OPERATIONS RESEARCH
Vol. 00, No. 0, Xxxxx 0000, pp. 000–000
ISSN 0030-364X | EISSN 1526-5463 | 00 | 0000 | 0001

INFORMS
DOI 10.1287/xxxx.0000.0000

A Computational Study of the Pseudoflow and Push-relabel Algorithms for the Maximum Flow Problem

Bala G. Chandran
Analytics Operations Engineering, Inc., bchandran@nltx.com,

Dorit S. Hochbaum
Department of Industrial Engineering and Operations Research and Walter A. Haas School of Business, University of California, Berkeley hochbaum@ieor.berkeley.edu, http://www.ieor.berkeley.edu/~hochbaum/

We present the results of a computational investigation of the pseudoflow and push-relabel algorithms for the maximum flow and minimum *s-t* cut problems. The two algorithms were tested on several problem instances from the literature. Our results show that our implementation of the pseudoflow algorithm is faster than the best known implementation of push-relabel on most of the problem instances within our computational study.

Subject classifications: Flow algorithms; parametric flow; normalized tree; lowest label; pseudoflow algorithm; maximum flow
Area of review: Networks/graphs

1. Introduction

The *maximum flow* or *max-flow* problem on a directed capacitated graph with two distinguished nodes—a source and a sink—is to find the maximum amount of flow that can be sent from the source to the sink while satisfying flow balance constraints (flow into each node other than the source and the sink equals the flow out of it) and capacity constraints (the flow on each arc does not exceed its capacity).

The *minimum s-t cut* problem, henceforth referred to as the *min-cut* problem, defined on the above graph, is to find a bi-partition of nodes—one containing the source and the other containing the sink—such that the sum of capacities of arcs from the source set to the sink set is minimized.

Ford and Fulkerson (1956) established the *max-flow min-cut theorem*, which states that the value of the max flow in a network is equal to the value of the min cut. Algorithms developed so far for the max-flow problem implicitly solve the min-cut problem and have the same theoretical complexity for both problems.

Max-flow and min-cut problems are of considerable practical interest, with applications that range from job scheduling to mining. The chapter on maximum flows in the book by Ahuja, Magnanti and Orlin (1993) describes these and other applications. The wide applicability of these problems has resulted in a substantial amount of theoretical and experimental work on the subject.

Among algorithms for the max-flow and min-cut problems, the push-relabel algorithm (sometimes referred to as the preflow-push algorithm) of Goldberg and Tarjan (1988) performs well in theory as well as in practice. The complexity of this algorithm (for the max-flow and min-cut problems) is $O(nm\log(\frac{n^2}{m}))$, using the dynamic trees data structure of Sleator and Tarjan (1983). Several studies have shown push-relabel to be computationally very efficient (for example, Ahuja et al. 1997, Anderson and Setubal 1991, Derigs and Meier 1989, Goldberg and Cherkassky 1997). The highest level variant of the push-relabel algorithm was found to have the best performance in practice (see Goldberg and Cherkassky (1997) and page 242 of Ahuja, Magnanti and Orlin (1993)).

Hochbaum (1997) introduced the pseudoflow algorithm for the maximum flow problem, based on an algorithm of Lerchs and Grossman (1965) for the maximum closure problem. A lowest label variant of the pseudoflow algorithm has the strongly polynomial complexity of $O(nm \log n)$ using dynamic trees. Anderson and Hochbaum (2002) introduced the highest label pseudoflow variant that has the same strongly polynomial complexity, and performed an extensive computational study comparing the pseudoflow algorithm to push-relabel. The highest label pseudoflow algorithm was found to perform competitively with push-relabel on many problem instances.

The pseudoflow algorithm can be initialized with any pseudoflow. A straightforward variation of the push-relabel algorithm can also be started with any pseudoflow. Anderson and Hochbaum (2002) observed that pseudoflow algorithms can sometimes be sped up significantly if started with a good initial pseudoflow.

The contribution of this paper is that we demonstrate that our implementation of the highest label pseudoflow algorithm with a generic initialization is faster than the best known implementation of highest level push-relabel on most instance classes.

This paper is organized as follows: we establish our notation in Section 2. We describe the push-relabel in Section 3, followed by the pseudoflow algorithm in Section 4. In Section 5, we show how the pseudoflow and push-relabel algorithms can be initialized with a pseudoflow. We then present our experimental setup in Section 6, followed by our results in Section 7.

2. Preliminaries

Let G_{st} be a graph (V_{st}, A_{st}), where $V_{st} = V \cup \{s,t\}$ and $A_{st} = A \cup A_s \cup A_t$ in which A_s and A_t are the source-adjacent and sink-adjacent arcs respectively. The number of nodes $|V_{st}|$ is denoted by n, while the number of arcs $|A_{st}|$ is denoted by m. A flow vector $f = \{f_{ij}\}_{(i,j)\in A_{st}}$ is said to be *feasible* if it satisfies

(i) Flow balance constraints: for each $j \in V$, $\sum_{(i,j)\in A_{st}} f_{ij} = \sum_{(j,k)\in A_{st}} f_{jk}$ (i.e., inflow(j) = outflow(j)), and

(ii) Capacity constraints: the flow value is between the lower bound and upper bound capacity of the arc, i.e., $\ell_{ij} \le f_{ij} \le u_{ij}$. We assume henceforth that $\ell_{ij} = 0$.

A *maximum flow* is a feasible flow f^* that maximizes the flow out of the source (or into the sink). The value of the maximum flow is $\sum_{(s,i)\in A_s} f^*_{si}$.

Given a capacity-feasible flow, an arc (i,j) is said to be a *residual arc* if $(i,j) \in A_{st}$ and $f_{ij} < c_{ij}$ or $(j,i) \in A_{st}$ and $f_{ji} > 0$. For $(i,j) \in A_{st}$, the residual capacity of arc (i,j) with respect to the flow f is $c^f_{ij} = c_{ij} - f_{ij}$, and the residual capacity of the reverse arc (j,i) is $c^f_{ji} = f_{ij}$. Let A^f denote the set of residual arcs for flow f in G_{st} which are all arcs or reverse arcs with positive residual capacity.

A *preflow* is a flow vector that satisfies capacity constraints but inflow into a node is allowed to exceed the outflow. The *excess* of a node $v \in V$ is the inflow into that node minus the outflow denoted by $e(v) = \sum_{(u,v)\in A_{st}} f_{uv} - \sum_{(v,w)\in A_{st}} f_{vw}$. A *pseudoflow* is a flow vector that satisfies capacity constraints but may violate flow balance in either direction (inflow into a node need not equal outflow). A negative excess is called a *deficit*.

3. The push-relabel algorithm

The push-relabel algorithm was developed by Goldberg and Tarjan (1988). In this section, we provide a sketch of a straightforward implementation of the algorithm. For a more detailed description, see Ahuja, Magnanti and Orlin (1993).

The push-relabel algorithm works with *preflows*, i.e., a flow that satisfies capacity constraints but flow into a node is allowed to exceed its outflow. A node with strictly positive excess is said to be *active*.

Each node v is assigned a label $\ell(v)$ that satisfies (i) $\ell(t) = 0$, and (ii) $\ell(u) \leq \ell(v) + 1$ if $(u, v) \in A^f$. A residual arc (u, v) is said to be *admissible* if $\ell(u) = \ell(v) + 1$.

Initially, all nodes are assigned a label of 0, and source-adjacent arcs are saturated creating a set of source-adjacent active nodes (all other nodes have zero excess). An iteration of the algorithm consists of selecting an active node in V, and attempting to push its excess to its neighbors along an admissible arc. If no such arc exists, the node's label is increased by 1. The algorithm terminates with a maximum preflow when there are no more active nodes with label less than n. The set of nodes of label n then forms the source set of a minimum cut and the current preflow is maximum in that it sends as much flow into the sink node as possible. This ends Phase 1 of the Push-relabel algorithm. In Phase 2, the algorithm transforms the maximum preflow into a maximum flow. In practice, Phase 2 is much faster than Phase 1. A high-level description of the push-relabel algorithm is shown in Figure 1.

Figure 1 High-level description of Phase I of the generic push-relabel algorithm.

```
/*
Generic push-relabel algorithm for maximum flow. The nodes with label equal to n at
termination form the source set of the minimum cut.
*/

procedure push-relabel(V_st, A_st, c):
    begin
        Set the label of s to n and that of all other nodes to 0;
        Saturate all arcs in A_s;
        while there exists an active node u ∈ V of label less than n do
            if there exists an admissible arc (u, v) do
                Push a flow of min{e(u), c^f_uv} along arc (u, v);
            else do
                Increase label of u by 1 unit;
    end
```

In the highest label and lowest label variants, an active node with highest and lowest labels respectively are chosen for processing at each iteration. In the FIFO variant, the active nodes are maintained as a queue in which nodes are added to the queue from the rear and removed from the front for processing.

The generic version of the push-relabel algorithm runs in $O(n^2 m)$ time. Using the dynamic trees data structure of Sleator and Tarjan (1983), the complexity is improved to $O(nm \log \frac{n^2}{m})$.

Two heuristics that are employed in practice significantly improve the running time of the algorithm:

1. Gap relabeling: If the label of an active node is ℓ and there are no nodes of label $\ell - 1$, the sink is no longer reachable through this node. The node is now known to be in the source set, and this node and all its arcs are removed from the graph for the rest of Phase 1 of the algorithm.

2. Global relabeling: The labels of the nodes are periodically recomputed by a finding the distance of each node from the sink in the residual graph. This "tightens" the labels and leads to significantly improved performance in practice.

Once a minimum cut has been identified, a feasible flow is recovered by flow decomposition (discussed in Section 5). In practice, the time for Phase 1 dominates the time for Phase 2.

