Simplifications and speedups of the pseudoflow algorithm

Simplifications and speedups of the pseudoflow algorithm

Dorit S. Hochbaum *
Department of Industrial Engineering and Operations Research,
University of California, Berkeley
email: hochbaum@ieor.berkeley.edu

James B. Orlin †
MIT Sloan School of Management
Cambridge, MA
email: jorlin@mit.edu

March 4, 2012

Abstract

The pseudoflow algorithm for solving the maximum flow and minimum cut problems was devised in Hochbaum in [13]. The complexity of the algorithm was shown in [13] to be $O(nm \log n)$. In [5] Chandran and Hochbaum demonstrated that the pseudoflow algorithm is very efficient in practice, and that the highest label version of the algorithm tends to perform best. Here we improve the running time of the highest label pseudoflow algorithm to $O(n^3)$ using simple data structures and to $O(nm \log(n^2/m))$ using the dynamic trees data structure. Both these algorithms use a new form of DFS implementation that is likely to be fast in practice as well. In addition, we give a new simpler description of the pseudoflow algorithm by relating it to the simplex algorithm as applied to the *maximum preflow* problem defined here. The interpretation of the generic pseudoflow algorithm as a simplex-like algorithm for the maximum preflow problem motivates the pseudoflow algorithm and highlights differences between the pseudoflow algorithm and the preflow-push algorithm of Goldberg and Tarjan.

Keywords:

Maximum flow, minimum cut, pseudoflow algorithm, data structure, dynamic trees.

*Research supported in part by NSF award No. DMI-0620677, CMMI-1200592 and CBET-0736232.
†Research supported in part by ONR grant N00014-05-1-0165.

1 Introduction

The *maximum s-t 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 bipartition 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, in a seminal study [9], established that the maximum flow value is equal to the minimum cut capacity and demonstrated a primal-dual relationship between the two problems.

The literature on maximum flow algorithms includes numerous algorithms. Most notable for efficiency among feasible flow (primal) algorithms is Dinic's algorithm [7], which has a running time complexity $O(n^2m)$, where n is the number of nodes and m is the number of arcs. Karzanov [14] and Malhotra et al. [16] independently improved the running time of Dinic's algorithm to $O(n^3)$. Goldberg and Rao [12] based their algorithm on an extension of Dinic's algorithm for unit capacity networks with run time of $O(\min\{n^{2/3}, m^{1/2}\}m\log(n^2/m)\log U)$, where U is the largest capacity of an arc. This algorithm currently has the best weakly polynomial running time for the maximum flow problem. Goldberg and Tarjan's push-relabel algorithm works with preflows, i.e., flows that permit excesses. (In principle, it could also be implemented to work with pseudoflows, which are flows that permit both excesses and deficits.) The push-relabel algorithm with dynamic trees implementation [17] has complexity of $O(mn\log\frac{n^2}{m})$, [11]. Using a different approach King et al. [15] devised an algorithm of complexity $O(mn\log_{m/(n\log n)} n)$. This algorithm currently has the best strongly polynomial running time for the maximum flow problem.

Hochbaum [13] developed the pseudoflow algorithm for solving the maximum flow and minimum cut problems. As indicated by its name, the algorithm maintained pseudoflows at each iteration. However, in this paper, the pseudoflow algorithm always maintains a preflow. The complexity of the pseudoflow algorithm was shown in [13] to be $O(nm\log n)$ using the dynamic trees data structure [17]. A highest label version of the algorithm was shown to have

the same complexity in [13].

Goldberg and Tarjan's push-relabel algorithm has been considered extremely efficient in practice on standard benchmark instances (see [1, 3, 6, 10]). The highest level variant of the push-relabel algorithm was found to have the best performance in practice at the time (see [10] and page 242 of [2]).

In [5], Chandran and Hochbaum showed that their best implementation of the pseudoflow algorithm is approximately a factor two faster than the best implementation of push-relabel on the standard benchmark instances. The highest label version of the pseudoflow algorithm performed best in practice. Boykov and Kolmogorov's algorithm was reported in [4] to perform better than the push-relabel algorithm for benchmark instances of vision problems. More recently, Fishbain et al. [8] presented an extensive study of the performance of various max-flow min-cut algorithms for vision problems, and demonstrated that Hochbaum's pseudoflow algorithm (HPF) is faster in practice than both Boykov and Kolmogorov's algorithm and the push-relabel algorithm [8].

We show here that a highest label-DFS (Depth-First-Search) version of the pseudoflow algorithm runs in $O(n^3)$ time. Moreover, the running time of the algorithm can be improved to $O(nm \log(n^2/m))$ using the dynamic trees data structure. We refer to these two algorithms as the DFS algorithm and the DT algorithm, respectively.

In addition to new complexity bounds, we give here a new and simpler description of the pseudoflow algorithm. We define the *maximum preflow problem*, which is to find a feasible preflow that maximizes that total amount reaching the sink of the graph. We then relate the pseudoflow algorithm's tree structure, maintained at each iteration, to a basic feasible solution maintained by a simplex algorithm applied to the maximum preflow problem. This description simplifies the presentation of the pseudoflow algorithm while simultaneously highlighting the distinctions between the pseudoflow algorithm and two other fundamental algorithms: the simplex algorithm for linear programming and the preflow-push algorithm of Goldberg and Tarjan [11].

In Section 2, we give definitions and preliminaries and properties of the maximum preflow problem. In Section 3, we present the pseudoflow algorithm by describing it as a simplex-like algorithm for the maximum preflow problem in that it moves from one basic solution to another (but not necessarily an adjacent basic solution). In Section 4, we show how to speed up the highest label push version of the pseudoflow algorithm so that it runs in $O(n^3)$ time. In Section

5, we show how to further modify the highest label push variant of Section 4 using the dynamic trees data structure so that it runs in $O(nm \log(n^2/m))$ time, thus matching the running time of the preflow-push algorithm of Goldberg and Tarjan [11].

2 Definitions and preliminaries

Let G be a graph (V, A), where the node set V has n nodes and the arc set A has m arcs. Let t be a designated sink node. We will define later the *maximum preflow* problem on G where for each node $i \in V \setminus \{t\}$ there is an associated nonnegative supply $b(i)$. For each arc $(i, j) \in A$, there is an associated finite capacity u_{ij} on arc flows. We assume without loss of generality that there is a directed path from each node to node t. Otherwise, nodes from which there is no path to t are deleted along with their incident arcs. We also assume that for every pair of nodes i and j, at most one of the arcs (i, j) and (j, i) is in A – that is, no anti-parallel arcs. This assumption is used to simplify the description of the residual network (which is defined later) and to simplify the notation in the paper. However, the algorithms and analysis all extend to the case in which (i, j) and (j, i) are both permitted to be present.

An undirected path between nodes i and j is often denoted by $P(i, j)$. A path $P(i_1, i_k)$ is represented by its sequence of nodes $(i_1, i_2, \ldots, i_k)$ so that for $j = 1, \ldots, k-1$ either $(i_j, i_{j+1}) \in A$ or $(i_{j+1}, i_j) \in A$.

A *rooted forest* of G is a subgraph that has no undirected cycles, and where each component of the forest has a designated node referred to as the *root node*. We make no assumptions on the directions of the arcs in a rooted forest.

We refer to each component of a rooted forest as a *branch*. Each branch of a rooted forest F has exactly one root node. Let root(i) denote the root node in the same branch of F as node i. If nodes i and j are in the same branch of F, we let $P^F(i, j)$ be the unique path in F from i to j. If nodes i and j are in different branches, then $P^F(i, j)$ is undefined. If i is not a root node, then the *parent* of node i is the node j that immediately follows node i on $P^T(i, root(i))$. We also write $j = p(i)$. If j is the parent of node i, then node i is a *child* of node j. We use $ch(j)$ to denote the ordered list of children of node j. Its order is from "left" to "right". For each root node i, $p(i) = \emptyset$.

We will often use the notation T for a rooted forest. The motivation for this slightly misleading notation is that the forest may be transformed into a tree by adding a super-root

node r_s connected to all the root nodes of T. In fact, the rooted forests in the maximum preflow problem (defined soon) correspond to trees in the maximum flow problem.

For a rooted forest T, let T_k be the rooted subtree induced by node k that consists of node k and all of its descendants in the same branch. We observe that if k is not the root, then T_k is not a branch. Suppose that each node has a label. (We will define the labels later.) A subtree of T_k that is rooted at a node k that has label d and that is restricted to only contain nodes of label d is denoted by T_k^d.

For $D_1, D_2 \subseteq V$ the set of arcs going from D_1 to D_2 is denoted by (D_1, D_2). The sum of the capacities on the arcs from D_1 to D_2 is denoted as: $C(D_1, D_2) = \sum_{i\in D_1, j\in D_2} u_{ij}$. A special case is the bipartition of V into S and $\bar{S}$ where $C(S, \bar{S}) = \sum_{i,j:i\in S, j\in\bar{S}} u_{ij}$. In general, $\bar{S}$ denotes the set $V\backslash S$.

A flow vector $\mathbf{f} = \{f_{ij}\}_{(i,j)\in A}$ is said to be *feasible* if it satisfies

(i) flow balance constraints: for each $j \in V\backslash\{t\}$, $\sum_{(i,j)\in A} f_{ij} = \sum_{(j,k)\in A} f_{jk}$ (i.e., inflow(j) = outflow(j)), and

(ii) capacity constraints: for all $(i, j) \in A$, the flow value is between the lower bound and upper bound capacity of the arc, i.e., $0 \le f_{ij} \le u_{ij}$.

For a flow vector $\mathbf{f} = \{f_{ij}\}_{(i,j)\in A}$, we let $f(D_1, D_2) = \sum_{i\in D_1, j\in D_2} f_{ij}$.

To define the maximum preflow problem on a graph $G = (V, A)$ where the set of nodes V contains a sink node t we introduce the concepts of excess of supply: For a given flow vector $\mathbf{x} = \{x_{ij}\}_{(i,j)\in A}$, satisfying the capacity constraints, the *excess* at node $i \in V \setminus \{t\}$ is

$$e(i) = e_{\mathbf{x}}(i) = b(i) + \sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij}$$

i.e., the excess of i with respect to flow $\mathbf{x}$ is $b(i)$ + inflow(i) − outflow(i). We will omit the reference to $\mathbf{x}$ whenever it is self evident.

A flow vector is said to be a *feasible preflow* if it satisfies the capacity constraints, but could violate the flow balance constraints with inflow$(j) \ge$ outflow(j). A preflow of 0 on all arcs is therefore a feasible solution for the maximum preflow problem.

The **maximum preflow problem** is to find a feasible preflow $\mathbf{x}$ that maximizes the flow into node t, that is, $\sum_{(i,t)\in A} x_{it}$.

