PAR. In particular, for larger problem the CH codes exhibit very large variance from one instance to another, which is not the case for the push-relabel codes. CH is always slower than P2R and PAR, losing by an order of magnitude on the largest problem. CH-n is faster than P2R and PAR by about a factor of 1.3 on the smallest problem, but slower by about a factor of 3.4 on the largest one.

On Wash-Long problem family, P2R and PAR are faster than the CH codes, although the latter are within a factor of two. It looks like the CH codes are asymptotically slower, but the difference in the growth rates are small. On Wash-Wide problem family, CH and CH-n are the fastest codes, but P2R and PAR never lose by much more than a factor of two. On Wash-Line problems, the four codes are within a factor of two of each other, but while P2R and PAR show clear linear growth rate, CH and CH-n running times grow erratically with the problem size.

On Acyc-Dense problems, where P2R is about 1.5 times faster than PAR, CH and CH-n performance falls in the middle of the push-relabel codes: the CH codes are almost as fast as P2R for the smallest problem and about as fast as PAR for the largest one.

5.2 Vision Instances

Stereo vision problems three problem sequences: tsukuba has 16, sawtooth – 20, venus – 22 subproblems. As suggested in [3], we report the total time for each problem sequence (Table 1), but for operation counts we report the average over subprobelms for each sequence (Table 2). On these problems, BK is the fastest code by a large margin. PAR is about a factor of six slower, and P2R is 10–20% slower than PAR. However, PAR does improve on HI-PR by a factor of two to three. CH performance is extremely poor; it loses to BK by over three orders of magnitude. CH-n is faster than PAR and P2R, but by less than a factor of two.

Operation counts show that PAR and P2R perform between 8 and 14 scans per vertex on the stereo problems. This suggests that BK performance is not too far from optimal, as the push-relabel codes need to reduce the number of scans per vertex to about two in order to catch up with BK.

On multiview instances, PAR and P2R perform similarly to each other, and about a factor of two better than HI-PR. The comparison of these codes to BK is non-conclusive. We consider four multiview instances: camel and gargoyle, each in two sizes, small and medium. BK is faster than P2R/PAR on camel, by about a factor of four on the smaller problem and by less than a factor of two on the larger one. BK is slower on gargoyle instances by a factor of four to six.

CH crashes on multiview problems. CH-n fails feasibility and optimality self-checks on the smaller problems and fails to allocate sufficient memory for the larger problems.

Next we consider segmentation instances. PAR and P2R performance is very close, and two to three times better than that of HI-PR. CH-n is faster by less than a factor of two – much less for liver and babyface instances. BK is competitive with PAR and P2R on most problems except for liver-6-10 and babyface-6-10, where BK is about 2.5 and 4 times faster, respectively. CH crashes on the segmentation problems.

6 Concluding Remarks

We introduce the two-level push-relabel algorithm, P2R, and show that its performance is close to that of our improved implementation of PAR and superior to that of HI-PR. The new algorithm is also significantly better than the CH code and better overall than the CH-n code. For vision problems, P2R is worse than the BK algorithm for stereo problems, but competitive for multiview and segmentation problems.

An interesting question is why P2R and PAR perform better than HI-PR. Partial intuition for this is as follows. The motivation behind PAR given in [22] is that it mitigates the ping-pong effect (e.g., a push from u to v immediately followed by a relabeling of v and a push back to u). In PAR, a flow is pushed along the partial augmenting path before any vertex can push the flow back. In P2R, the flow pushed from u to v is pushed to v's neighbors before v would push flow back to u. In addition, P2R avoids activating some vertices, which has a theoretical motivation in the bipartite case [1] and seems to help in general.

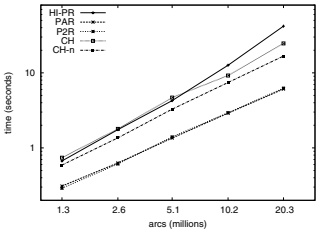
The main idea behind P2R is to push flow on length two paths, and not to create new excesses in the middle of such paths. This idea generalizes several previous algorithms. It would be interesting to investigate this idea in related contexts, such as minimum-cost flow, bipartite matching, assignment, and parametric flow problems.