4. The pseudoflow algorithm

In this section we provide a description of the pseudoflow algorithm. Our description here is different from than in Hochbaum (1997, 2008) although the algorithm itself is the same. This description uses terminology and concepts similar in spirit to push-relabel in order to help clarify the similarities and differences between the two algorithms1. .

The first step in the pseudoflow algorithm, called the *Min-cut Stage* finds a minimum cut in G_{st}. Source-adjacent and sink-adjacent arcs are saturated throughout this stage of the algorithm; consequently, the source and sink have no role to play in the Min-cut Stage.

The algorithm may start with any other pseudoflow that saturates arcs in $A_s \cup A_t$. Other than that, the only requirement of this pseudoflow is that the collection of *free arcs*, namely the arcs that satisfy $\ell_{ij} < f_{ij} < u_{ij}$, form an acyclic graph.

Each node in $v \in V$ is associated with at most one *current arc* $(u, v) \in A^f$; the corresponding *current node* of v is denoted by $\text{current}(v) = u$. The set of current arcs in the graph satisfies the following invariants at the beginning of every major iteration of the algorithm:

Property 1 *(a) The graph does not contain a cycle of current arcs.*
(b) If $e(v) \neq 0$, then node v does not have a current arc.

Each node is associated with a *root* that is defined constructively as follows: starting with node v, generate the sequence of nodes $\{v, v_1, v_2, \ldots, v_r\}$ defined by the current arcs $(v_1, v), (v_2, v_1), \ldots, (v_r, v_{r-1})$ until v_r has no current arc. Such node v_r always exists, as otherwise a cycle will be formed, which would violate Property 1(a). Let the unique root of node v be denoted by $\text{root}(v)$. Note that if a node v has no current arc, then $\text{root}(v) = v$.

The set of current arcs forms a *current forest*. Define a *component* of the forest to be the set of nodes that have the same root. It can be shown that each component is a directed tree, and the following properties hold for each component.

Property 2*In each component of the current forest,*
(a) The root is the only node without a current arc.
(b) All current arcs are pointed away from the root.

While it is tempting to view the ability to maintain pseudoflows as an important difference between the two algorithms, it is trivial to modify the push-relabel algorithm (as shown in Section 5) to handle pseudoflows.

The key difference between the pseudoflow and push-relabel algorithms is that the pseudoflow algorithm allows flow to be pushed along arcs (u, v) in which $\ell(u) = \ell(v)$ whereas this is not allowed in push-relabel. Goldberg and Rao (1998) proposed a maximum flow algorithm with complexity superior to that of push-relabel that relied on being able to send flow along arcs with $\ell(u) = \ell(v)$.

4.1. Initialization

The pseudoflow algorithm starts with a pseudoflow and an associated current forest. Anderson and Hochbaum (2002) showed that, in practice, the choice of the pseudoflow and initial current forest can have a significant impact on the running time of the algorithms.

The generic initialization is the *simple* initialization: source-adjacent and sink-adjacent arcs are saturated while all other arcs have zero flow.

If a node v is both source-adjacent and sink-adjacent, then at least one of the arcs (s, v) or (v, t) can be pre-processed out of the graph by sending a flow of $\min\{c_{sv}, c_{vt}\}$ along the path $s \to v \to t$. This flow eliminates at least one of the arcs (s, v) and (v, t) in the residual graph. We henceforth assume w.l.o.g. that no node is both source-adjacent and sink-adjacent.

The simple initialization creates a set of source-adjacent nodes with excess, and a set of sink-adjacent nodes with deficit. All other arcs have zero flow, and the set of current arcs is selected to be empty. Thus, each node is a singleton component for which it serves as the root, even if it is *balanced* (with 0-deficit).

A second type of initialization is obtained by saturating all arcs in the graph. The process of saturating all arcs could create nodes with excesses or deficits. Again, the set of current arcs is empty, and each node is a singleton component for which it serves as the root. We refer to this as the *saturate-all initialization* scheme.

4.2. A labeling pseudoflow algorithm

In the labeling pseudoflow algorithm, each node $v \in V$ is associated with a distance label $\ell(v)$ with the following property.

Property 3 *The node labels satisfy:*
(a) For every arc $(u,v) \in A^f$, $\ell(u) \leq \ell(v)+1$.
(b) For every node $v \in V$ *with strictly positive deficit,* $\ell(v) = 0$.

Collectively, the above two properties imply that $\ell(v)$ is a lower bound on the distance (in terms of number of arcs) in the residual network of node v from a node with strict deficit. A residual arc (u,v) is said to be *admissible* if $\ell(u) = \ell(v)+1$.

A node is said to be *active* if it has strictly positive excess. Given an admissible arc (u,v) with nodes u and v in different components, we define an *admissible path* to be the path from root(u) to root(v) along the set of current arcs from root(u) to u, the arc (u,v), and the set of current arcs (in the reverse direction) from v to root(v).

We say that a component of the current forest is a *label-n component* if for every node v of the component $\ell(v) = n$. We say that a component is a *good active component* if its root node is active and if it is not a label-n component.

An iteration of the pseudoflow algorithm consists of choosing a good active component and attempting to find an admissible arc from a *lowest labeled* node u in this component. (Choosing a lowest labeled node for processing ensures that an admissible arc is never between two nodes of the same component.) If an admissible arc (u,v) is found, a *merger* operation is performed. The merger operation consists of pushing the entire excess of root(u) towards root(v) along the admissible path and updating the excesses and the arcs in the current forest to preserve Property 1.

If no admissible arc is found, $\ell(u)$ is increased by 1 unit. A schematic description of the merger operation is shown in Figure 2. The pseudocode for the generic labeling pseudoflow algorithm is given in Figures 3 through 5.

4.3. The monotone pseudoflow algorithm

In the generic labeling pseudoflow algorithm, finding the lowest labeled node within a component may take excessive time. The *monotone implementation* of the pseudoflow algorithm efficiently finds the lowest labeled node within a component by maintaining an additional property of *monotonicity* among labels in a component.

Property 4 (Hochbaum (1997, 2008)) *For every current arc* (u,v), $\ell(u) = \ell(v)$ *or* $\ell(u) = \ell(v) - 1$.

This property implies that within each component, the root is the lowest labeled node and node labels are non-decreasing with their distance from the root. Given this property, all the lowest labeled nodes within a component form a sub-tree rooted at the root of the component. Thus,

Figure 2 (a) Components before merger (b) Before pushing flow along admissible path from r_i to r_j (c) New components generated when arc (u, v) leaves the current forest due to insufficient residual capacity.

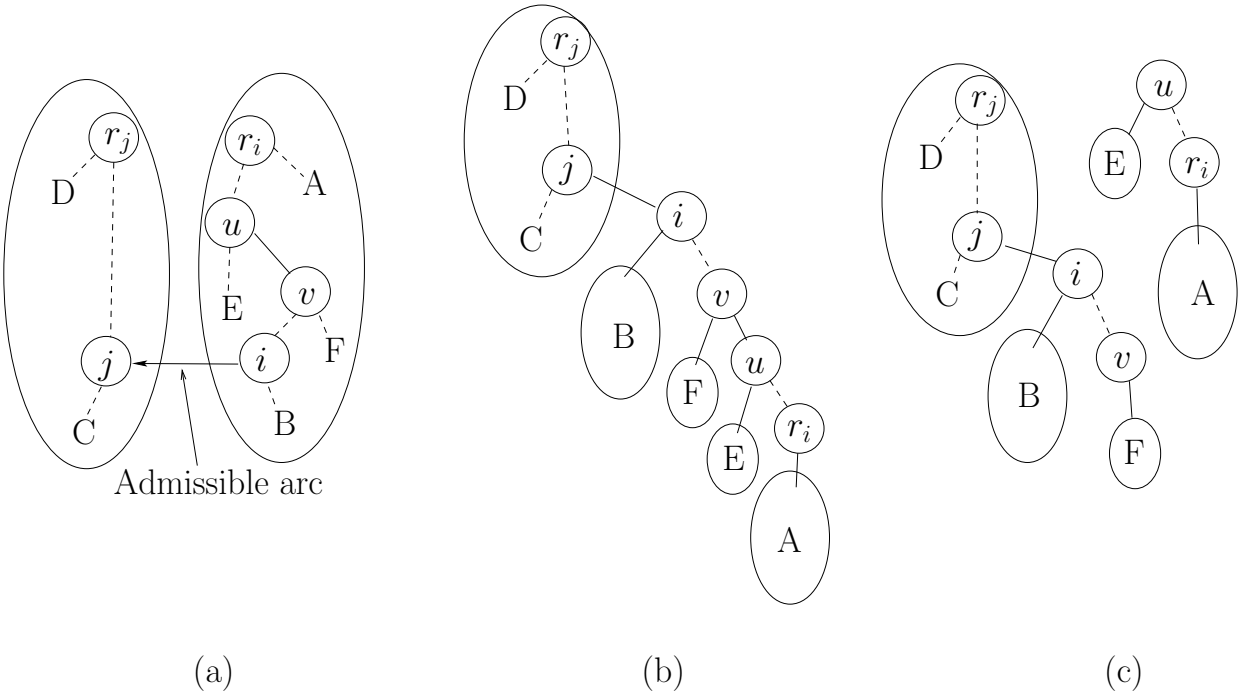


Figure 3 Generic labeling pseudoflow algorithm.

```
/*
Min-cut stage of the generic labeling pseudoflow algorithm. All nodes in label-n components form the nodes in the source set of the min-cut.
*/

procedure GenericPseudoflow (V_st, A_st, c):
    begin
        SimpleInit (A_s, A_t, c);
        while ∃ a good active component T do
            Find a lowest labeled node u ∈ T;
            if ∃ admissible arc (u, v) do
                Merger (root(u), ··· , u, v, ··· , root(v));
            else do
                ℓ(u) ← ℓ(u) + 1;
    end
```

once a good active component is identified, all the lowest labeled nodes within the component are examined for admissible arcs by performing a depth-first-search in the sub-tree starting at the root.