Given a feasible preflow, an arc $(i, j) \in A$ is said to be *saturated* if $x_{ij} = u_{ij}$. An arc (i, j) is said to be a *residual arc* if $(i, j) \in A$ and $x_{ij} < u_{ij}$ or if $(j, i) \in A$ and $x_{ji} > 0$. For $(i, j) \in A$,

the *residual capacity* of arc (i, j) with respect to the flow $\mathbf{x}$ is $u^{\mathbf{x}}_{ij} = u_{ij} - x_{ij}$, and the *residual capacity* of the reverse arc (j, i) is $u^{\mathbf{x}}_{ji} = x_{ij}$. We will also denote the residual capacity of arc (i, j) as $r_{\mathbf{x}}(i, j) = u^{\mathbf{x}}_{ij}$. Let $A^{\mathbf{x}}$ denote the set of residual arcs for flow $\mathbf{x}$ in G, which are all arcs or reverse arcs with positive residual capacity. Let $G^{\mathbf{x}} = (V, A^{\mathbf{x}})$ denote the *residual network with respect to* $\mathbf{x}$.

The maximum preflow (max-preflow) problem is closely related to the well-studied maximum flow (max-flow) problem. In the max-flow problem the objective is to send as much flow as possible from a source node s to node t while keeping flow at all other nodes balanced. The value of the maximum flow is then the amount of flow along any cut $(B, \bar{B})$ for $\{s\} \subseteq B \subseteq V \setminus \{t\}$, and in particular it is equal to $e(t) = f(V \setminus \{t\}, \{t\})$. There are reasons that the maximum preflow problem merits separate attention.

(i) The max-flow problem on a graph $G = (V, A)$ containing a source node s and a sink node t can be solved by solving two maximum preflow problems. In the first problem, $b(s)$ is set to a large value, such as $b(s) = \sum_{(s,j)\in A} u_{sj}$. The solution to this problem determines a maximum preflow $\mathbf{x}^*$ of value $f^* = \sum_{(i,t)\in A} x^*_{it}$, reaching into t. The maximum flow value can then be at most f^*. Then the max-preflow problem can be solved again with $b(s)$ replaced by f^* and $b(i) = 0$ for all $i \in V \setminus \{s, t\}$ achieving a feasible balanced flow of value f^*. (Theorem 2.1 below establishes the correctness of this approach.) An alternative second problem is to replace $b(i)$ by $e(i)$ for all $i \in V \setminus \{s, t\}$, make s the sink node rather than t, and set $b(t) = 0$. Then solve the preflow problem of maximizing the excess of s along the arcs of the residual graph $G^{\mathbf{x}^*}$. (The correctness of this approach follows from the algorithm of Goldberg and Tarjan [11]. A third possibility is to attain a feasible balanced flow which is also the maximum flow by employing *flow decomposition* for sending the excesses of all nodes of $V \setminus \{s, t\}$ back to the source (see e.g., [13] for a description of flow decomposition in this context.)

(ii) The feasibility problem for the minimum cost flow problem can be represented by a single max-preflow problem. This is done by adding a sink node t and arcs from all demand nodes j going to t with capacity $u_{jt} = -b(j)$. The original problem is feasible if there is a max-preflow in the transformed problem that saturates the arcs adjacent to node t.

The reader is referred to [2] Chapter 7 for additional algorithms that rely on solving a max-preflow problem first.

A *t-cut* is a partition of the node set V into subsets S and $\bar{S}$, where $t \in \bar{S}$. We define the

preflow capacity of the t-cut $(S, \bar{S})$ to be $C(S, \bar{S}) + \sum_{j\in\bar{S}} b(j)$.

The following theorem is a consequence of the Max-flow Min-cut Theorem of Ford and Fulkerson for the maximum flow problem [9] as well as in Goldberg and Tarjan [11]. To establish the correctness of the max-preflow min-cut theorem on the graph $G = (V, A)$, we construct a corresponding graph $G_s = (V \cup \{s\}, A_s)$ for the max-flow problem. This is done by adding to G a source node s, adding arcs from s to all nodes $i \in \{j \in V \mid b(j) > 0\}$ of capacity $u_{si} = b(i)$ and setting all supplies to be 0.

Theorem 2.1 *The maximum value of a preflow is equal to the minimum capacity of a t-cut for the maximum preflow problem.*

Proof: Let $G = (V, A)$ be a network for the max-preflow problem and let $G_s = (V \cup \{s\}, A_s)$ be the respective network for the max-flow problem constructed as above. We say that a preflow $\mathbf{x}$ in G_s is *s-saturating* if every arc out of s is saturated. It is easy to see that there is a 1:1 correspondence between s-saturating preflows in G_s and preflows in G. (The correspondence is obtained by deleting node s and its incident arcs from G_s.) This correspondence maintains the value of the flow into node t.

There is also a 1:1 correspondence between s-t cuts in G_s and t-cuts in G. If $(S, \bar{S})$ is an s-t cut in G_s, then $(S\backslash\{s\}, \bar{S})$ is the corresponding t-cut in G. Moreover, the capacity of the s-t cut in G_s is equal to the preflow capacity of the corresponding t-cut in G.

We are now ready to establish the theorem. Suppose that a preflow $\mathbf{x}^*$ in G_s is s-saturating and it maximizes the flow v^* into node t. Goldberg and Tarjan [11] proved that v^* is the capacity of some s-t cut $(S^*\backslash\{s\}, \bar{S^*})$. (It is the cut that is induced at the end of Phase 1 of their algorithm.)

Let $\mathbf{x}'$ be the preflow in G corresponding to $\mathbf{x}^*$. Then its flow into node t is v^*. Moreover, the correspondence between cuts shows that the capacity of $(S^*, \bar{S^*})$ is equal to the preflow capacity of $(S^*\backslash\{s\}, \bar{S^*})$. This completes the proof. ■

3 The pseudoflow algorithm for the maximum preflow problem

In this section, we present the pseudoflow algorithm in a manner that closely parallels the simplex algorithm for the maximum preflow problem. In particular, the algorithm moves from one basic solution to another basic solution by modifying flows on paths. But the iterations

do not correspond to single pivots, and the basic solution following an iteration is not adjacent in the polytope to the basic solution at the beginning of an iteration. The generic pseudoflow algorithm runs in $O(nm)$ iterations and in $O(n^2m)$ total time.

In this section, we describe first the basic solution construction. We then present the generic pseudoflow algorithm as in [13] and prove its correctness and complexity. We then present a version in which the order of the operations is modified slightly and show that it is equivalent to the generic version. This modified version is the basis of the more efficient algorithms presented in the following sections.

3.1 Basic preflows

A *basis* of the max-preflow problem consists of a 4-tuple (T, L, U, R), where (T, L, U) is a partition of the arc set A, and R is a subset of V. The subset T is a spanning forest; that is, it contains no cycles (in the undirected sense), and each node either has an endpoint in the arc set T or is considered a singleton in the forest; L is a subset of arcs whose flows are at their lower bounds, i.e., 0; U is a subset of arcs whose flows are at their upper bounds; and R is the subset of root nodes, which contains node t and which has exactly one node per component of T.

The *basic preflow* induced by basis (T, L, U, R) is the unique arc flow vector $\mathbf{x}$ such that

(i) $x_{ij} = 0$ for $(i, j) \in L$,

(ii) $x_{ij} = u_{ij}$ for $(i, j) \in U$, and

(iii) $e(i) = 0$ for $i \notin R$.

A basic preflow is *feasible* if it satisfies $0 \leq x_{ij} \leq u_{ij}$ for all $(i, j) \in T$ and if $e(i) \geq 0$ for all $i \in R$.

Each component of T in a given basis 4-tuple (T, L, U, R) is a branch that contains a unique root node. Singleton components are branches that consist of the root of the branch only.

The algorithm will initialize with the basic feasible preflow constructed by letting $T = \emptyset, L = A, U = \emptyset$, and $R = V$. In this initial preflow, all branches are singletons, and $e(i) = b(i)$ for each $i \in V \backslash \{t\}$.

3.2 The generic pseudoflow algorithm

An elementary procedure for sending flow on an arc (i,j) is $\mathsf{Push}(\mathbf{x},i,j)$. This operation applies to arcs in T and to an out-of-tree arc that will serve as a "merger arc". Let $\mathbf{x}$ be the current flow vector on network G.

procedure $\mathsf{Push}(\mathbf{x},i,j)$
  begin
    send $q := \min\{e(i), r_{\mathbf{x}}(i,j)\}$ units of flow on (i,j);
    {update excess and the flow vector}:
      $e(i) := e(i) - q$, $e(j) := e(j) + q$;
      if $((i,j) \in A)$ **then** $x_{ij} := x_{ij} + q$;
      else $x_{ij} := x_{ij} - q$;
      end if
    if $(q = r_{\mathbf{x}}(i,j))$ **then**
      delete (i,j) from T; {split i and j}
      add node i to R;
      return *saturating*;
    else return *nonsaturating*;
    end if
  end

If the flow q sent on (i,j) is $r_{\mathbf{x}}(i,j)$, we say that the push is *saturating*. Otherwise, we say that it is *nonsaturating*. In the generic algorithm, if $\mathsf{Push}(\mathbf{x},i,j)$ is saturating, then i becomes a root of a branch.

Let $\mathbf{x}$ be a basic feasible preflow. For each node $i \in V$, there is a *distance label* $d(i)$ satisfying the following:

Distance conditions:

(i) $d(t) = 0$;

(ii) for all $(i,j) \in A^{\mathbf{x}}$, $d(i) \le d(j) + 1$;

(iii) if $i \notin R$, then $d(i) \ge d(p(i))$ (referred to as *monotonicity*).

Because of the monotonicity, for each branch rooted at $i \in R$, the set of nodes in T_i of label $d = d(i)$ form a connected subtree in T_i, which is denoted by T_i^d. To see that T_i^d is connected,

note that otherwise there is a path from i to a descendant j of label d with an intermediate node of distance label greater than d. But this violates the monotonicity along such path that requires that labels can only go up with the distance from i.

An arc $(i, j) \in T$ is called *neutral* if $d(i) = d(j)$. A path in T is called *neutral* if every arc on the path is neutral. Whenever we refer to a neutral arc, we are referring to an arc of T. An arc $(i, j) \in A^{\mathbf{x}}$ is called *admissible* if $d(i) = d(j) + 1$. A node i is called *active* if $i \neq t$, $e(i) > 0$, and $d(i) < n$. We say that an arc (i, j) is *scanned* if it has been checked for admissibility, i.e., for whether $d(i) = d(j) + 1$, since the last time that the distance label of node i was increased. We say that a node i is *scanned* if all residual arcs that originate at i are scanned. A node i of label d is said to be d-visited if i has been scanned and it is determined that there is no admissible arc originating at i.