References

1. Ahuja, R.K., Orlin, J.B., Stein, C., Tarjan, R.E.: Improved algorithms for bipartite network flow problems. SIAM J. Comp. (to appear)
2. Babenko, M., Derryberry, J., Goldberg, A.V., Tarjan, R.E., Zhou, Y.: Experimental evaluation of parametric maximum flow algorihtms. In: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 256–269. Springer, Heidelberg (2007)
3. 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)
4. Chandran, B., Hochbaum, D.: A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem (submitted for publication) (2007)
5. Cheriyan, J., Maheshwari, S.N.: Analysis of Preflow Push Algorithms for Maximum Network Flow. SIAM J. Comput. 18, 1057–1086 (1989)

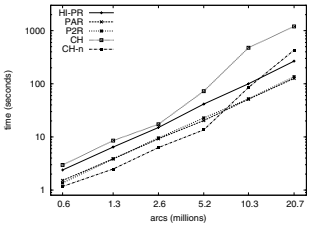
6. Cherkassky, B.V.: A Fast Algorithm for Computing Maximum Flow in a Network. In: Karzanov, A.V. (ed.) Collected Papers, Combinatorial Methods for Flow Problems, vol. 3, pp. 90–96. The Institute for Systems Studies, Moscow (1979); English translation appears in AMS Trans. 158, 23–30 (1994) (in Russian)
7. Cherkassky, B.V., Goldberg, A.V.: On Implementing Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19, 390–410 (1997)
8. Cherkassky, B.V., Goldberg, A.V., Martin anbd, P., Setubal, J.C., Stolfi, J.: Augment or push? a computational study of bipartite matching and unit capacity flow algorithms. Technical Report 98-036R, NEC Research Institute, Inc. (1998)
9. Derigs, U., Meier, W.: Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. ZOR — Methods and Models of Operations Research 33, 383–403 (1989)
10. 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)
11. Ford, L.R., Fulkerson, D.R.: Maximal Flow Through a Network. Canadian Journal of Math. 8, 399–404 (1956)
12. Gabow, H.N.: Scaling Algorithms for Network Problems. J. of Comp. and Sys. Sci. 31, 148–168 (1985)
13. Goldberg, A.V.: An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. J. Algorithms 22, 1–29 (1997)
14. Goldberg, A.V., Kennedy, R.: An Efficient Cost Scaling Algorithm for the Assignment Problem. Math. Prog. 71, 153–178 (1995)
15. Goldberg, A.V., Rao, S.: Beyond the Flow Decomposition Barrier. J. Assoc. Comput. Mach. 45, 753–782 (1998)
16. Goldberg, A.V., Tarjan, R.E.: A New Approach to the Maximum Flow Problem. J. Assoc. Comput. Mach. 35, 921–940 (1988)
17. Goldberg, A.V.: Recent developments in maximum flow algorithms. In: Arnborg, S. (ed.) SWAT 1998. LNCS, vol. 1432, pp. 1–10. Springer, Heidelberg (1998)
18. Goldberg, A.V.: The Partial Augment–Relabel Algorithm for the Maximum Flow Problem. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 466–477. Springer, Heidelberg (2008)
19. 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)
20. 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)
21. King, V., Rao, S., Tarjan, R.: A Faster Deterministic Maximum Flow Algorithm. J. Algorithms 17, 447–474 (1994)
22. Mazzoni, G., Pallottino, S., Scutella, M.G.: The Maximum Flow Problem: A Max-Preflow Approach. Eur. J. of Oper. Res. 53, 257–278 (1991)
23. Negrus, C.S., Pas, M.B., Stanley, B., Stein, C., Strat, C.G.: Solving maximum flow problems on real world bipartite graphs. In: Proc. 11th International Workshop on Algorithm Engineering and Experiments, pp. 14–28. SIAM, Philadelphia (2009)
24. Iossa, A., Cerulli, R., Gentili, M.: Efficient Preflow Push Algorithms. Computers & Oper. Res. 35, 2694–2708 (2008)
25. Sleator, D.D., Tarjan, R.E.: A Data Structure for Dynamic Trees. J. Comput. System Sci. 26, 362–391 (1983)
26. Tuncel, L.: On the Complexity of Preflow-Push Algorithms for Maximum-Flow Problems. Algorithmica 11, 353–359 (1994)