In the generic labeling algorithm, a node was relabeled if no admissible arc was found from the node. In the monotone implementation, a node u is relabeled only if no admissible arc is found *and* for all current arcs (u, v) in the component, $\ell(v) = \ell(u) + 1$. This feature, along with the merger process, inductively preserves the monotonicity property. The pseudocode for the Min-cut Stage of the monotone implementation of the pseudoflow algorithm is given in Figure 6.

Figure 4 Simple initialization in the generic labeling pseudoflow algorithm.

```
/*
Saturates source- and sink-adjacent arcs.
*/
procedure SimpleInit(A_s, A_t, c):
    begin
        f, e ← 0;
        for each (s, i) ∈ A_s do
            e(i) ← e(i) + c_si;
        for each (i, t) ∈ A_t do
            e(i) ← e(i) − c_it;
        for each v ∈ V do
            ℓ(v) ← 0;
            current(v) ← ∅;
    end
```

Figure 5 Push operation in the generic labeling pseudoflow algorithm.

```
/*
Pushes flow along an admissible path and preserves invariants.
*/
procedure Merger(v_1, ··· , v_k):
    begin
        for each j = 1 to k − 1 do
            if e(v_j) > 0 do
                δ ← min{c(v_j, v_{j+1}), e(v_j)};
                e(v_j) ← e(v_j) − δ;
                e(v_{j+1}) ← e(v_{j+1}) + δ;
            if e(v_j) > 0 do
                current(v_j) ← ∅;
            else do
                current(v_j) ← v_{j+1};
    end
```

The monotone implementation simply delays relabeling of a node until a later point in the algorithm, which does not affect correctness of the labeling pseudoflow algorithm.

4.4. Complexity summary

In the monotone pseudoflow implementation, the node labels in the admissible path are non-decreasing. To see that notice that for a merger along an admissible arc (u, v) the nodes along the path $\text{root}(u), \cdots, u$ all have equal label and the nodes along the path $v, \cdots, \text{root}(v)$ have non-decreasing labels (from Property 4). A merger along an admissible arc (u, v) either results in arc (u, v) becoming current, or in (u, v) leaving the residual network. In both cases, the only way (u, v) can become admissible again is for arc (v, u) to belong to an admissible path, which would require $\ell(v) \geq \ell(u)$, and then for node u to be relabeled at least once so that $\ell(u) = \ell(v) + 1$. Since $\ell(u)$ is

Figure 6 The monotone pseudoflow algorithm.

```
/*
Min-cut stage of the monotone implementation of pseudoflow algorithm. All nodes in
label-n components form the nodes in the source set of the min-cut.
*/
procedure MonotonePseudoflow (V_st, A_st, c):
    begin
        SimpleInit (A_s, A_t, c);
        while ∃ a good active component T with root r do
            u ← r;
            while u ≠ ∅ do
                if ∃ admissible arc (u, v) do
                    Merger (root(u), ··· , u, v, ··· , root(v));
                    u ← ∅;
                else do
                    if ∃w ∈ T : (current(w) = u) ∧ (ℓ(w) = ℓ(u)) do
                        u ← w;
                    else do
                        ℓ(u) ← ℓ(u) + 1;
                        u ← current(u);
    end
```

bounded by n, (u, v) can lead to $O(n)$ mergers. Since there are $O(m)$ residual arcs, the number of mergers is $O(nm)$.

The work done per merger is $O(n)$ since an admissible path is of length $O(n)$. Thus, total work done in mergers including pushes, updating the excesses, and maintaining the arcs in the current forest, is $O(n^2m)$.

Each arc (u, v) needs to be scanned at most once for each value of $\ell(u)$ to determine if it is admissible since node labels are non-decreasing and $\ell(u) \leq \ell(v)+1$ by Property 3. Thus, if arc (u, v) were not admissible for some value of $\ell(u)$, it can become admissible only if $\ell(u)$ increases. The number of arc scans is thus $O(nm)$ since there are $O(m)$ residual arcs, and each arc is examined $O(n)$ times.

The work done in relabels is $O(n^2)$ since there are $O(n)$ nodes whose labels are bounded by n.

Finally, we need to bound the work done in the depth-first-search for examining nodes within a component. Each time a depth-first-search is executed, either a merger is found or at least one node is relabeled. Thus, the number of times a depth-first-search is executed is $O(nm + n^2)$ which is $O(nm)$. The work done for each depth-first-search is $O(n)$, thus total work done is $O(n^2m)$.

Lemma 4.1 *The complexity of the monotone pseudoflow algorithm is* $O(n^2m)$.

Regarding the complexity of the algorithm, Hochbaum (1997, 2008) showed that an enhanced variant of the pseudoflow algorithm is of complexity $O(mn \log n)$. That variant uses dynamic trees data structure and is not implemented here. Recently, however, Hochbaum and Orlin (2007) showed that the "highest label" version of the pseudoflow algorithm has complexity $O(mn \log \frac{n^2}{m})$ and $O(n^3)$, with and without the use of dynamic trees, respectively.

4.5. Lowest and highest label variants

In the lowest label pseudoflow algorithm, a good active component with the lowest labeled root is processed at each iteration. In the highest label algorithm, a good active component with the highest labeled root node is processed at each iteration.

4.6. Implementation

We now describe some details of our implementation. To maintain simplicity of the code, we do not use any sophisticated data structures.

4.6.1. Limiting the number of arc scans During the labeling algorithm, the arcs adjacent to each node are examined at most once for each value of the node's label. To implement this, we maintain a pointer at each node to the arc that was last scanned to find a merger. If any node is visited more than once for a given label, the search for mergers resumes from the last scanned arc, thus ensuring that each arc is scanned at most once for each label. When a node is relabeled, the pointer is reset to the start of its list of adjacent arcs.

4.6.2. Root management The lowest and highest label variants require that all roots with positive excess and of a particular label be available when queried. To achieve this, the roots are maintained in an array of buckets, where a bucket contains all roots with positive excess and with a particular label. The order in which roots within a bucket are processed for mergers appears to make a difference to the pseudoflow algorithm. Anderson and Hochbaum (2002) experimented with three root management policies:

1. **FIFO:** Each bucket is maintained as a queue; roots are added to the rear of the queue, and roots are retrieved from the front of the queue.
2. **LIFO:** Each bucket is maintained as a stack; roots are added to the top of the stack, and roots are retrieved from the top of the stack.
3. **Wave:** This is a variant of the LIFO policy. Each bucket is still maintained as a stack, with roots being added to the top of the stack and being retrieved from the top. However, when the excess of a root changes while it is in the bucket, it is moved up to the top of the stack.

Note that the wave management policy is the same as the LIFO policy for the lowest label variant since the excess of a root with positive excess does not change while it is in a bucket. (When a root is processed in the lowest label algorithm, all mergers are from a component with positive excess to one with non-positive excess, leaving all other roots with positive excess unchanged.)

4.6.3. Gap Relabeling We use the gap-relabeling heuristic of Derigs and Meier (1989), who introduced it in the context of push-relabel. When we process a component whose root has label ℓ and there are no nodes in the graph with label $\ell - 1$, we conclude that the entire component has no residual paths to the sink and is hence a part of the source set of a min cut. The entire component can thus be ignored for the rest of the algorithm. In practice, this is achieved by setting the labels of all nodes in that component to n.

4.7. Flow recovery

The Min-cut Stage of the pseudoflow algorithm terminates with the minimum cut and a pseudoflow. *Flow recovery* refers to the process of converting the pseudoflow at the end of the Min-cut Stage to a maximum feasible flow.

Given a feasible flow in a network, *flow decomposition* refers to the process of representing the flow as the sum of flows along a set of *s*-*t* paths, and flows along a set of directed cycles, such that no two paths or cycles are comprised of the same set of arcs (details are in Ahuja, Magnanti and Orlin (1993), pages 79-83). Hochbaum (1997, 2008) showed that flow recovery can be done in $O(m \log n)$ by flow decomposition in a related network.

An equivalent implementation of flow decomposition that was used by Anderson and Hochbaum (2002) starts with each excess node and performs a depth-first-search (DFS) in the reverse flow graph (arcs with strictly positive flow) to identify paths back to the source node or cycles, and decreases flow along these paths or cycles until all excesses have been returned to the source. For the strict deficit nodes, a DFS is performed starting at each deficit node to find paths to the sink, and flow is reduced along these paths until all deficits have been returned to the sink.

Our initial experiments indicated that this form of flow recovery is not very robust since this procedure could end up finding several paths/cycles with relatively small amounts of flow on them. In order to correct this, we use a different approach that is theoretically less efficient, but is faster and more reliable in practice.

Our experiments indicated that the time spent in flow recovery is in most cases small compared to the time to find the minimum cut.

5. Initialization for push relabel and pseudoflow algorithms

As discussed in Section 4.1, the pseudoflow algorithm can be initialized with any pseudoflow and a corresponding current forest. In this section, we describe how to initialize both the pseudoflow and push-relabel algorithms with an arbitrary pseudoflow. We do this by constructing a graph corresponding to the pseudoflow such that the min cut and max flow can be obtained by solving the min cut and max flow problems on this new graph.

For a pseudoflow f in the graph $G_{st} = (V_{st}, A_{st})$, we construct a graph G', where pseudoflow f is feasible, by adding the following arcs to A_{st}:

1. A set of excess arcs denoted by $A'_s = \{(i,s)\ \forall i \in V : e(i) > 0\}$. Each arc (i,s) has capacity $c'_{is} = e(i)$.

2. A set of deficit arcs denoted by $A'_t = \{(t,i)\ \forall i \in V : e(i) < 0\}$. Each arc (t,i) has capacity $c'_{ti} = |e(i)|$.