Let i be an active root node. An arc $(j, j') \notin T$ is called a *merger arc with respect to node* i if the following are true:

(i) $\text{root}(j) = i$;

(ii) the path $P^T(j, i)$ is neutral;

(iii) (j, j') is admissible; we denote the set of merger arcs with respect to node i as $\text{WRT}(i)$.

We are now prepared to describe the generic Pseudoflow Algorithm. Let $\mathbf{x} = \{x_{ij}\}_{(i,j)\in A}$ denote the flow vector in the network.

The Generic Pseudoflow Algorithm

begin

{initialize with a basic feasible preflow}:

$\forall (i, j) \in A$, $x_{ij} = 0$;

$T = \emptyset, L = A, U = \emptyset, R = V$;

{In the following (T, L, U, R) will be maintained implicitly}

set $d(t) = 0$ and $d(i) = 1$ for all $i \in V \setminus \{t\}$;

while (there is an active root node) **do**

select an active root node i;

if ($\text{WRT}(i) = \emptyset$) **then** {there is no merger arc with respect to node i}

let $T_i^{d(i)}$ be the set of nodes connected to node i by a neutral path in T_i;

{*relabel*}:

for (all j in $T_i^{d(i)}$) **do**

$d(j) := d(j) + 1$;
end for
else {there is a merger arc with respect to node i}
let $(j, j') \in \mathrm{WRT}(i)$;
{*invert*}:
for (each residual arc (v, w) on the path from $r_j := root(j)$ to j) **do**
$p(v) := w$;
end for
{*merge*} $p(j) := j'$; {add (j, j') to T}
{now $root(j) = root(j')$}
{*PushPath*}:
for (each residual arc (v, w) on the path from r_j to $root(j')$) **do**
$isSaturating := \mathsf{Push}(\mathbf{x}, v, w)$;
if ($isSaturating = saturating$) **then**
$p(v) = \emptyset$; {set node v as a root}
end if
end for
end if
end while
output $\mathbf{x}$ as a feasible basic preflow and its corresponding basis (T, L, U, R);
end

In Figure 1, the major operations of merger, inversion, PushPath with $\mathsf{Push}(\mathbf{x}, v, w)$ saturating, are illustrated. In (b), j becomes the ancestor of all nodes of T_{r_j} in (a), w is the parent of v, and r_j is the descendant of v, w and j. T_{r_j} in (a) is now part of $T_{r'_j}$ in (b) rooted at $r_{j'}$. In (b), the unique path from j to r_j is given in three sections, the path p_w from j to w; arc (w, v) and the path p_v from v to r_j.

3.3 Correctness and complexity of the generic pseudoflow algorithm

Theorem 3.1 *The generic pseudoflow algorithm solves the maximum preflow problem in $O(nm)$ iterations and $O(n^2m)$ total time.*

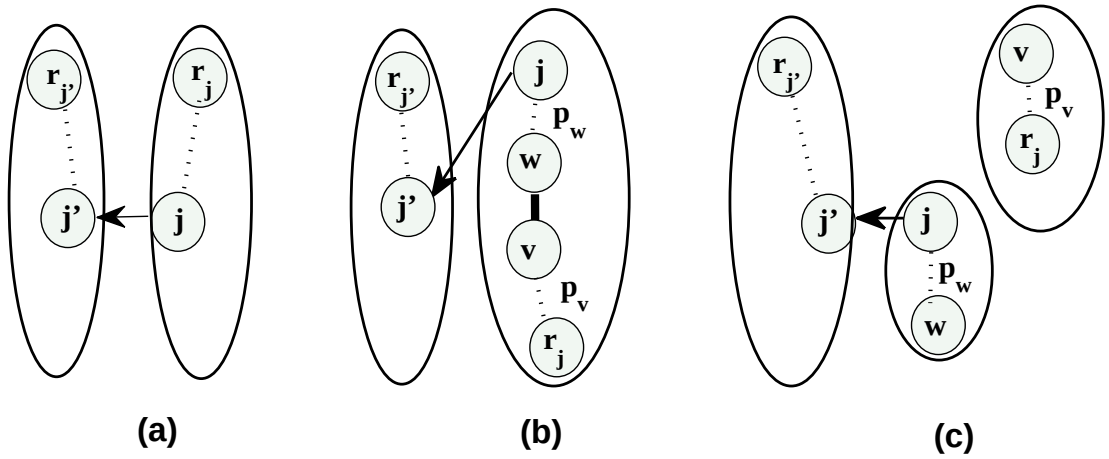

Figure 1: (a) Branches prior to merger on arc (j, j') (b) The invert process (c) Pushing flow along the path from r_j to r'_j where $\mathsf{Push}(\mathbf{x}, v, w)$ is saturating.

Proof: To verify correctness, we claim first that the distance conditions are satisfied after each call to PushPath. The distance conditions may be affected by the inversion process and by relabeling. The inversion procedure maintains the validity of the distance labels since the section of the path that is inverted consists of neutral arcs (of equal labels). After merger with the arc (j, j'), $p(j) = j'$ and $d(j) = d(j') + 1$. In particular, monotonicity is retained.

Relabeling occurs if there are no merger arcs with respect to node i. In that case, there are no admissible arcs out of $T_i^{d(i)}$, and the distance conditions are satisfied after the relabeling.

When the algorithm terminates, there are no active nodes, although it is possible that there are nodes at level n that have excess. Let $S = \cup_{\{i|e(i)>0\}} T_i$. Then by [13], the set S is the source set of a minimum cut and the preflow is therefore maximum. (It is further shown in [13] that S is also a *minimal* source set of a minimum cut.) To establish this in a self-contained proof, we use the construction in Theorem 2.1 of the graph G_s. Nodes that have positive excess in G, $V^+ = \{i \,|\, e(i) > 0\}$, have in the graph G_s positive flows on the respective arcs from s, $(\{s\}, V^+)$. Therefore all residual arcs in $(V^+, \{s\})$ have positive residual capacity, and the nodes in $V^+ \cap S$ can send their excess back to s along these arcs. This leaves the flow on the cut arcs $(S, \bar{S})$ unchanged and thus still saturated. Also, the amount of flow reaching t does not change. From Theorem 2.1 the cut $(S, \bar{S})$ is minimum in G and thus the preflow corresponding to it is maximum.

To establish the complexity, we show that the work to find merger arcs, across all iterations, is $O(nm)$; that the complexity of relabeling, across all iterations, is $O(n^2)$; and finally, that the number of iterations is $O(nm)$ and the pushing flow work per iteration is $O(n)$.

The complexity of finding merger arcs per node is the work for scanning the node's neighbor list. Per given label of a node, this complexity is the number of outgoing arcs from the node, i.e., the outdegree of the node, in the residual graph. Since the sum of outdegrees of all nodes is $O(m)$ and the number of possible labels is $O(n)$, it follows that the complexity of finding merger arcs is $O(nm)$.

Since labels of nodes can only increase and labels are bounded by n, each node can be relabeled $O(n)$ times and the complexity of relabeling is $O(n^2)$.

In every **while** -iteration, there is exactly one merger execution. We will show that each arc (u, v) can become a merger arc $O(n)$ times, at most one time for every distance label that u can have. To evaluate the number of merger iterations, we recall that if (u, v) is a merger arc, then it is admissible and $k = d(u) = d(v) + 1$. There are two cases to consider: one is when the push on the merger arc is saturating and the second when it is nonsaturating.

1. $\mathsf{Push}(\mathbf{x}, u, v)$ **saturating**: Arc (u, v) then becomes out-of-tree and residual in the direction (v, u). Once the arc left the tree it can join again only as a merger arc in the direction that has positive residual capacity. In order for (v, u) to become a merger arc, it must be true that $d(v) = d(u) + 1 \geq k + 1$. Therefore the label of v must increase by at least two units, from $k - 1$ to $k + 1$ before (v, u) can serve as merger arc.
2. $\mathsf{Push}(\mathbf{x}, u, v)$ **nonsaturating**: Arc (u, v) is then in the tree T following the merger, and needs to leave the tree before it can serve as merger arc again. Thus there must be a saturating push on (u, v) before it can leave the tree. This is then the case of $\mathsf{Push}(\mathbf{x}, u, v)$ saturating.

We conclude that (u, v) can only serve as merger arc at most $O(n)$ times and the total number of merger iterations is $O(nm)$.

During each iteration, the call to *invert* and *PushPath* can be carried out in $O(n)$ time, for a total running time of $O(n^2m)$. All other steps are bounded in time by $O(nm)$. Thus the complexity of the algorithm is $O(n^2m)$ as claimed. ■

At a micro level, the pseudoflow algorithm would be a special case of the preflow-push algorithm of Goldberg-Tarjan except for one important detail. The pseudoflow algorithm sends flow on the arcs in $P^T(r_j, root(j'))$, and many of these arcs are neutral. In the preflow-push

algorithm, all flow is sent only on (non-neutral) arcs (i, j) in which $d(i) = d(j) + 1$.

Chandran and Hochbaum [5] showed that the pseudoflow algorithm is faster empirically than the preflow-push algorithm. We conjecture that the flexibility to send flow on neutral arcs is important, and leads to an overall speedup of the algorithm.

3.4 An equivalent down-up version of the generic algorithm

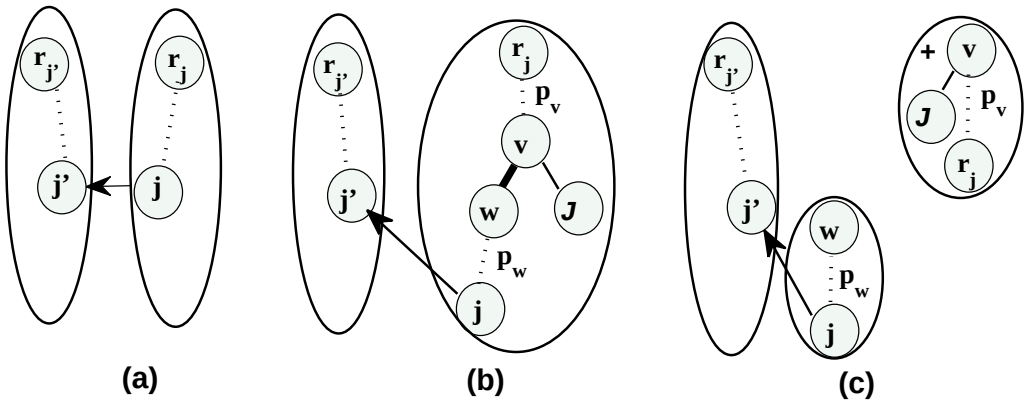

Figure 2: (a) Branches prior to merger on arc (j, j') (b) *PushPath-down* on the path from r_j to j encounters first saturating push, $\mathsf{Push}(v, w)$ saturating (c) v becomes a root node and the path section P_v from r_j to v is inverted. Pushing flow continues from w.