Appendix: Experimental Data



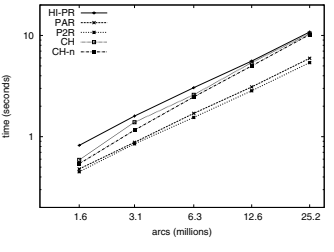
n	0.3 M	0.5 M	1.1 M	2.0 M	4.1 M
m	1.3 M	2.6 M	5.1 M	10.2 M	20.3 M
HI-PR	8.16	11.25	13.25	20.04	33.97
PAR	5.20	5.32	5.47	5.86	5.92
P2R	5.10	5.40	5.95	6.19	6.11

Fig. 1. RMF-Long problem data: time (plot), scans/vertex (table)



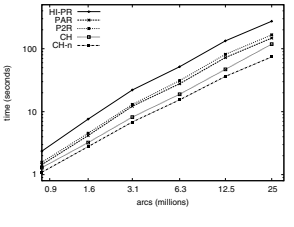
n	0.1 M	0.3 M	0.5 M	1.0 M	2.1 M	4.2 M
m	0.6 M	1.3 M	2.6 M	5.2 M	10.3 M	20.7 M
HI-PR	41.39	52.93	60.90	76.55	88.64	102.43
PAR	44.19	50.57	55.54	58.74	69.39	76.28
P2R	41.44	47.64	53.32	60.12	64.20	72.51

Fig. 2. RMF-Wide problem data: time (plot), scans/vertex (table)



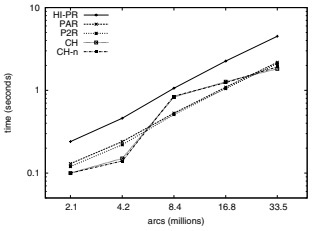
HI-PR sc/n	3.98	3.62	3.26	2.68	2.55
PAR sc/n	2.71	2.12	2.02	1.66	1.56
P2R sc/n	2.60	2.30	1.93	1.69	1.55

Fig. 3. Wash-Long problem data: time (plot), scans/vertex (table)



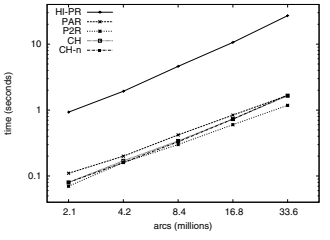
n	0.13 M	0.26 M	0.52 M	1.0 M	2.1 M	4.2 M	8.4 M
m	0.4 M	0.8 M	1.6 M	3.1 M	6.3 M	12.5 M	25.0 M
HI-PR	16.20	19.99	25.51	30.46	32.32	39.92	39.22
PAR	8.72	12.00	14.14	18.5	19.15	24.61	23.96
P2R	9.20	12.68	15.10	18.84	20.38	26.28	25.47

Fig. 4. Wash-Wide problem family: time (plot), scans/vertex (table)



n	41 K	66 K	104 K	165 K	262 K
m	2.1 M	4.2 M	8.4 M	16.8 M	33.5 M
HI-PR sc/n	2.07	2.06	2.05	2.05	2.04
PAR sc/n	1.13	1.11	1.10	1.09	1.07
P2R sc/n	1.08	1.07	1.06	1.05	1.04

Fig. 5. Wash-Line problem data: time (plot), scans/vertex (table)



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 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
P2R sc/n	1.31	1.48	1.33	1.32	1.33

Fig. 6. Acyclic-Dense problem data: time (plot), scans/vertex (table)

Table 1. Stereo vision data – running times. Problems ordered by size $(n + m)$.

name	n	m	HI-PR	PAR	P2R	CH	CH-n	BK
BVZ-tsukuba	110,594	513,467	8.07	3.60	3.91	643.62	2.80	0.59
BVZ-sawtooth	164,922	796,703	12.45	6.81	8.00	3,127.23	5.45	1.01
BVZ-venus	166,224	795,296	23.65	10.19	11.02	2,707.32	7.23	1.86
KZ2-tsukuba	199,822	1,341,101	30.39	11.81	13.09	4,020.49	6.85	1.82
KZ2-sawtooth	294,936	1,956,194	31.88	14.57	16.69	13,472.85	11.70	2.77
KZ2-venus	301,610	2,026,283	61.64	21.31	26.75	12,898.89	15.84	4.49