The capacities of all other arcs in G' are the same as that in G, i.e., $c'_{ij} = c_{ij}\ \forall\ (i,j) \in A_{st}$. Since only added arcs are directed from the sink and into the source, the *minimum cut partition in G' is the same as that in G_{st}.*

Let the flow vector f' in G' be obtained by setting $f'_{ij} = f_{ij}\ \forall (i,j) \in A_{st}$ and $f'_{ij} = c'_{ij}\ \forall (i,j) \in A'_s \cup A'_t$. The flow f' is feasible in G' since flow balance is enforced at all nodes by construction. Hence, the residual graph $G'^{f'}$ for this feasible flow will have the same minimum cut partition as that in G', and hence the same as that in G_{st}.

This leads to the following claim:

Claim 5.1 *The minimum cut in graph G_{st} initialized with an arbitrary pseudoflow f can be obtained using any minimum cut algorithm by solving for the minimum cut in the graph $G'^{f'}$.*

To summarize, our initialization procedure given a pseudoflow f in G_{st} is to generate the graph $G'^{f'}$ which has node set V_{st} and the following arcs:

1. For each $(i,j) \in A_{st}$ with flow f_{ij} and capacity c_{ij}, $A'^{f'}$ contains two arcs—an arc (i,j) with capacity $c_{ij} - f_{ij}$ and an arc (j,i) with capacity f_{ij}.

2. For each node $i \in V$ with excess $e(i) > 0$, $A'^{f'}$ contains the arc (s,i) with capacity $e(i)$.

3. For each node $i \in V$ with excess $e(i) < 0$, $A'^{f'}$ contains the arc (i,t) with capacity $|e(i)|$.

We now show how to convert a maximum flow in $G'^{f'}$ to a maximum flow in G_{st}.

Claim 5.2 *Given a maximum flow in $G'^{f'}$, it is possible to construct a maximum flow in G_{st} in $O(m \log n)$ time.*

Table 1 Average running times for Dimacs machine calibration tests.

	Test 1			Test 2		
	real	user	system	real	user	system
No optimization	0.4	0.4	0.0	3.3	3.3	0.0
`-O4` flag	0.2	0.1	0.0	2.0	1.9	0.0

L et the maximum flow in $G'^{f'}$ be denoted by f^*. Since $G'^{f'}$ is simply the residual graph corresponding to a feasible flow in G', the maximum flow in G' is given by $(f'_{ij} + f^*_{ij} - f^*_{ji})$ for each arc $(i,j) \in A'$.

Since we have a feasible flow, we decompose the maximum flow in G' into a set of simple paths and cycles. Since arcs in $A'_s \cup A'_t$ are either directed into the source or out of the sink, they cannot belong to any of the simple paths from s to t. Thus, the flows on arcs A'_s and A'_t can only belong to the set of cycles in the decomposed flow. Eliminating the flow on all cycles which contain A'_s and A'_t, we obtain a flow vector such that the flow on the arcs A'_s and A'_t is zero. This flow vector is feasible to G_{st} since

1. The flows on A'_s and A'_t are zero, so these arcs can be removed from G'; this gives us G_{st}.
2. Eliminating cycles preserves flow balance at all nodes.
3. The flow $f' \leq c'_{ij} = c_{ij}$ for all arcs in A_{st}. Therefore, reducing f' will generate a capacity-feasible flow in G_{st}.

Further, since we only eliminated flows along cycles, the resulting flow vector achieves the same total flow as the maximum flow in G' and is hence optimal to G_{st}.

G' has at most $(m+n)$ arcs and n nodes, so the total work done in flow decomposition is $O(mn)$. Using the dynamic trees data structure of Sleator and Tarjan (1983), this can be improved to $O(m \log n)$. Q.E.D.

6. Experiments

This section describes our experimental setup and testing methodology.

6.1. Implementations

We implemented five variants of the pseudoflow algorithm: highest label with FIFO buckets (pseudo_hi_fifo), highest label with LIFO buckets (pseudo_hi_lifo), highest label with wave buckets (pseudo_hi_wave), lowest label with FIFO buckets (pseudo_lo_fifo), and highest label with LIFO buckets (pseudo_hi_lifo). The latest version of the code (version 3.21) is available at Pseudoflow solver (2007).

The implementation of the highest level push-relabel algorithm `hi_pr` (version 3.6) was obtained from Goldberg (2007). This implementation is an improvement over the `h_prf` implementation by Goldberg and Cherkassky (1997) which was previously considered to be the best implementation of push-relabel.

6.2. Computing Environment

The experiments were run on a Sun UltraSPARC workstation with a 270 MHz CPU and 192 MB of RAM. All codes were written in C and compiled with the `gcc` compiler using the `-O4` optimization flag.

We performed the machine calibration experiment as suggested by DIMACS (1990). Table 1 shows the running times for the two tests with and without compiler optimization.

6.3. Problem Classes

The problem instances we use are the well-known generators used in DIMACS (1990) and are available as part of CATS (2007). Unless otherwise stated, the instance generated depends on a random seed. These problem classes are:

- **AC**: The acyclic dense network family with parameter k has $n = 2^k$ nodes and $n(n+1)/2$ arcs.
- **AK**: The AK generator was designed by Goldberg and Cherkassky (1997) as a hard set of instances for push-relabel and Dinic's algorithms. Given a parameter k, the program generates a unique network with $4k+6$ nodes and $6k+7$ arcs. The instance does not depend on a random seed in that the graph, given the number of nodes, is unique.
- **GENRMF-Long**: This family is created by the RMFGEN generator of Goldfarb and Grigoriadis (1988). A network with $n = 2^x$ nodes is generated using parameters $a = 2^{x/4}$ and $b = 2^{x/2}$.
- **GENRMF-Wide**: This family is created by the RMFGEN generator. A network with $n = 2^x$ nodes is generated using parameters $a = 2^{2x/5}$ and $b = 2^{x/5}$.
- **Washington RLG-Long**: A network with $n = 2^x$ nodes in this family is generated by the Washington generator using **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 generator using **function** = 2, **arg1** = 2^{x-6}, **arg2** = 64, and **arg3** = 10^4.
- **Washington Line-Moderate**: A network with $n = 2^x$ nodes in this family is generated by the Washington generator using **function** = 6, **arg1** = 2^{x-2}, **arg2** = 4, and **arg3** = $2^{(x/2)-2}$.
- **Maximum Closure**: A set of nodes $D \subseteq V$ in a directed graph $G = (V, A)$ is called *closed* if all the successors of nodes in D are also contained in D. The *maximum closure problem* is stated as follows: Given a directed graph $G = (V, A)$, and node weights (positive, negative or zero) b_i for all $i \in V$, find a closed subset $V' \subseteq V$ such that $\sum_{j \in V'} b_j$ is maximum. The maximum closure problem has many applications such as open-pit mining (Picard 1976). The maximum closure problem reduces to a minimum s-t cut problem in a so-called "closure" graph where all source-adjacent and sink-adjacent arcs have finite capacity and all other arcs have infinite capacity.

The instance generator creates graphs from the following four inputs.

1. n: The number of nodes in the graph.
2. p: The probability of existence of each arc (i, j). Note that this could create cycles in the graph.
3. w: The probability that each node is weighted. Given that a node is weighted, its weight is an integer uniformly distributed in $[-10000, 10000]$.
4. s: A random seed to initialize the random number generator.

6.4. Testing Methodology

We first compared the different variants of the pseudoflow algorithms (highest with FIFO, LIFO, and Wave buckets and lowest with FIFO and LIFO buckets) to see if there is a difference in their performance (all algorithms have the same theoretical complexity). The relative running times of the algorithms for small instances for small instances is shown in Table 2.

The lowest label algorithm is clearly dominated by the highest label algorithm for all instances except the AC problem family. While there is usually little difference between the highest label variants, the FIFO variant clearly out-performs the others on the GENRMF-Wide family of instances. Overall, we chose the FIFO variant to be the best overall and compared this variant to the highest level push relabel algorithm, which was shown to be best of the push-relabel variants by Goldberg and Cherkassky (1997).

For each problem type of a particular size, we generated 10 instances each using a different seed. The sequence of seeds was itself generated randomly. For each instance, we averaged times over 5 runs. Thus, for instances that depend on a random seed, each data point for a given problem

Table 2 Relative running times of highest and lowest label pseudoflow algorithms on "small" instances. Times include time for flow recovery.

			Highest Label			Lowest Label	
Instance family	n	m	FIFO	LIFO	Wave	FIFO	LIFO
AC	256	32,640	1.320	1.760	1.440	1.360	1.000
	512	130,816	1.397	1.381	1.365	1.143	1.000
	1,024	523,776	1.472	1.385	1.442	1.121	1.000
	2,048	2,096,128	1.840	2.001	1.849	1.162	1.000
AK	8,198	12,295	1.009	1.000	1.005	2.017	2.012
	16,390	24,583	1.002	1.000	1.001	2.021	2.019
	32,774	49,159	1.000	1.001	1.001	2.367	2.358
	65,542	98,311	1.000	1.003	1.001	2.097	2.084
GENRMF-Long	9,100	41,760	1.057	1.000	1.039	16.600	16.471
	15,488	71,687	1.000	1.079	1.072	22.967	23.088
	30,589	143,364	1.162	1.000	1.011	34.253	35.073
	65,536	311,040	1.000	1.095	1.021	50.851	53.685
GENRMF-Wide	8,214	38,813	1.000	1.646	1.434	2.217	8.706
	16,807	80,262	1.000	1.467	1.845	2.521	13.211
	32,768	157,696	1.000	2.131	1.993	2.519	16.328
	63,504	307,440	1.000	2.839	2.478	3.605	18.750
RLG-Long	16,386	49,088	1.052	1.000	1.017	24.036	24.217
	32,770	98,240	1.092	1.000	1.092	55.414	56.138
	65,538	196,544	1.124	1.000	1.030	123.417	126.579
	131,074	393,152	1.088	1.000	1.022	259.719	267.464
RLG-Wide	8,194	24,448	1.068	1.000	1.004	6.829	6.938
	16,386	48,896	1.071	1.001	1.000	8.455	8.568
	32,770	97,792	1.042	1.006	1.000	8.336	8.224
	65,538	195,584	1.089	1.000	1.096	7.907	8.022
Line-moderate	4,098	65,023	1.004	1.000	1.062	9.996	10.912
	8,194	187,352	1.029	1.000	1.033	9.926	10.969
	16,386	522,235	1.000	1.060	1.139	19.649	20.985
	32,770	1,470,491	1.035	1.007	1.000	17.231	18.529

size is the average of 50 runs. For the AK problem family (where the graph is unique for a given problem size), each data point was the average of 5 runs of the instance.