A minor variant in the generic algorithm is to delay the inversion until either the push on the admissible path is complete, or else there is a saturating push. This means that for a merger arc (j, j') the process of pushing on the path from $r_j = root(j)$ to j is in the down direction in the branch T_{r_j}, whereas the push on the portion from j to j' and subsequently to $r_{j'} = root(j')$ is done in the up direction. The reason for noting that this down-up version is equivalent to the generic algorithm is that it makes a difference in the DFS variant of the algorithm, by preserving the "down" direction. In the DFS variant subsequent pushes are executed in the down direction only, while delaying the pushes in the up direction to the end of a "phase".

The down-up generic algorithm is illustrated in Figure 2. In (b), J indicates all descendants of v other than T_w, i.e., $J = (T_v \setminus \{v\}) \setminus T_w$. The complexity of this down-up version is the same as that of the generic algorithm.

4 A speed-up for the highest label DFS variant - the DFS algorithm

In this section, we consider the variant of the pseudoflow algorithm in which we always select an active node whose distance label is maximum. If implemented in a straightforward manner, this variant has a worst case running time of $O(n^2m)$, which is the same as the generic pseudoflow algorithm. In order to speed it up to $O(n^3)$, we make several changes to the down-up generic algorithm. One change is that we skip the inversion following a saturating push resulting in excess at intermediate stages being stored at nonroot nodes. A second change is a modification of the down-up generic algorithm in which *PushPath-down* is executed repeatedly (and without inverting) while postponing the process of pushing the flow further, *PushPath-up* in the generic algorithm. This postponement is called *delayed normalization* in [13]. In this implementation, at intermediate stages, the set of arcs T may not correspond to the basic arcs of a basic preflow. Further changes are in the implementation of the search for merger arcs and the pushing of the flow jointly within a DFS process. These are described below in detail.

A "phase" consists of a series of pushes, where pushes originate from active nodes at the highest distance label (also referred to as *level*), which we denote here as d'. A phase ends when the highest level of an active node is no longer d'. A phase d' consists of two parts.

The first part of phase d' is called PushDown(d'). We seek out merger arcs (j, j') where root(j) is an active node at level d' and j is at level d' as well. When locating a merger arc (j, j'), we push flow from root(j) to j and then push flow on (j, j'). Unlike the generic down-up algorithm, Push($\mathbf{x}, v, w$) saturating is not followed by inversion of the component containing v, and consequently v has positive excess while being a nonroot node (see Figure 3(c)). Secondly, we *do not* push flow from node j' to its root $r_{j'}$ until the next part of the phase. This possibly leaves j' as a nonroot node with positive excess. We continue PushDown(d') until there are no more merger arcs originating at level d'. At termination of PushDown(d'), there are two types of nodes that are nonroot nodes with excess. The level of one type of nodes is d'. All these d'-level nodes were relabeled at termination of PushDown(d'). The other type of nodes are all at level $d' - 1$. All these $(d' - 1)$-level nodes were heads of merger arcs during PushDown(d'). The algorithm is illustrated in Figure 3, where J in Figure 3(c) indicates the descendants of v other than T_w.

The second part of phase d' is called FinalPush(d'). During FinalPush(d'), we push flow from

nonroot nodes (at level $d'-1$) with positive excess until all excess resides entirely at root nodes.

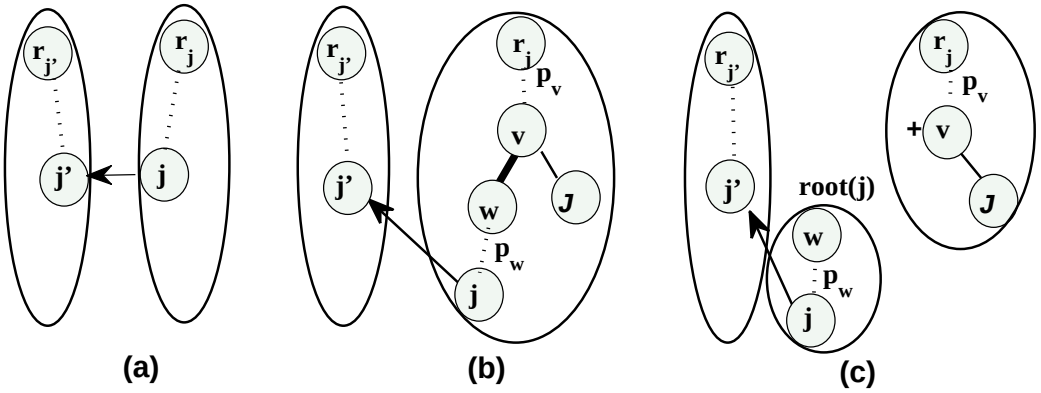

Figure 3: (a) Branches prior to merger on arc (j, j') (b) PushPath on the path from r_j to j encounters first saturating push, $\mathsf{Push}(v, w)$ saturating (c) DFS continues from node w and v is a nonroot with excess. w is now $root(j)$ and it has positive excess.

In $\mathsf{PushDown}(d')$, we use Depth First Search (DFS) order while searching for a merger arc. However, once a node has been d'-visited, we immediately push flow to its child, even if a merger arc is ultimately not found in that subtree. Likewise, the process of backtracking is accompanied by pushing excess back from a node to its parent. The relabeling of nodes is integrated with the DFS process.

We are now ready to describe the detailed implementation of DFS in $\mathsf{PushDown}(d')$ for d' the highest label of an active (root) node. Let the set of active root nodes of highest label d' be $\{r_1, \ldots, r_\ell\}$. We arrange this set in a DFS list in LIFO order as List $= (r_1, \ldots, r_\ell)$, where r_1 is the top element in List.

Let $V(d')$ be the set of nodes that are labeled d' and that are active or have as root an active node of label d'. Let $T(d')$ be the set of the neutral arcs within $V(d')$. Thus $T(d')$ forms a forest that contains subtrees of all the active branches with roots of label d'. For any node j, we use $ch_{d'}(j) \subseteq ch(j)$ to denote the ordered list of child node i of j such that $(j, i) \in T(d')$. Although $T(d')$ remains a forest throughout phase d', the set of nodes in $V(d')$ is shrinking as nodes get relabeled. A node can get relabeled only if it is a leaf of the forest $T(d')$. A leaf node that has been d'-visited and has children only of label $d'+1$ is relabeled. This restriction on relabels ensures that the monotonicity of the distance labels is satisfied throughout the algorithm, and the intersection of each branch of T with the forest $T(d')$ is a connected component.

Let $N(j)$ be a pointer to the next unscanned out-of-tree arc originating at j. If all such arcs have been scanned, then $N(j) = \emptyset$ and j is d'-visited. In this case, there is no need to scan node j again while its label is still d'. The value of $N(j)$ is reset to the first neighbor on the list whenever node j is relabeled.

During PushDown(d'), for Push($\mathbf{x}, v, w$) saturating it is w that is becoming a root node in R, as in Figure 3(c), unlike the generic algorithm. That leaves node v with nonnegative excess as a nonroot node, and v remains on List if its excess is positive and w is added to List in the first position. That is, w serves as next node to be processed in the DFS procedure.

The only inversion that takes place in the DFS algorithm is when there is a nonsaturating push on a merger arc (j, j') and the nodes in the branch containing j join the branch rooted at root(j'). In that case, the path between node j and root(j) is inverted.

Throughout the algorithm, List contains all active nodes of $V(d')$–that is, all positive excess nodes of label d' are in List. Operation TopL(List) picks the top node on List but does not remove it from List, while operation PopL(List) picks the top node on List and removes it from List. Operation PushL(i) adds node i to the top on List (LIFO order). Before the first call to phase d' all nodes $j \in V(d')$ satisfy $N(j) \neq \emptyset$ (they are d'-unvisited at this phase).

Let List:= $(r_1, \ldots, r_\ell)$ initially. List is a global data structure throughout the procedure.

procedure PushDown(d')
  begin
    if (List$\neq \emptyset$) **then**
      j :=TopL(List); {excess $e(j) > 0$}
      if ($N(j) \neq \emptyset$ and there is a merger arc (j, j') originating at j) **then**
        $isSaturating$:= Push($\mathbf{x}, j, j'$);
        if ($isSaturating = saturating$ and $e(j) > 0$) **then**
        {do nothing, node j is still on List and node j' is a nonroot node with excess}
        else
          PopL(List); {remove node j from List}
          {*invert*}:
            for (each residual arc (v, w) on the path from $root(j)$ to j) **do**
              $p(v) := w$;
            end for
          {*merge*} $p(j) := j'$; {add (j, j') to T}

```
            end if
            PushDown(d′);
        else {j is d′-visited and excess is pushed down to a descendant of j}
            for (each child of j: i ∈ ch_{d′}(j)) do
                isSaturating := Push(x, j, i);
                if (e(j) = 0) then
                    PopL(List); {remove node j from List}
                end if
                PushL(i); {add node i on List}
                if (isSaturating = saturating) then
                    p(i) := ∅; {set node i as a root}
                end if
                PushDown(d′);
            end for
            {backtracking}:
                {relabel}:
                    d(j) := d(j) + 1;
                    if (j = TopL(List)) then
                        PopL(List); {remove node j from List}
                    end if
                {return excess to parent}:
                    if (p(j) ≠ ∅) then
                        isSaturating := Push(x, j, p(j));
                        PushL(p(j)); {add node p(j) on List}
                        if (isSaturating = saturating) then
                            p(j) := ∅; {set node j as a root}
                        end if
                    end if
        end if
    end if
end
```

When PushDown(d') terminates, there are some active nodes of label $d'+1$ but no merger arcs with respect to these nodes; there are no active nodes of label d'; and there are nonroot nodes with excess that have label $d'-1$.

We note that in every call to PushDown(d'), we process each arc of $T(d')$ at most twice: Once in the forward DFS involving a push in the downward direction from a node to its child; the second time an arc is processed is in the backtrack process where the excess is pushed back to the node's parent (unless the node is a root). The backtrack push from node i to $p(i)$ is associated with node i being relabeled $d'+1$ and consequently removed from $V(d')$. Once the invert process takes place, all these relabeled nodes, and their subtrees in $T(d')$ have been removed from $V(d')$ as they are no longer in a subtree rooted in an excess node of label d'.

In the procedure FinalPush(d') excess is pushed from the nonroot nodes with excess, each of which necessarily has the label $d'-1$, to their respective roots. We are interested in the subtrees of T in which all the leaves are nonroot nodes with positive excess. The set of these subtrees is $W(d'-1) = \{v : v$ is in a branch containing an active nonroot node j' of label $d'-1$ and v is on the path from j' to root$(j')\}$. We note that for all $v \in W(d'-1)$, $d(v) \leq d'-1$ because of the monotonicity.