Table 2. Stereo vision data – operation counts

name	n	m	HI-PR	PAR	P2R
BVZ-tsukuba	110,594	513,467	10.37	8.15	9.67
BVZ-sawtooth	164,922	796,703	14.73	9.55	10.84
BVZ-venus	166,224	795,296	19.34	11.87	13.01
KZ2-tsukuba	199,822	1,341,101	10.84	6.63	8.05
KZ2-sawtooth	294,936	1,956,194	20.96	12.34	13.29
KZ2-venus	301,610	2,026,283	20.03	9.65	12.51

Table 3. Multiview reconstruction – running times

name	n	m	HI-PR	PAR	P2R	CH	CH-n	BK
gargoyle-sml	1,105,922	5,604,568	4.39	2.72	2.28	dnf	dnf	12.29
camel-sml	1,209,602	5,963,582	8.50	4.14	4.41	dnf	dnf	1.14
gargoyle-med	8,847,362	44,398,548	125.03	47.5	52.32	dnf	dnf	193.58
camel-med	9,676,802	47,933,324	159.53	67.03	72.41	dnf	dnf	43.61

Table 4. Multiview reconstruction – operation counts

name	n	m	HI-PR	PAR	P2R
gargoyle-sml	1,105,922	5,604,568	8.93	8.61	6.77
camel-sml	1,209,602	5,963,582	15.91	12.61	13.14
gargoyle-med	8,847,362	44,398,548	29.25	19.95	19.49
camel-med	9,676,802	47,933,324	33.76	25.33	24.83

Table 5. Segmentation data – running times. Instances sorted by size $(n + m)$.

name	n	m	HI-PR	PAR	P2R	CH	CH-n	BK
bone-xyzx-6-10	491,522	2,972,389	1.06	0.35	0.34	dnf	0.20	0.28
bone-xyzx-6-100	491,522	2,972,389	1.08	0.37	0.35	dnf	0.25	0.33
bone-xyz-6-10	983,042	5,929,493	2.39	0.82	0.80	dnf	0.50	0.85
bone-xyz-6-100	983,042	5,929,493	2.46	0.85	0.86	dnf	0.58	1.32
bone-xyzx-26-10	491,522	12,802,789	2.88	0.90	0.93	dnf	0.60	0.85
bone-xyzx-26-100	491,522	12,802,789	3.16	0.93	0.89	dnf	0.58	1.01
bone-xy-6-10	1,949,698	11,759,514	6.65	1.80	1.94	dnf	1.19	1.77
bone-xy-6-100	1,949,698	11,759,514	6.90	1.95	2.01	dnf	1.36	3.10
bone-xyz-26-10	983,042	25,590,293	6.36	2.01	1.99	dnf	1.20	2.52
bone-xyz-26-100	983,042	25,590,293	6.75	2.13	2.10	dnf	1.39	3.4
liver-6-10	4,161,602	25,138,821	34.88	18.73	20.77	dnf	14.18	7.59
liver-6-100	4,161,602	25,138,821	46.74	20.32	21.87	dnf	17.59	19.68
babyface-6-10	5,062,502	30,386,370	52.55	27.88	31.58	dnf	22.77	6.90
babyface-6-100	5,062,502	30,386,370	71.37	28.74	33.70	dnf	37.85	15.15

Table 6. Segmentation data – operation counts

name	n	m	HI-PR	PAR	P2R
bone-xyzx-6-10	491,522	2,972,389	3.97	2.42	2.28
bone-xyzx-6-100	491,522	2,972,389	4.03	2.47	2.33
bone-xyz-6-10	983,042	5,929,493	4.63	2.75	2.62
bone-xyz-6-100	983,042	5,929,493	4.76	2.88	2.82
bone-xyzx-26-10	491,522	12,802,789	3.96	2.36	2.37
bone-xyzx-26-100	491,522	12,802,789	4.45	2.38	2.27
bone-xy-6-10	1,949,698	11,759,514	5.96	2.89	2.91
bone-xy-6-100	1,949,698	11,759,514	6.01	3.08	2.93
bone-xyz-26-10	983,042	25,590,293	4.59	2.72	2.67
bone-xyz-26-100	983,042	25,590,293	4.83	2.82	2.79
liver-6-10	4,161,602	25,138,821	13.95	14.80	14.07
liver-6-100	4,161,602	25,138,821	17.88	15.81	14.69
babyface-6-10	5,062,502	30,386,370	20.78	20.17	19.03
babyface-6-100	5,062,502	30,386,370	26.75	20.14	19.66