In order to demonstrate the effect of initialization on push-relabel and pseudoflow, we consider only the AK problem class where the saturate-all initialization was found to substantially improve the performance of both algorithms.

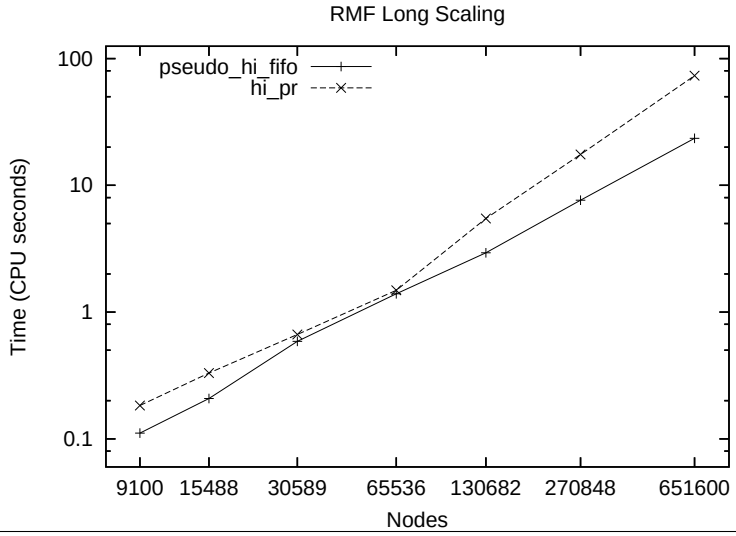
We collected operation counts for all runs. For each family, we report on the "common" operations to both algorithms, i.e., relabels, arc scans, and pushes. For push-relabel, the number of relabels does not include those performed during global relabeling.

We report run-times (time to find minimum cut and time to find the maximum flow) for all instances. These run-times are CPU times obtained using the `getrusage` function in C, and does not include time to read the input or print the solution. Since operation counts (which involve long integer addition) from interfering with the run-time, we ran each instance twice—once with operation counts being collected and once without—and report run-times are from the runs during which no operation counts were collected.

7. Results

In this section, we provide the results of our experiments for each problem family.

Figure 7 Run times and operation counts for GENRMF-Long instances.



SIMPLE INITIALIZATION

		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
9,100	41,760	0.111	0.183	1.000	1.649	0.126	0.213	1.000	1.689
15,488	71,687	0.208	0.330	1.000	1.588	0.234	0.375	1.000	1.602
30,589	143,364	0.587	0.665	1.000	1.132	0.652	0.779	1.000	1.195
65,536	311,040	1.395	1.484	1.000	1.064	1.512	1.711	1.000	1.132
130,682	625,537	2.935	5.472	1.000	1.864	3.325	6.081	1.000	1.829
270,848	1,306,607	7.627	17.509	1.000	2.295	8.169	18.582	1.000	2.275
651,600	3,170,220	23.483	73.272	1.000	3.120	24.893	76.553	1.000	3.075

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
9,100	41,760	62,168	65,820	53,765	38,953	192,386	384,584
15,488	71,687	103,083	117,142	93,847	69,189	337,308	656,046
30,589	143,364	223,333	228,170	205,848	137,018	754,240	1,283,615
65,536	311,040	542,475	473,126	511,926	292,081	1,913,935	2,933,002
130,682	625,537	1,175,273	1,620,216	1,046,223	1,158,262	3,958,944	11,401,882
270,848	1,306,607	2,650,312	4,685,107	2,564,604	3,381,599	9,808,078	30,466,028
651,600	3,170,220	7,858,941	16,987,429	6,826,351	13,254,386	26,496,009	131,500,478

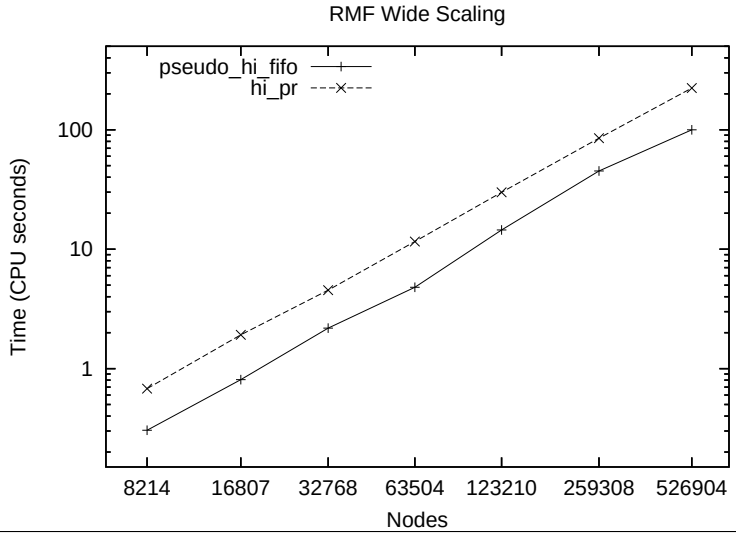
7.1. GENRMF-Long

Push-relabel is slower than pseudoflow on the smaller instances but appears to almost catch up with pseudoflow for the medium-sized instances. However, it subsequently becomes relatively slower, and the gap between the two algorithms appears to increase with problem size. This appears to correlate with the number of relabels performed. Push-relabel initially performs fewer relabels than does pseudoflow but performs more on the larger instances. The number of global relabels also goes up for the larger instances; the average number of global relabels performed on the six instances were 5.1, 5.2, 5.3, 5, 9.5, 13.3, and 21 respectively.

Note that the number of relabels and arc scans reported here for push relabel do not include the number of relabels performed during global relabeling.

7.2. GENRMF-Wide

Figure 8 Run times and operation counts for GENRMF-Wide instances.



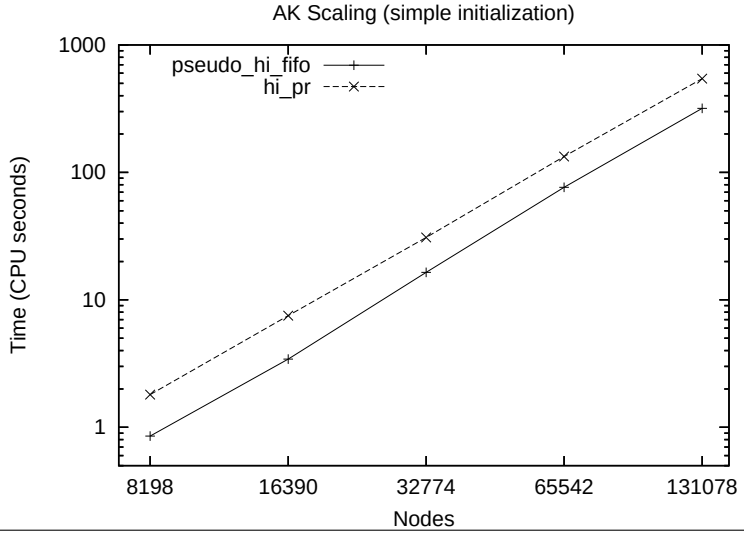
SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,214	38,813	0.304	0.679	1.000	2.229	0.321	0.735	1.000	2.292
16,807	80,262	0.808	1.920	1.000	2.375	0.872	2.078	1.000	2.384
32,768	157,696	2.184	4.540	1.000	2.079	2.313	5.668	1.000	2.450
63,504	307,440	4.803	11.575	1.000	2.410	5.046	16.255	1.000	3.221
123,210	599,289	14.489	29.938	1.000	2.066	15.906	37.209	1.000	2.339
259,308	1,267,875	45.228	84.931	1.000	1.878	50.007	107.161	1.000	2.143
526,904	2,586,020	99.951	223.596	1.000	2.237	102.228	301.137	1.000	2.946

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,214	38,813	318,368	242,160	79,765	119,899	324,192	1,264,679
16,807	80,262	795,596	610,992	185,494	302,675	760,255	3,118,748
32,768	157,696	1,906,289	1,245,410	385,992	677,448	1,581,645	7,157,116
63,504	307,440	4,109,525	2,946,034	850,820	1,627,468	3,509,773	17,566,754
123,210	599,289	13,924,860	7,446,489	1,798,138	4,018,154	7,442,460	41,840,711
259,308	1,267,875	40,741,173	19,432,456	3,902,188	10,923,983	16,160,617	117,956,644
526,904	2,586,020	82,202,433	46,445,026	9,423,555	26,388,184	38,835,979	283,460,191

The pseudoflow implementation is consistently faster than push-relabel (it performs more pushes than push-relabel but fewer relabels and arc scans). We also observed that push-relabel performs an order of magnitude more global relabels compared to other instances.

7.3. AK

Figure 9 Run times and operation counts for AK instances with simple initialization.