In this procedure we refer to "reverse topological order" in a rooted tree as the topological order that enforces a label of a node v to be less than that of its parent $p(v)$:

procedure FinalPush(d')
  begin
    order the nodes of $W(d'-1)$ in reverse topological order;
    for (each node v in $W(d'-1)$ (taken in the reverse order) such that $v \notin R$) **do**
      $isSaturating :=$ Push($\mathbf{x}, v, p(v)$);
      if ($isSaturating = saturating$) **then**
        $p(v) := \emptyset$; {set node v as a root}
      end if
    end for
  end

We now describe the $O(n^3)$ version of the pseudoflow algorithm.

Algorithm Highest Label DFS Pseudoflow

begin

Initialize();

while (there are any active nodes) **do**

let d' be the highest label of an active node;

{let $r_1, \ldots, r_\ell$ be the initial active d'-labeled root nodes}

initialize List as List:= $(r_1, \ldots, r_\ell)$;

for (each node $j \in V$) **do**

set $N(j)$ to the first unscanned out-of-tree arc originating at j;

end for

PushDown(d');

FinalPush(d');

end while

end

The procedure Initialize() creates the initial data structures as well as the initial basis. This includes setting the preflow values to be equal to 0 on all arcs. The branches are then all singletons and each node is a root of its own branch. The nodes with excess are those that have positive supply. The distance labels of nodes in $V\backslash\{t\}$ are set to 1.

Theorem 4.1 *The highest label DFS pseudoflow algorithm solves the maximum preflow problem in $O(n^3)$ time.*

Proof: As in the generic pseudoflow algorithm and the proof of Theorem 3.1, the procedure Highest Label DFS Pseudoflow terminates when there are no active nodes, that is, all nodes in $V\backslash\{t\}$ with positive excess have a distance label of n. In this case, it ends with a maximum preflow and a minimum cut.

To establish the complexity, we first bound the number of phases, as was done in [11]. We separate phases into *increasing phases*, in which the maximum distance label d' increases after the phase, and *decreasing phases*, in which d' decreases after the phase. The latter happens when all branches with active roots labeled d' were merged into other branches (no split occurred in the final Pushdown phase.) The number of increasing phases is $O(n^2)$, and the total increase in d' following these phases is also $O(n^2)$. Thus, the total decrease in d' subsequent to decreasing

phases is $O(n^2)$, and so the number of such phases is $O(n^2)$. Thus the number of phases is $O(n^2)$.

We next bound the time spent in each phase. FinalPush(d') consists of $O(n)$ pushes and runs in $O(n)$ time. Thus, the total time spent in all phases on FinalPush is $O(n^3)$.

As before the total time spent in all phases on finding merger arcs is at most $O(mn)$. Also, similar to the complexity of relabeling in the generic algorithm, there are up to n values for d'. Therefore the total time spent in all phases on *relabel* is $O(n^2)$. All we need to show now is that the time for PushDown(d') is $O(n)$ per phase.

Claim 4.1 *The complexity of* PushDown(d')*, excluding finding merger arcs, is $O(n)$ per phase. More precisely:*
(i) Each arc gets excess pushed from parent to child at most once per phase;
(ii) Each arc gets excess returned to parent from child at most once per phase;
(iii) Each arc is inverted at most once per phase;
(iv) Each node is relabeled at most once.

Proof: During PushDown(d'), an arc (j,i) can be visited at most twice in the DFS process and once in the *invert* process. The first time, j is selected as an active node on List and i is the next child of j in $ch_{d'}(j)$. In that case the push is in the downwards direction. If the push is saturated, then arc (j,i) is removed from T and from $T(d')$ and will not be included in any subsequent $T(d')$.

The second time that arc (j,i) is visited is in the backtrack process where node i is a leaf in the DFS tree (indicated by having no children in $T(d')$). In that case node i gets relabeled and then returns its excess to its parent, $j = p(i)$. The arc (j,i) therefore cannot participate again in $T(d')$ in any subsequent phase - the label of i is $d'+1$ and hence (j,i) is no longer neutral for d'.

The only other form of processing arcs is in the *invert* process. This happens when a merger arc (j,j') is found and $r(j,j') \geq e(j)$. In that case, the entire branch rooted at root(j) joins the branch rooted at root(j') and no longer participates in this phase's PushDown(d').

■

With the claim, we thus established that the total run time of the highest label DFS algorithm consists of:
$O(n^3)$ for all calls to FinalPush,

$O(nm)$ for finding mergers in all calls to PushDown,

$O(n^2)$ for all calls to Relabel, and

$O(n^3)$ for all calls to PushDown, excluding finding mergers.

The total is therefore $O(n^3)$ as claimed. ■

5 An $O(nm \log(n^2/m))$ implementation of the DFS algorithm – the DT algorithm

In this section, we describe a dynamic tree implementation of the DFS algorithm of the previous section that runs in $O(nm \log(n^2/m))$ time. We call this algorithm the DT algorithm, where DT stands for Dynamic Trees.

The dynamic trees data structure was developed by Sleator and Tarjan [17]. They used this data structure to develop an $O(nm \log n)$ algorithm for the maximum flow problem. Goldberg and Tarjan [11] showed how to use the dynamic trees data structure to speed up the preflow-push algorithm for the maximum flow problem to $O(nm \log(n^2/m))$ time. Our approach follows similar reasoning; however, as is typical with dynamic tree algorithms, there are a number of technical details that need to be worked out in order to achieve this running time.

Dynamic trees rely on partitioning the arcs of a tree into subtrees. In order to clearly differentiate the subtrees of dynamic trees from branches, we will refer to them as *D-trees*. Each D-tree has a special node called the *D-root*. In our implementation of the dynamic trees data structure, each root node of T will be a D-root of a dynamic tree, but the converse is not true. All nodes in one D-tree are required to have the same label. We let D-root(j) denote the root of the D-tree containing node j. Each D-tree is a connected subgraph of T.

With the dynamic trees implementation, the main distinction compared to the DFS algorithm is that we keep track of d'-visited nodes throughout all phase calls with label d' so they are skipped over. This is done with the help of a flag, which is set to 0 initially, and to 1 when $N(j) = \emptyset$ (d'-visited). In the DFS algorithm d'-visited nodes can be visited again in subsequent calls to phase d' only to verify that they are d'-visited. This can result in additional run time of $O(n)$ per phase. In contrast, the DT algorithm *skips* over such nodes. The skip is triggered once the top node on List is found to be flagged as d'-visited. In that case, the skip operation is to proceed from node i to unvisited$_{d'}(i)$, defined to be a left-most descendant of i that has flag$_{d'}$ value 0 and hence has not been visited.

The total work for PushDown is $O(mn)$ for pushing down on arcs and backtracking and scanning for all merger arcs. Each D-tree will have at most q nodes. We refer to a D-tree as *large* if it has at least $q/2$ nodes, and refer to it as *small* otherwise. Thus, there are at most $2m/n$ large D-trees at any time. With the push-down and backtrack operations, we will also include an update of the dynamic trees to ensure that most of them are *large*, and that will require a complexity of $O(\log q)$ per operation. This, plus the complexity of the skips and the inversions will be shown to have total complexity of $O(mn\log q)$.

For FinalPush, the main challenge in adapting the DFS version to the DT version is in the reverse topological order in which the nonroot excess nodes are processed. This requires setting the topological order on D-roots of the respective D-trees.

For each node j of the D-tree that is not a D-root, we let Value-up(j) be the residual capacity $r_{\mathbf{x}}(j,p(j))$. If node j is a D-root, then Value-up(j) $=\infty$. Similarly, we let Value-down(j,i) ($i\in ch(j)$) be the residual capacity $r_{\mathbf{x}}(j,i)$. Note that Value-up(j) is associated with a node j and its unique path to an ancestor, while Value-down(j,i) is associated with the arc from a node j to one of its children i. If node j is a leaf of a D-tree, then Value-down($j,\emptyset$) $=\infty$. For each node j, we let size(j) denote the number of nodes in the D-tree containing node j. For each node j we have a second value, flag$_{d'}(j)$, which assumes the value 0 if node j has not completely been scanned for admissible arcs at phase d', i.e., $N(j)\neq\emptyset$, and the value 1 if node j is d'-visited, i.e., $N(j)=\emptyset$. Each node j has the value, flag-DT$_{d'}(j)$, which is 0 if node j has a *descendant* i (i can be equal to j), contained in the same D-tree, with flag$_{d'}(i)=0$. Thus flag-DT$_{d'}$(D-root)$=1$ if all nodes in the D-tree rooted at D-root have been d'-visited, and equals 0 otherwise. One more label value associated with each node is unvisited$_{d'}(j)$. This is a label of node j with flag-DT$_{d'}(j)=0$ indicating a descendant i of node j that is the closest to j in the same D-tree with flag$_{d'}(i)=0$. If flag$_{d'}(j)=0$, then this label is j; else, it is a left-most descendant i of j such that all nodes on the path from $p(i)$ to j have flag value 1 and flag$_{d'}(i)=0$. It is shown in the Appendix that this label is maintained with constant number of dynamic tree operations per update.

Dynamic trees support various operations. We adapt these operations and present them in the context used here. One basic operation, find-minvalue(j), finds the minimum value of a node on the path in T from j to D-root(j). This operation is omitted from the list of operations since it is not used here. On the other hand, operations such as find-first-down(i,w,val) do not appear explicitly on the list of basic operations, but can be shown (as was shown e.g.,

in [13]) to be a dynamic tree operation that can be implemented, on a D-tree of size q, in $O(\log q)$ complexity. All of the operations below are either part of the original dynamic trees data structure, or can be implemented in a straightforward manner using dynamic trees.

(i) D-root(j). Finds the D-root of the D-tree containing node j.

(ii) Find-first-up(i, w, val). Finds a node j on the path from i to w (that could be its D-root(i)) that is closest to node i and such that Value-up(j) $\leq val$, or returns w otherwise.

(iii) Find-first-down(i, w, val). Finds the node j on the path from i to w in D-tree(i) that is closest to node i and such that Value-down($j, c(j)$) $\leq val$, where $c(j)$ is the unique child of j on the path, or returns w otherwise.

(iv) FindSize(j). Returns size(j).

(v) AddValue(i, w, val). Adds the real number val to the flows of all residual arcs (j, j') on the path in T from node i to node w, and updates Value-up and Value-down for all internal nodes of the path by subtracting or adding the value of val: If i is a descendant of w, then Value-up of (i) and all internal nodes is reduced by val and Value-down of arc $(w, c(w))$ and of all the internal arcs is increased by val. The subroutine simultaneously decreases $e(i)$ by val and increases $e(w)$ by val. The case when i is an ancestor of w is analogous.

(vi) Link(i, j). This operation assumes that i and j belong to two different D-trees. It merges the D-tree containing node i with the D-tree containing node j, lets $p(i) = j$, and sets the root of the merged tree to D-root(j). It sets Value-up(i) to $r_{\mathbf{x}}(i, j)$.

(vii) Cut(j). This operation breaks the D-tree containing node j into two D-trees by deleting the arc $(j, p(j))$. Node j becomes the D-root of its tree.

(viii) Invert(j). This operation takes the D-tree rooted at D-root(j), and roots it at j instead. Consequently the parent-child relationship on the path from D-root(j) to j is inverted.

The dynamic trees data structure stores the residual capacities of arcs in the D-trees implicitly. However, if nodes i and j are in different D-trees, then $r_{\mathbf{x}}(i, j)$ is explicitly maintained. The following theorem was proved by Sleator and Tarjan [17].

Theorem 5.1 *For D-trees of size $O(q)$, the running time of each D-tree operation is $O(\log q)$ in an amortized sense. That is, over a sequence of $K > n$ consecutive operations on the D-trees, the running time is $O(K \log q)$.*

During the DFS process of scanning for merger arcs, there are no dynamic tree operations involved. Thus each arc $(p(j), j)$ traversed during the DFS search operation will be cut by calling for Cut(j) prior to pushing flow on it. After node j has been fully scanned, it is assigned the flag value of 1; in addition, there will be a Link$(j, p(j))$ operation if the sum of the sizes of the two D-trees containing j and $p(j)$ is less than or equal to q. During the backtrack operation from j to $p(j)$, the arc $(j, p(j))$ is cut from the D-tree, with operation Cut(j). Node j then has a label of $d' + 1$ whereas $d(p(j)) = d'$. If node j then has a child in T and if its left-most child i is in a different D-tree from j, then Link(i, j) is run if the sum of their respective sizes is less than or equal to q. These dynamic tree operations during the DFS process add $O(\log q)$ steps each. Since each arc is traversed down at most once in DFS and at most once in backtrack per label d' (see Claim 4.1), the total complexity added is $O(n \log q)$ per arc, or $O(mn \log q)$ for all arcs.

The operations that directly involve the dynamic trees are, first in PushDown(d'): the invert operation once a merger has been identified; the skip subroutine that skips all d'-visited nodes along the downwards DFS search path and pushes the flow down from j to the next unvisited node, unvisited$_{d'}(j)$. In FinalPush(d') the operation is that of pushing up flows from nonroot nodes with excess to their respective roots. All these operations are analogous in the sense that they use, along the path, only the D-roots of the respective D-trees encountered, except possibly for the first one, which may not be a D-root.

In order to implement the FinalPush(d') operations in reverse topological order, we demonstrate how to construct a so-called *auxiliary graph*, which represents the tree T but which involves as nodes only D-roots.

In the remainder of this section, we explain how to modify PushDown(d') and FinalPush(d') so that their running time is $O(nm \log q)$ over all iterations.

We first show how to modify PushDown(d') using the dynamic trees data structure. We then provide full details on how to modify FinalPush(d').

5.1 Modifying PushDown(d')

The major operations of PushDown(d') involve pushing flow from a parent to child, or returning excess from a child to a parent. These operations are implemented simply on an arc $(p(j), j)$ by cutting it first and then performing the push, and then, if no split, merging again the two D-trees subject to size restriction. In the backtrack to parent, the child node j is relabeled and

the arc $(j, p(j))$ is cut, with Cut(j).

When we reach a d'-visited node j in the DFS process, we use the skip operation to proceed to unvisited$_{d'}(j)$, a descendant of j that has not been fully scanned yet, and push the flow on the path from j to unvisited$_{d'}(j)$. In the next lemma we prove that such a descendant always exists. In fact, *every* leaf node is such a descendant.

Lemma 5.1 *All leaf nodes in $T(d')$ are d'-unvisited.*

Proof: Consider the DFS algorithm without skips. Once the DFS algorithm reaches a d'-visited node j (no merger arc from j), it scans the children of j. It then either continues the DFS process to one of the children of label d', or else relabels j, removes it from $T(d')$ and $V(d')$, and backs up to a parent. Therefore, a node that is d'-visited cannot be a leaf node. ■

We add the operation combine(i, j) for arc $(i, j) \in T$ when nodes i, j do not belong to the same D-tree (tested by comparing their respective D-roots). This operation checks if size(i)+size$(j) \leq q$ and if so, it calls Link(i, j).

We later show that each D-tree operation, applied to a path of length P, requires no more than $O((P/q) \log q)$ steps, plus an additional complexity of $O(mn \log q)$ throughout the algorithm, for all operations. We first present the DT version of PushDown. Recall that List:= $(r_1, \ldots, r_\ell)$ initially.

```
procedure DT-PushDown(d')
    begin
        if (List≠ ∅) then
            j :=TopL(List); {excess e(j) > 0}
            if (flag_{d'}(j) = 1) then
                D-Push-down(j,unvisited_{d'}(j), e(j));
                DT-PushDown(d');
            else
                if (there is a merger arc (j, j') originating at j) then
                    isSaturating := Push(x, j, j');
                    if (isSaturating = saturating and e(j) > 0) then
                    {do nothing, node j is still on List and node j' is a nonroot node with excess}
                    else
```

PopL(List); {remove node j from List}
{*invert*} invert path from root(j) to j;
{*merge*} $p(j) := j'$; {add (j, j') to T} combine(j, j');
end if
DT-PushDown(d');
else {j is d'-visited and excess is pushed down to a descendant of j}
$\text{flag}_{d'}(j) := 1$;
for (each child of j: $i \in ch_{d'}(j)$) **do**
if (j and i are in the same D-tree) **then**
Cut(i);
end if
$isSaturating :=$ Push$(\mathbf{x}, j, i)$;
if $(e(j) = 0)$ **then**
PopL(List); {remove node j from List}
end if
PushL(i); {add node i on List}
if $(isSaturating = saturating)$ **then**
$p(i) := \emptyset$; {set node i as a root}
else
combine(j, i);
end if
DT-PushDown(d');
end for
{backtracking}:
{relabel}:
$d(j) := d(j) + 1$;
if $(j =$ TopL(List)$)$ **then**
PopL(List); {remove node j from List}
end if
{return excess to parent}:
if $(p(j) \neq \emptyset)$ **then**
$isSaturating :=$ Push$(\mathbf{x}, j, p(j))$;

```
                        PushL(p(j)); {add node p(j) on List}
                        if (isSaturating = saturating and e(j) > 0) then
                            p(j) := ∅; {set node j as a root}
                        end if
                    end if
                end if
            end if
        end if
    end
```

An important pair of routines push flow from a node j with a positive excess $e(j)$, up to a node (usually an ancestor root node) w along a path in T (implemented as D-Push-up$(j, w, e(j))$ in the following), or down to a descendant node w along a path in T (implemented as D-Push-down$(j, w, e(j))$ in the following).

```
procedure D-Push-down(j, w, e(j))
    begin
        if (j ≠ w and j has a child in T and e(j) > 0) then
            k :=Find-first-down(j, w, e(j));
            if (k = w) then
                AddValue(j, w, e(j));
                PopL(List); {remove node j from List}
                PushL(w); {add node w on List}
            else {r_x(k, c(k)) ≤ e(j)}
                PopL(List); {remove node j from List}
                AddValue(j, k, e(j));
                Push(x, k, c(k));
                p(c(k)) := ∅; {set node c(k) as a root}
                {k has excess e(k) = e(j) − r_x(k, c(k))}
                if (e(k) > 0) then
                    PushL(k); {add node k on List}
                end if
```

```
            Cut(c(k));
            e(c(k)) := r_x(k, c(k));
            D-Push-down(c(k), w, e(c(k)));
        end if
    end if
end
```

The next procedure is used only in FinalPush(d') (next section), but its structure is similar to that of D-Push-down($j, w, e(j)$) and is therefore placed here.

```
procedure D-Push-up(j, w, e(j))
    begin
        if (j ≠ w and j is not a root of T and e(j) > 0) then
            k :=Find-first-up(j, w, e(j));
            if (k = w) then
                AddValue(j, w, e(j));
            else {r_x(k, p(k)) ≤ e(j)}
                AddValue(j, k, e(k));
                Push(x, k, p(k));
                p(k) := ∅; {set node k as a root}
                {k has excess e(k) = e(j) − r_x(k, p(k))}
                Cut(k);
                e(p(k)) := r_x(k, p(k));
                D-Push-up(p(k), w, e(p(k)));
            end if
        end if
    end
```

Throughout the algorithm, all calls to D-Push-down or D-Push-up can create up to $O(mn)$ new D-trees that are potentially small, that is, of size $< q/2$. All such created D-trees must have, by construction, a D-root that belongs to R, the set of roots in T.

The procedures Find-first-up($j, w, e(j)$), Find-first-down($j, w, e(j)$), AddValue($j, w, e(j)$),

and invert(j) are all built on the respective basic operations on D-trees. We explain next how to construct these operations along paths that include multiple D-trees. This is shown for AddValue($j, w, e(j)$) (in the up direction) where for the others the implementation is analogous. In the procedure AddValue, we use the name AddValue-D for the AddValue operation used within a single D-tree. Every time an out of D-tree arc is processed, from a D-root to its parent, the combine operation is invoked as well.

procedure AddValue($j, w, e(j)$)
  begin
    while (j and w are not in the same D-tree) **do**
      AddValue-D(j,D-root(j), $e(j)$);
      $i := p$(D-root(j));
      Push(**x**, D-root(j), i);
      combine(D-root(j), i);
      $j := i$;
    end while
    AddValue-D($j, w, e(j)$);
  end

The significance of using combine is that it results in a processed part of the tree T that has on all paths large D-trees at least on every alternate level. The number of additional small D-trees that are created at the root level is $O(mn)$ throughout the algorithm. The additional D-trees are created only when there is a split and a node becomes a D-root. Since throughout the algorithm there are at most $O(mn)$ splits, the number of small D-trees is at most $O(mn)$.

An analogous argument applies for the invert path operation, the AddValue operation and the find-first-down/up operations. All these take a path of length P that contains at most $O(P/q)$ D-trees. Each such operation thus runs in at most $O((P/q)\log q)$ time per path. This argument is made formal in the next lemma. For unvisited$_{d'}(j)$ and therefore the skip operation, it is shown in the appendix that the same complexity applies also.

Lemma 5.2 *The complexity of invert, AddValue and the other operations is* $O(mn \log q)$ *throughout the algorithm.*

Proof: We first consider the application of these operations to a single path of length P.

Because combine() is integrated into all these operations, it follows that along a path of length P there are at most $O(1 + P/q)$ D-trees, and this results in $O((1 + P/q) \log q)$ time.

Each arc (j, i) has flow sent on it $O(1)$ times when it has a label of d', thus $O(n)$ times in total. Thus the sum of the lengths of all paths on which the operations apply is $O(nm)$, and the number of paths is $O(nm)$. Summing $O((1+P/q) \log q)$ over the lengths of all paths results in a total running time of $O(nm \log q)$. ■

Theorem 5.1 *The complexity of* DT-PushDown *throughout the algorithm is* $O(mn \log(n^2/m))$.