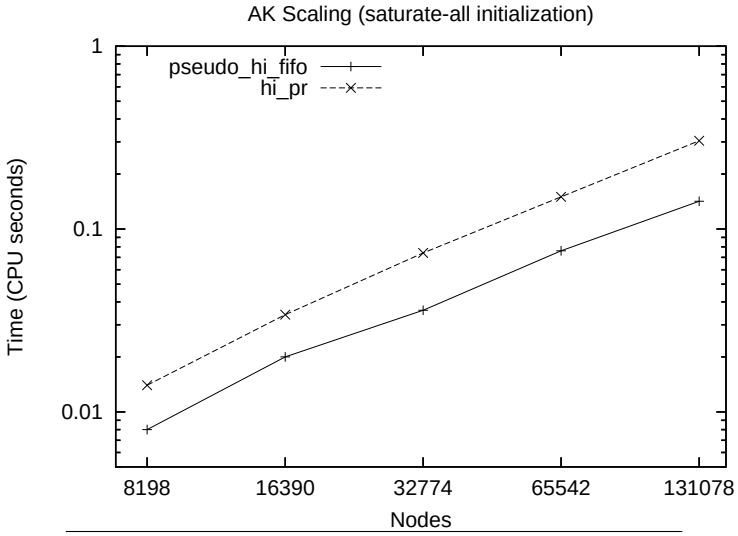
SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,198	12,295	0.852	1.806	1.000	2.120	0.856	1.810	1.000	2.114
16,390	24,583	3.428	7.520	1.000	2.194	3.438	7.534	1.000	2.191
32,774	49,159	16.428	30.876	1.000	1.879	16.446	30.906	1.000	1.879
65,542	98,311	76.322	133.248	1.000	1.746	76.356	133.310	1.000	1.746
131,078	196,615	317.770	544.794	1.000	1.714	317.840	544.926	1.000	1.714

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,198	12,295	2,108,420	2,112,487	10,243	10,245	16,388	2,141,498
16,390	24,583	8,411,140	8,419,460	20,483	20,485	32,772	8,478,188
32,774	49,159	33,599,492	33,615,862	40,963	40,965	65,540	33,731,795
65,542	98,311	134,307,844	134,341,737	81,923	81,925	131,076	134,576,974
131,078	196,615	537,051,140	537,116,756	163,843	163,845	262,148	537,581,429

The highest label pseudoflow FIFO variant is faster than push-relabel, though it is noted that the AK family of instances were designed to be a hard set of problems for push-relabel and poor performance is to be expected. We see that while both pseudoflow and push-relabel perform roughly the same number of pushes and relabels, push-relabel performs a significantly larger number of arc scans.

The run-time of both algorithms decreases significantly when the saturate-all initialization is used, as shown in Figure 10. We report only the time to minimum cut for this initialization since the transformation described in Section 5 preserves only the minimum cut in the graph and additional work (equivalent to flow decomposition) needs to be done to recover the maximum flow in the original graph.

Figure 10 Run times and operation counts for AK instances with saturate-all initialization.



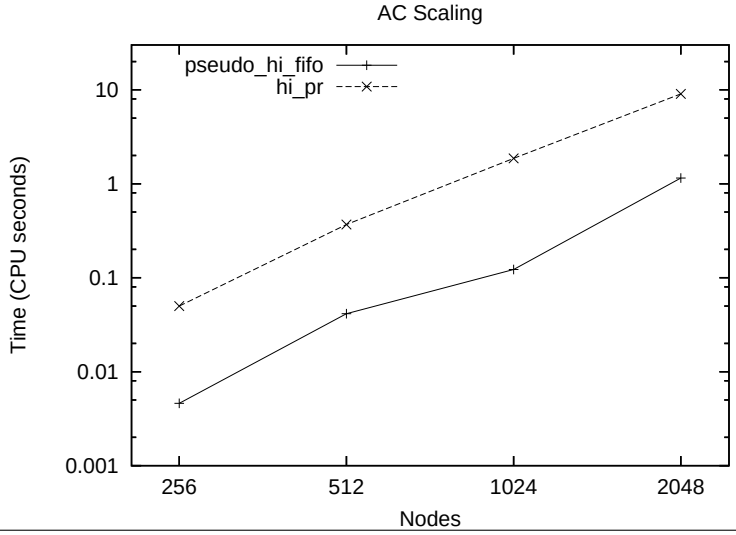
SATURATE-ALL INITIALIZATION					
		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,198	12,295	0.008	0.014	1.000	1.750
16,390	24,583	0.020	0.034	1.000	1.700
32,774	49,159	0.036	0.074	1.000	2.056
65,542	98,311	0.076	0.150	1.000	1.974
131,078	196,615	0.142	0.304	1.000	2.141

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,198	12,295	2,049	8,802	6	4,097	4,095	22,157
16,390	24,583	4,097	18,337	6	8,193	6,389	45,770
32,774	49,159	8,193	36,800	6	16,383	16,383	91,740
65,542	98,311	16,385	71,795	6	32,769	32,767	179,657
131,078	196,615	32,769	146,743	6	65,537	34,458	364,427

An AK instance is an acyclic network in which saturating all arcs preserves flow balance for many nodes. This results in a residual network in which a large number of nodes are unreachable from the source and/or from which the sink is unreachable, which makes the problem easy to solve.

7.4. Acyclic Dense

Figure 11 Run times and operation counts for acyclic dense (AC) instances.



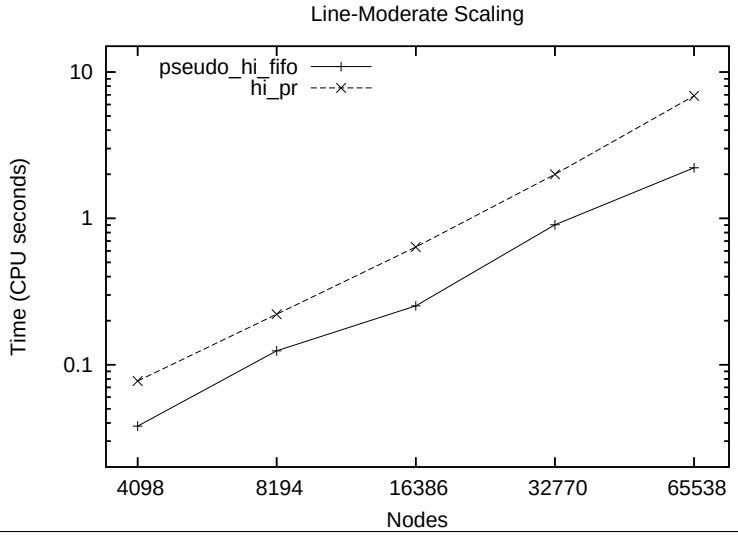
SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
256	32,640	0.005	0.050	1.000	10.870	0.007	0.057	1.000	8.576
512	130,816	0.041	0.369	1.000	8.903	0.053	0.395	1.000	7.489
1,024	523,776	0.123	1.862	1.000	15.189	0.146	1.968	1.000	13.481
2,048	2,096,128	1.149	9.056	1.000	7.880	1.342	9.493	1.000	7.072

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
256	32,640	562	1,571	365	522	23,246	126,603
512	130,816	1,441	3,782	1,195	1,377	178,125	653,232
1,024	523,776	2,650	7,954	1,913	2,862	495,358	2,702,803
2,048	2,096,128	7,803	20,378	6,594	7,135	4,451,181	13,078,897

The pseudoflow implementation is significantly faster than push-relabel on all instance sizes; we observe that push-relabel performs significantly many more arc scans and pushes. We noticed that the number of pushes per merger was comparable to the number of nodes (for the largest instances, it is 40–50% of the number of nodes). We suspect that these long push sequences (along arcs in which both end points possibly have the same label) are responsible for pseudoflow's good performance.

7.5. Line Moderate

Figure 12 Run times and operation counts for Line Moderate instances.



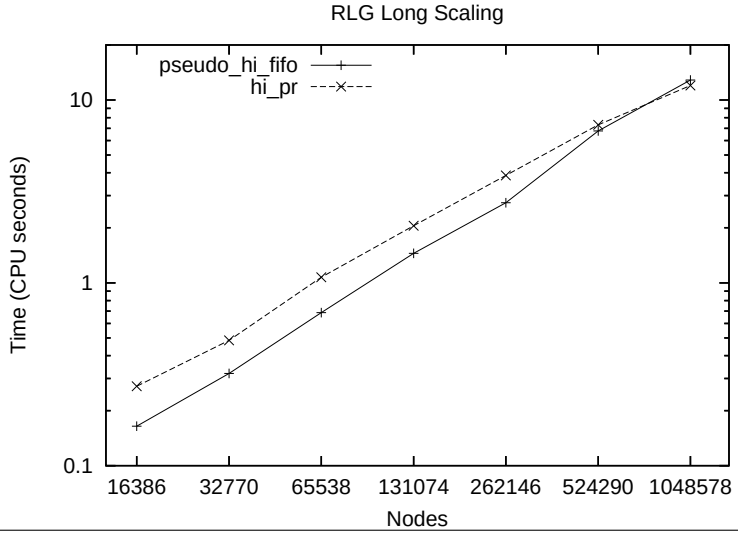
SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
4,098	65,023	0.038	0.077	1.000	2.037	0.045	0.099	1.000	2.181
8,194	187,352	0.124	0.221	1.000	1.780	0.164	0.304	1.000	1.851
16,386	522,235	0.252	0.637	1.000	2.523	0.294	0.778	1.000	2.650
32,770	1,470,491	0.904	2.002	1.000	2.215	1.185	2.651	1.000	2.238
65,538	4,186,085	2.215	6.880	1.000	3.106	2.852	8.499	1.000	2.980

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
4,098	65,023	11,125	12,746	8,662	4,730	78,639	176,949
8,194	187,352	21,617	25,229	19,675	9,273	272,051	483,119
16,386	522,235	41,045	49,754	32,916	18,101	550,411	1,285,367
32,770	1,470,491	81,213	99,056	79,929	35,709	2,148,608	3,498,246
65,538	4,186,085	153,277	189,258	146,661	70,134	5,251,717	9,594,976

The pseudoflow implementation is again faster on all instances due to fewer arc scans. This was one of only two families where pseuduflow performed more relabels.