Proof: For a label d', each arc is processed at most once in the pushing down of flow and at most once in the backtracking. Since there are up to n labels the processing of arcs requires $O(mn)$ throughout. Since we use the Cut and combine operations, these contribute $O(mn \log q)$ complexity.

In each phase, each arc can participate at most once in the skip or invert operations, and thus at most once in the AddValue and Find-first-down operations. That is because the skip operation is part of the DFS process during a phase, and once the search has gone down the tree with the skip operation, the same arcs cannot be skipped again but only backtracked on, as already accounted for above. For the invert operation, all arcs on the path that is inverted are then removed from $T(d')$ and are no longer processed in the current phase. Thus the total lengths of the paths processed by skip and invert, per phase, is at most $O(n)$.

The skip operation and the other operations can also be invoked at most $O(n)$ times per phase, and possibly on very short paths. If the paths are of length $P > q$, then the complexity per skip is $O((P/q) \log q)$. Otherwise the complexity per operation is $O(\log q)$.

Since there are $O(n^2)$ phases, and the skip and invert operations can process multiple paths of total length at most n arcs per phase, the total work of these operations is $O((n^3/q) \log q)$. Having chosen $q = n^2/m$, this complexity is $O(mn \log(n^2/m))$ throughout the entire algorithm.

Adding the complexity of all these operations and that of the pushing down and backtracking on an arc the complexity is $O(mn \log(n^2/m))$. Finding all mergers throughout the algorithm adds only $O(mn)$ steps.

■

5.2 Modifying FinalPush(d')

The key in constructing the DT analogue of FinalPush(d') is to process the topological order final push on D-roots only. In a *preprocessing step* of DT-FinalPush, all nonroot nodes with excess (and of label $d'-1$) send their excess to their respective D-roots. For each such node j', this is done using D-Push-up(j',D-root(j'), $e(j')$). Since nonroot nodes with excess are created only as result of mergers, and there are $O(mn)$ mergers throughout the algorithm, this process takes $O(mn \log q)$ throughout the algorithm.

We can thus assume that all nonroot nodes with excess are D-roots. We next construct the auxiliary graph induced on D-root nodes only that contain the D-root nodes with excess, and all their D-root ancestors. This is the graph on which the reverse topological push will be processed.

Figure 4(a) is one branch of T after the preprocessing step. The thick black arcs are in the dynamic tree. The gray arcs are arcs of T that are not in the dynamic tree. Nodes with no black arc leaving are D-roots. The gray nodes are active. Note that all active nodes are D-roots. The only D-roots that are not active in the branch are nodes 6 and 9. The only nodes that are not D-roots are nodes 2, 3 and 7.

We let the set of active D-root nodes created at the preprocessing step be called ActiveSet. We will be interested in those active D-roots that do not belong to R. We denote $S =$ActiveSet$\backslash R$. The auxiliary graph is the graph of the nodes of S and their D-root ancestors, on which the reverse topological order is defined.

procedure AuxiliaryGraph(S)

begin

$V' := S$, $A := \emptyset$;

for (each node i of S) **do**

$v := i$;

while (v is not a root node of T) **do**

$w :=$D-root($p(v)$);

$V' := V' \bigcup \{w\}$;

$A' := A' \bigcup \{(v, w)\}$;

$v := w$;

end while

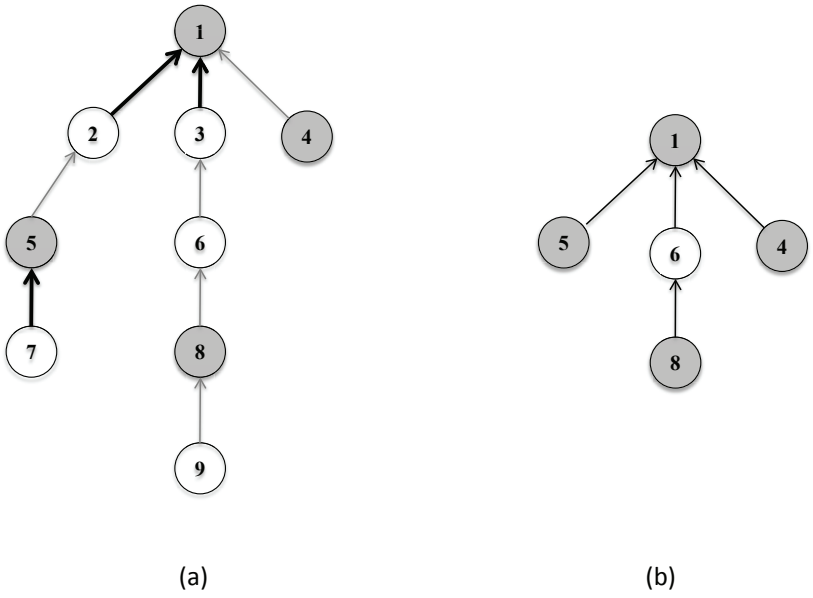

Figure 4: (a) A branch of T after preprocessing. (b) The auxiliary graph for Figure 4(a).

end for

end

The graph $G' = (V', A')$ is the auxiliary graph. The auxiliary graph output for the graph given in Figure 4(a) is depicted in Figure 4(b). In this particular graph only D-roots are present.

5.2.1 DT-FinalPush(d') and its complexity

We are now ready to present the procedure FinalPush with dynamic trees. Here ActiveSet is the set of nonroot nodes that are active at the beginning of DT-FinalPush and DRootSet is the set of all D-roots.

procedure DT-FinalPush(d')

begin

{Preprocessing}:

for (each node $j' \in$ ActiveSet\DRootSet) **do**

D-Push-up(j',D-root(j'), $e(j')$);

{update ActiveSet} ActiveSet:=ActiveSet\\{j'}$\bigcup${D-root(j')};

end for

$S :=$ ActiveSet$\backslash R$;
let $G' = (V', A')$ be created by AuxiliaryGraph(S);
order the nodes of V' in reverse topological order in G';
for (node $j' \in V' \setminus R$ in the reverse topological order) **do**
Push$(\mathbf{x}, j', p(j'))$;
D-Push-up$(p(j')$,D-root$(p(j')), e(p(j')))$;
combine$(j', p(j'))$;
end for
end

Theorem 5.2 *The complexity of DT-FinalPush throughout the algorithm is* $O(nm \log q)$.

Proof: As discussed above, the preprocessing step takes $O(nm \log q)$ time. The complexity of the remainder of the algorithm is determined by the number of nodes in V'. This is because the creation of the reverse topological order takes $O(|V'|)$ time, and the Push and D-Push-up operations take $O(\log q)$ per node of V'.

Because the operation combine is invoked throughout the DT-FinalPush procedure, for every two consecutive D-trees on a path in T from a node to its root, at least one of the D-trees is large. Equivalently, in the auxiliary graph G', for every two consecutive nodes on a path from a node to a root node, at least one of these two nodes represents a large D-tree, which is of size at least $q/2$. Hence, the number of large D-trees in G' is at most $2n/q$ and the size of $|V'|$ is at most the number of leaves in G' plus $4n/q$.

Every leaf node of G' is created from a nonroot active node, which is created, in turn, by a merger operation. Therefore, throughout the algorithm, the total number of leaf nodes in all auxiliary graphs created is at most $O(mn)$.

Hence, throughout the algorithm and all calls to DT-FinalPush, the total number of nodes in all auxiliary graphs is at most $n^2 \cdot 4n/q + mn$. Since we choose $q = n^2/m$, this number is $O(mn)$. The statement of the theorem thus follows.

■

Now we can describe the complete DT-algorithm.

DT-algorithm

```
begin
    Initialize();
    for (each label d ∈ {1, ..., n}) do
        for (each node j ∈ V) do
            flag_d(j) := 0; DT-flag_d(j) := 0; unvisited_d(j) := j;
        end for
    end for
    while (there are any active nodes) do
        let d' be the highest label of an active node;
        {let r_1, ..., r_ℓ be the initial active d'-labeled root nodes.}
        initialize List as List:= (r_1, ..., r_ℓ);
        DT-PushDown(d');
        DT-FinalPush(d');
    end while
end
```

We thus conclude that:

Theorem 5.3 *The pseudoflow algorithm using dynamic trees, the DT-algorithm, solves the maximum preflow problem in $O(nm\log(n^2/m))$ time.*

6 Conclusions

We provide here a unique and new perspective on the pseudoflow algorithm. Two variants of the algorithm, the DFS variant and the dynamic trees variant, devised here, have complexities of $O(n^3)$ and $O(nm\log(n^2/m))$ respectively, improving on previous complexity bounds for the algorithm. We believe that the algorithmic concepts introduced here will lead to improvements in running time of maximum flow algorithms not only in theory, but in practice as well.

Acknowledgement

The authors wish to express their thanks to Cheng Lu for useful discussions, and his help in adjusting the pseudocodes.

References

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

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network flows: Theory, algorithms, and applications, Prentice-Hall, New Jersey, 1993.

[3] R.J. Anderson and J.C. Setubal, Goldberg's algorithm for maximum flow in perspective: a computational study, Network flows and matching: First DIMACS implementation challenge, DIMACS series in Discrete Mathematics and Theoretical Computer Science, Vol. 12, 1991, pp. 123–133.

[4] Y. Boykov and V. Kolmogorov, An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision, IEEE Transactions on Pattern Analysis and Machine Intelligence 26 (2004), 1124–1137.

[5] B.G. Chandran and D.S. Hochbaum, A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem, Operations Research 57 (2009), 358–376.

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

[7] E.A. Dinic, Algorithm for solution of a problem of maximal flow in a network with power estimation, Soviet Math. Dokl. 11 (1970), 1277–1280.

[8] B. Fishbain, D.S. Hochbaum, and S. Mueller, Competitive analysis of minimum-cut maximum-flow algorithms in vision problems, UC Berkeley manuscript, May 2010.

[9] L.R. Ford and D.R. Fulkerson, Maximal flow through a network, Canadian J. of Math. 8 (1956), 339–404.

[10] A.V. Goldberg and B.V. Cherkassky, On implementing the push-relabel method for the maximum flow problem, Algorithmica 19 (1997), 390–410.

[11] A.V. Goldberg and R.E. Tarjan, A new approach to the maximum flow problem, J. of the ACM 35 (1988), 921–940.

[12] A.V. Goldberg and S. Rao, Beyond the flow decomposition barrier, J. of the ACM 45 (1998), 783–797.

[13] D.S. Hochbaum, The Pseudoflow algorithm: A new algorithm for the maximum flow problem, Operations Research 58 (2008), 992–1009.

[14] A.V. Karzanov, Determining the maximal flow in a network with a method of preflows, Soviet Math. Dokl. 15 (1974), 434–437.

[15] V. King, S. Rao, and R.E. Tarjan, A faster deterministic maximum flow algorithm, J. of Algorithms 17 (1994), 447–474.