7.6. RLG Long

Figure 13 Run times and operation counts for RLG-Long instances.



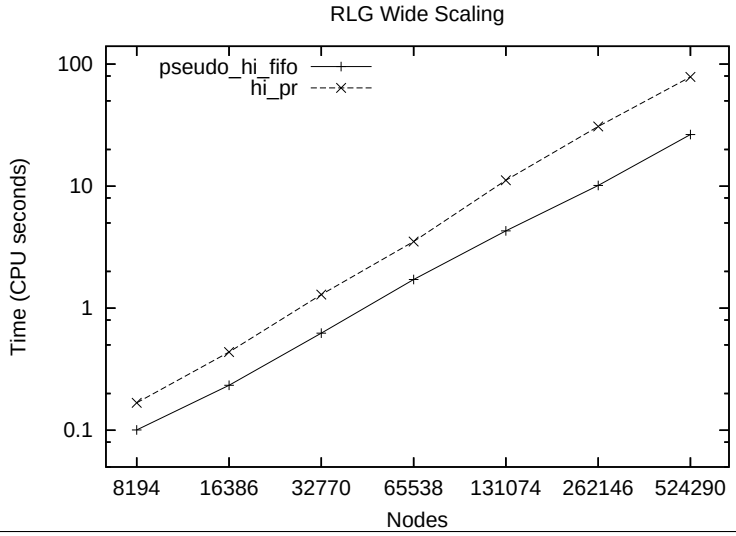
SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
16,386	49,088	0.164	0.272	1.000	1.653	0.188	0.310	1.000	1.653
32,770	98,240	0.319	0.484	1.000	1.517	0.357	0.548	1.000	1.535
65,538	196,544	0.687	1.073	1.000	1.562	0.778	1.220	1.000	1.568
131,074	393,152	1.449	2.054	1.000	1.417	1.646	2.365	1.000	1.437
262,146	786,368	2.744	3.863	1.000	1.408	3.156	4.472	1.000	1.417
524,290	1,572,800	6.771	7.318	1.000	1.081	7.637	8.614	1.000	1.128
1,048,578	3,145,664	12.867	12.007	1.072	1.000	14.552	14.195	1.025	1.000

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
16,386	49,088	95,273	146,351	80,491	49,978	191,360	424,079
32,770	98,240	176,177	278,940	154,187	97,107	362,301	826,264
65,538	196,544	343,385	580,581	327,157	207,667	757,382	1,757,393
131,074	393,152	678,708	1,096,443	693,000	386,306	1,587,599	3,303,730
262,146	786,368	1,285,622	1,995,391	1,294,217	699,376	2,957,115	6,080,414
524,290	1,572,800	2,498,234	3,711,744	2,689,543	1,283,366	6,081,694	11,327,436
1,048,578	3,145,664	4,797,765	6,087,748	5,058,970	1,969,792	11,425,647	18,228,497

Push-relabel scales better than pseudoflow with problem size, although it is better than pseudoflow only on the largest instances. This was one of only two families where pseuduflow performed more relabels.

7.7. RLG Wide

Figure 14 Run times and operation counts for RLG-Wide instances.



SIMPLE INITIALIZATION									
		Minimum cut				Maximum flow			
		Time (sec.)		Relative time		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,194	24,448	0.101	0.168	1.000	1.666	0.114	0.188	1.000	1.652
16,386	48,896	0.234	0.437	1.000	1.872	0.257	0.474	1.000	1.841
32,770	97,792	0.624	1.293	1.000	2.072	0.680	1.375	1.000	2.022
65,538	195,584	1.719	3.521	1.000	2.049	1.843	3.691	1.000	2.002
131,074	391,168	4.299	11.163	1.000	2.597	4.627	11.506	1.000	2.487
262,146	782,336	10.135	30.838	1.000	3.043	11.181	31.618	1.000	2.828
524,290	1,564,672	26.488	78.538	1.000	2.965	30.359	80.322	1.000	2.646

		Pushes		Relabels		Arc scans	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel	pseudo-flow	push-relabel
8,194	24,448	59,204	100,287	45,181	35,101	110,922	291,507
16,386	48,896	127,923	230,186	91,458	81,527	226,446	670,912
32,770	97,792	276,483	587,053	201,804	216,799	496,752	1,753,941
65,538	195,584	588,634	1,296,152	456,952	482,079	1,108,638	3,882,692
131,074	391,168	1,303,234	3,345,024	979,796	1,291,646	2,372,407	10,169,333
262,146	782,336	2,723,785	7,895,722	2,105,484	3,137,422	5,060,915	24,296,062
524,290	1,564,672	5,780,423	18,368,338	5,081,726	7,440,283	11,899,792	56,996,936

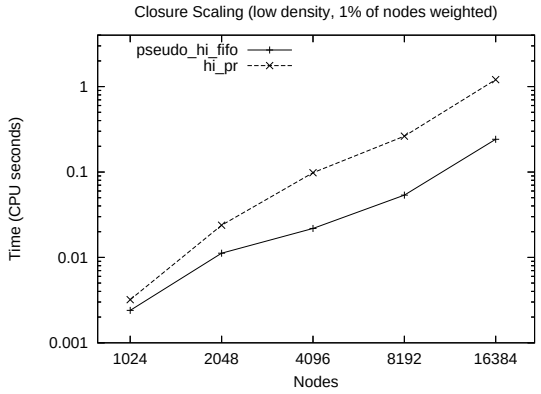
Pseudoflow is faster on all instances in addition to scaling better than push-relabel. The operation counts do not provide much insight into the differences between the two algorithms.

7.8. Maximum Closure

We report the results of our instances on nine classes of graphs: three values of density of weighted nodes in the graph (1%, 10% and 100%), and three values of arc densities (0.05%, 5% and 50%) for each of the weight densities.

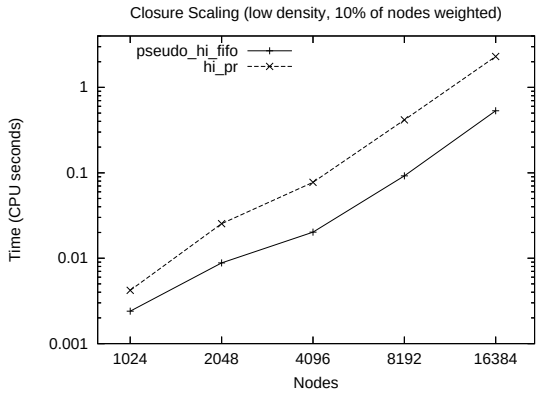
We report only the time to the minimum cut since the maximum closure problem, by definition, is only to solve for the minimum cut.

Figure 15 Results for low density closure instances (arc density 0.5%).



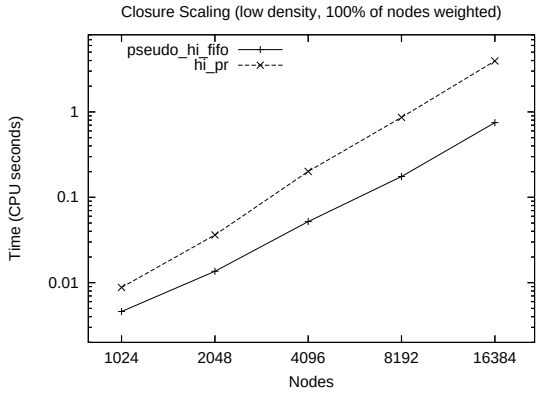
1% OF NODES WEIGHTED

		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
1,024	5,263	0.002	0.003	1.000	1.333
2,048	20,995	0.011	0.024	1.000	2.125
4,096	83,930	0.022	0.098	1.000	4.495
8,192	336,031	0.054	0.263	1.000	4.899
16,384	1,343,460	0.242	1.207	1.000	4.996



10% OF NODES WEIGHTED

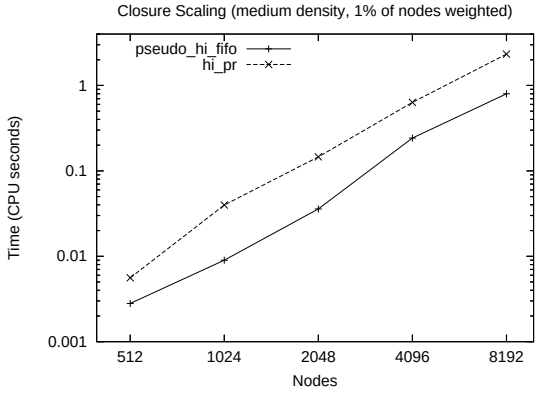
		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
1,024	5,333	0.002	0.004	1.000	1.750
2,048	21,199	0.009	0.025	1.000	2.886
4,096	84,154	0.020	0.077	1.000	3.832
8,192	336,167	0.092	0.416	1.000	4.512
16,384	1,345,002	0.533	2.300	1.000	4.315



100% OF NODES WEIGHTED

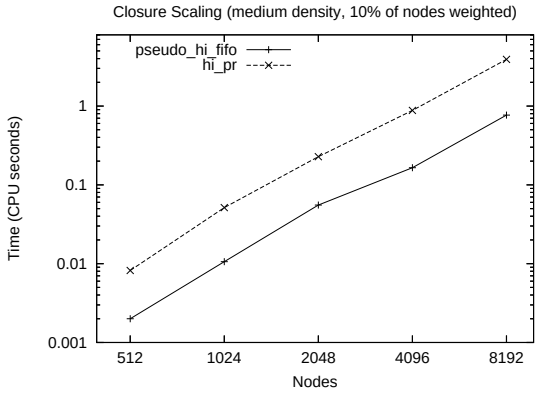
		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
1,024	6,269	0.005	0.009	1.000	1.913
2,048	22,977	0.014	0.036	1.000	2.662
4,096	88,091	0.052	0.201	1.000	3.869
8,192	344,121	0.175	0.863	1.000	4.938
16,384	1,359,906	0.749	3.950	1.000	5.271

Figure 16 Results for medium density closure instances (arc density 5%).



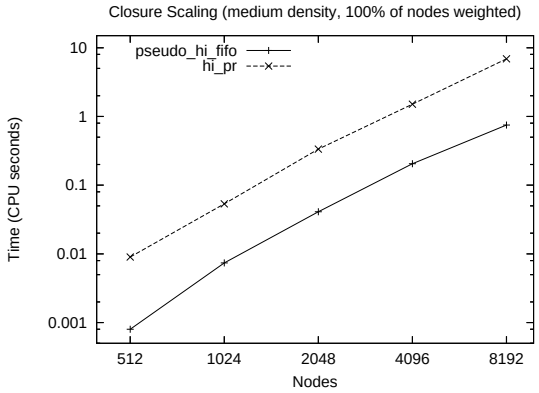
1% OF NODES WEIGHTED

		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
512	12,997	0.003	0.006	1.000	2.000
1,024	52,178	0.009	0.040	1.000	4.422
2,048	209,470	0.036	0.146	1.000	4.073
4,096	838,568	0.242	0.634	1.000	2.618
8,192	3,355,771	0.800	2.325	1.000	2.907



10% OF NODES WEIGHTED

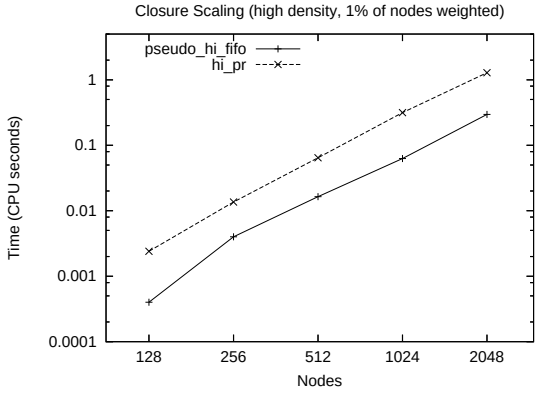
		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
512	12,983	0.002	0.008	1.000	4.100
1,024	52,245	0.011	0.051	1.000	4.849
2,048	209,440	0.055	0.228	1.000	4.116
4,096	838,571	0.165	0.879	1.000	5.322
8,192	3,355,380	0.767	3.906	1.000	5.094



100% OF NODES WEIGHTED

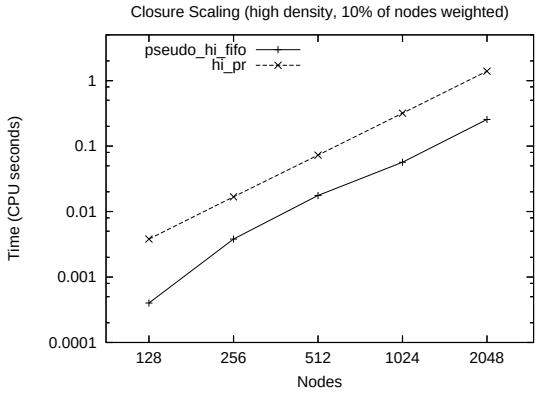
		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
512	13,509	0.001	0.009	1.000	11.250
1,024	53,211	0.007	0.053	1.000	7.216
2,048	211,418	0.041	0.335	1.000	8.166
4,096	842,690	0.205	1.508	1.000	7.356
8,192	3,363,345	0.749	6.903	1.000	9.214

Figure 17 Results for high density closure instances (arc density 50%).



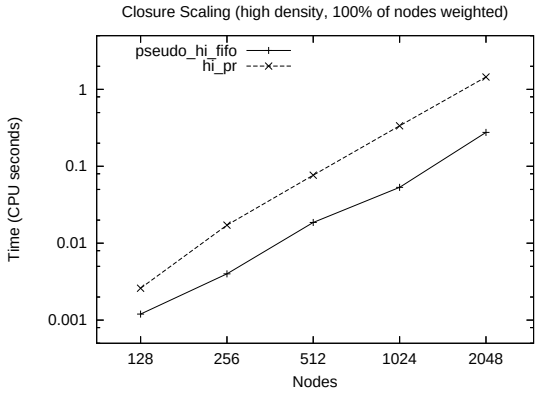
1% OF NODES WEIGHTED

		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
128	7,943	0.000	0.002	1.000	6.000
256	32,226	0.004	0.014	1.000	3.400
512	130,052	0.016	0.064	1.000	3.927
1,024	522,184	0.063	0.315	1.000	5.022
2,048	2,092,818	0.296	1.283	1.000	4.338



10% OF NODES WEIGHTED

		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
128	7,920	0.000	0.004	1.000	9.500
256	32,309	0.004	0.017	1.000	4.421
512	130,050	0.018	0.073	1.000	4.136
1,024	522,596	0.057	0.318	1.000	5.615
2,048	2,093,115	0.255	1.395	1.000	5.467



100% OF NODES WEIGHTED

		Minimum cut			
		Time (sec.)		Relative time	
n	m	pseudo-flow	push-relabel	pseudo-flow	push-relabel
128	8,043	0.001	0.003	1.000	2.167
256	32,561	0.004	0.017	1.000	4.300
512	130,548	0.019	0.077	1.000	4.118
1,024	523,278	0.053	0.336	1.000	6.312
2,048	2,094,890	0.276	1.448	1.000	5.238

The pseudoflow code is faster across all closure instances, and is scales better for the low density instances. The only observation we made from the operation counts was that while the number of pushes and relabels were roughly the same for both implementations, push-relabel performed an order of magnitude more arc scans than pseudoflow.

8. Conclusions

The highest label pseudoflow implementation is faster than push-relabel on all but the largest instances of one problem class (RLG-long). Our work is of significance because it was widely accepted until now that push-relabel was the fastest algorithm in practice for the maximum flow problem. Since the min-cut and max-flow are of interest both as stand-alone problems and as subroutines in other algorithms, our implementation could be used to efficiently solve a wide range of problems.

Among the different pseudoflow variants, the highest label algorithm was in general faster and more robust across different problem families than the lowest label algorithm. This is similar to past findings for push relabel (for example, by Ahuja et al. (1997)) in which the lowest label variant was found to be slower than the highest label variant.

We observed that the number of relabels performed by push-relabel was generally greater than that of pseudoflow, in spite of the global relabeling heuristic used in push-relabel. One possible explanation is that pseudoflow allows pushes along arcs where both ends of the arc have the same label. Such arcs would be inadmissible in push-relabel, needing at least one relabel in order to push flow along such an arc.

Endnotes

1. We thank an anonymous referee for proposing this description

Acknowledgments
The research of the second author was supported in part by NSF awards No. DMI-0620677 and CBET-0736232.

References

Andrew Goldberg's network optimization library. http://www.avglab.com/andrew/soft.html, accessed January 2007.

The first DIMACS algorithm implementation challenge: The core experiments. http://dimacs.rutgers.edu/pub/netflow/general-info/, accessed October 2004.

CATS: Combinatorial Algorithms Test Sets. http://www.avglab.com/andrew/CATS/gens/, accessed January 2007.

Ahuja, R. K, M. Kodialam, A. K. Mishra, and J. B. Orlin. 1997. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, **97**(3) 509–542.

Ahuja, R. K, T. L. Magnanti, and J. B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall.

Anderson, C., D. S Hochbaum. 2002. The performance of the pseudoflow algorithm for the maximum flow and minimum cut problems. UC Berkeley manuscript.

Anderson, R. J., J. C. Setubal. 1991. Goldberg's algorithm for maximum flow in perspective: a computational study. In *Network flows and matching: First DIMACS implementation challenge*, volume 12 of *DIMACS series in discrete mathematics and theoretical computer science*, 123–133.

Chandran, B., D. S. Hochbaum. Pseudoflow solver, accessed January 2007. http://riot.ieor.berkeley.edu/riot/Applications/Pseudoflow/maxflow.html.

Derigs, M., W. Meier. 1989. Implementing Goldberg's max-flow algorithm – a computational investigation. *ZOR – Methods and models of Operations research*, **33** 383–403.

Dinic, E. A. 1970. Algorithm for the solution of a problem of maximal flow in networks with power estimation. *Soviet Math. Doklady*, 11:1277–1280.

Ford, L. R., D. R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Math.*, 8:339–404.

Goldberg, A. V., B. V. Cherkassky. 1997. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410.

Goldberg, A. V., R. E. Tarjan. 1988. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940.

Goldfarb, D., M. Grigoriadis. 1988. A computational comparison of the Dinic and network simplex algorithms for maximum flow. *Annals of Operations Research*, 13:83–123.

Goldberg, A. V., S. Rao. 1998. Beyond the Flow Decomposition Barrier. *Journal of the ACM* 45(5): 783-797.

Hochbaum, D. S., J. B. Orlin. The pseudoflow algorithm in $O(mn \log \frac{n^2}{m})$ and $O(n^3)$. UC Berkeley manuscript. 2007. Submitted.

Hochbaum, D. S. 1997. The pseudoflow algorithm for the maximum flow problem. Manuscript, U C Berkeley, 1997 (Revised 2002). Extended abstract in The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem. Procdeedings of IPCO 98, June 1998. *Lecture Notes in Computer Science*, Bixby, Boyd and Rios-Mercado (Eds.), 1412, Springer, 325–337.

Hochbaum, D. S. 2008. The Pseudoflow algorithm: A new algorithm for the maximum flow problem. To appear in *Operations Research*.

Lerchs, H., I. Grossman. 1965. Optimum design of open pit mines. *Transactions, C.I.M*, 68:17–24.

Picard, J. 1976. Maximal Closure of a Graph and Applications to Combinatorial Problems. *Management Science*, 22:1268–1272.

Sleator, D. D., R. E. Tarjan. 1983. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391.