[16] V.M. Malhotra, M.P. Kumar, and S.N. Maheshwari, An $O(|V|^3)$ algorithm for finding maximum flows in networks, Information Proc. Letters 7 (1978), 277–278.

[17] D.D. Sleator and R.E. Tarjan, A data structure for dynamic trees, J. of Computer and System Sciences 26 (1983), 362–391.

Appendix: Dynamic trees maintenance of $\text{flag}_{d'}(j)$, DT-$\text{flag}_{d'}(j)$ and $\text{unvisited}_{d'}(j)$ labels

6.1 Overview

The purpose of the $\text{unvisited}_{d'}(j)$ label is to identify a left-most descendant of node j which has not been d'-visited, in the same D-tree, and that is closest to j, or indicate that such a node does not exist.

The $\text{unvisited}_{d'}(j)$ label is a pointer to a left-most descendant i of node j with flag-$\text{DT}_{d'}(j) = 0$ and $\text{flag}_{d'}(i) = 0$ that is "closest" to j in the same D-tree among all descendant nodes with $\text{flag}_{d'}$ value equal to 0. More precisely, if $\text{flag}_{d'}(j) = 0$, then this operation returns j; else, it returns a left-most descendant i of j, in the same D-tree, such that all nodes on the path from $p(i)$ to j have $\text{flag}_{d'}$ value 1 and $\text{flag}_{d'}(i) = 0$. We will show that for each label d', the overhead for maintaining the labels $\text{unvisited}_{d'}()$, $\text{flag}_{d'}()$ and DT-$\text{flag}_{d'}()$ is at most $O(m)$. We also show that the labels are correctly updated and maintained for each of the following operations occurring during the DFS process:

1. link
2. invert

3. update associated with a node having been d'-visited

4. cut.

Furthermore, the complexity of *invert* and *update* operations is dominated by the complexity of the DT algorithm; and the *link* and *cut* operations, that are performed on D-trees, have complexity $O(\log q)$ for D-trees on q nodes. Therefore throughout the algorithm their complexity does not increase the overall complexity of the DT algorithm.

Notations: DT will be used here as an abbreviation of D-tree. The value $\text{flag}_{d'}(j) = 1$ indicates that node j has been d'-visited; DT-$\text{flag}_{d'}(j) = 1$ if node j has no descendant in the same DT that is d'-unvisited; $\text{unvisited}_{d'}(j)$ is the closest d'-unvisited left-most descendant of j in the same DT. The value of this label is equal to "nil" if DT-$\text{flag}_{d'}(j)$=1.

The invariant that we maintain during the DFS process is that in each DT, the children of each node get the label DT-$\text{flag}_{d'}(i) = 1$ from left to right. That is, the list of the children of node j, in left-to-right order, have the DT-$\text{flag}_{d'}$ labels' sequence $(1, \ldots, 1, 0, \ldots, 0)$, where the left-most child with the label DT-$\text{flag}_{d'}(i) = 0$ is termed the *current-child*. Notice that some children of nodes in a DT may belong to other DTs.

The current child of node j in the same DT is denoted by $cc_{d'}(j)$. The child that is next to (to the right of) $cc_{d'}(j)$ and in the same DT, is denoted by $cc^{+}_{d'}(j)$. If $cc^{+}_{d'}(j) = nil$, then $cc_{d'}(j)$ is the right-most, and last, child of j in the same DT.

The main DT operation used here is analogous to, and somewhat simpler than, AddValue(i, w, val). It is ReplaceUnvisited$(i, index, d')$ (the node w here is always the D-root(i)); replaces the $\text{unvisited}_{d'}$ label by $index$ in all $\text{unvisited}_{d'}$ labels for all nodes v on the path in T from node i to D-root(i).

A second, and similar, DT operation is ReplaceDT-flag(i, val, d'). It replaces the DT-$\text{flag}_{d'}$ labels of all nodes v on the path in T from node i to D-root(i) by val.

6.2 Link

When two DTs are merged along arc (j, i), it is between a node j in one DT and the D-root i of the other, so that i becomes a child of j. This is illustrated in Figure 5 where the black nodes are those that have been d'-visited, with DT-$\text{flag}_{d'}$ value 1, for which all descendants in the same DT tree have $\text{flag}_{d'}$ value 1 (have been d'-visited). Thus, for black nodes, the $\text{unvisited}_{d'}$ label value is "nil". The other nodes still have d'-unvisited descendants. With the link operation, node i becomes a child of j positioned to the left of the *current-child* of j, $cc_{d'}(j)$, (the left-most

child of j with DT-flag$_{d'}$ value 0).

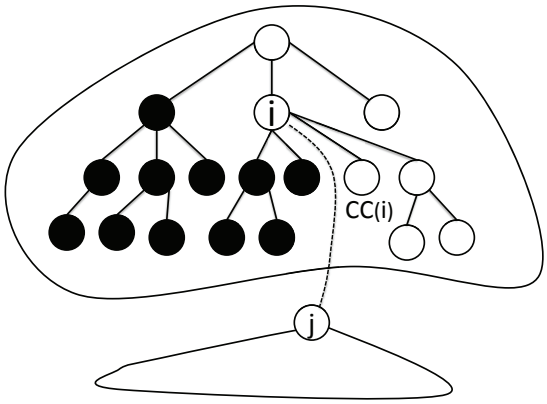

Figure 5: A link operation.

The update of the visited related labels depends on the values of the flag$_{d'}$ and the DT-flag$_{d'}$ labels of j and i as follows:

Case (a): DT-flag$_{d'}(i) = 1$. Here no update is needed and $cc_{d'}(j)$ remains the current child of j.

Case (b): DT-flag$_{d'}(i) = 0$. Then $cc_{d'}(j) := i$.

(b1) If DT-flag$_{d'}(j) = 0$ and flag$_{d'}(j) = 0$ (so unvisited$_{d'}(j) = j$), then there is no further update.

(b2) If flag$_{d'}(j) = 1$ (in which case unvisited$_{d'}(j) \neq j$), then ReplaceUnvisited(j,unvisited$_{d'}(i)$).

(b3) If DT-flag$_{d'}(j) = 1$ (and then flag$_{d'}(j) = 1$), then ReplaceDT-flag$(j, 0, d')$.

Each of the DT operations in (b2) and (b3) takes $O(\log q)$ steps for a dynamic tree on q nodes.

6.3 Invert

The invert operation occurs during the algorithm only along a path between a node j and D-root(j). When invert occurs, then all nodes on the path [D-root(j), ..., $p(j)$] must have been d'-visited and have flag$_{d'}$ values equal to 1. After the invert, node $p(j)$ becomes a child of j. Node $p(j)$ is inserted to the left of the current child of j. Also, after the invert operation, D-root(D-root(j)) = j. Note that prior to the invert, the current child of j may not belong to the same DT as j.

Let $w = cc_{d'}$(D-root(j)) prior to the invert operation and $w' = cc^{+}_{d'}$(D-root(j)) the next child

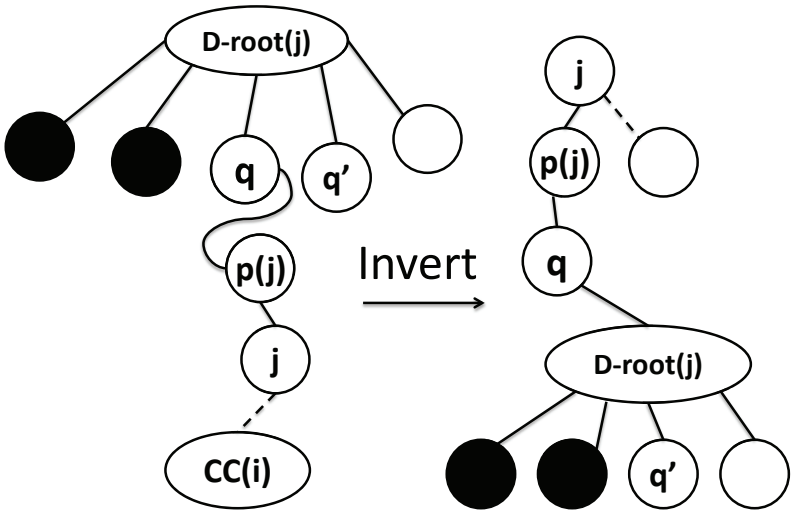

Figure 6: The invert operation.

of D-root(j) (the one to the right of w). After the invert, w becomes the parent of D-root(j) and w' becomes the current child of D-root(j), as illustrated in Figure 6. Node $p(j)$ is inserted as left-most d'-unvisited child of j after the invert, so $cc_{d'}(j) := p(j)$.

The formal update procedure occurring after a inversion is:

procedure update-invert

begin

if $(w' \neq nil)$ **then** $\{\text{DT-flag}_{d'}(w') = 0\}$

ReplaceUnvisited(D-root(j), w', d');

{for all i along the path [D-root(j), ..., j], $\text{unvisited}_{d'}(i) := w'$}

else

backtrack([D-root(j), ..., j]);

end if

end

The purpose of backtrack([D-root(j), ..., j]) is to shift the current child pointer of each node on the inverted path [D-root(j), ..., j] to the next child on its right.

procedure backtrack([u, ...,D-root(u)])

begin

$v := u$
while ($v \neq \emptyset$ and $cc^+_{d'}(v) = nil$) **do**
DT-flag$_{d'}(v) := 1$;
$v := p(v)$;
end while
if ($v \neq \emptyset$) **then**
$v' := cc^+_{d'}(v)$;
ReplaceUnvisited(v, v', d'); {for all z on $[v, \ldots,$D-root$(u)]$, unvisited$_{d'}(z) := v'$}
end if
end

The complexity of the backtrack subroutine appears to be linear in the length of the path for each invert and thus potentially of high complexity throughout the algorithm (for the $O(mn)$ inverts that occur throughout the algorithm, it would be $O(mn^2)$ complexity). However, this operation is charged to the backtracking of the DFS process and for every label d', each arc can be visited at most once in the backtrack process. Thus the total charge for the backtracking is $O(m)$ per label value, and $O(mn)$ for the entire algorithm.

6.4 Update associated with a node having been d'-visited

A node has been d'-visited when all its neighbors adjacent to admissible arcs haven been scanned. In this case, flag$_{d'}(j) := 1$.

procedure update-visited(j)
begin
if ($cc_{d'}(j) = nil$) **then**
DT-flag$_{d'}(j) := 1$;
unvisited$_{d'}(j) := nil$;
backtrack$([j, \ldots,$D-root$(j)])$;
else
ReplaceUnvisited$(j, cc_{d'}(j), d')$;
end if
end

6.5 Cut

A cut is the partition of a DT obtained by removing arc (i, j), where $j = p(i)$, from the DT. The following procedure shows how to update the labels of j and its ancestors when arc (i, j) is removed and i is no longer a child of j in the same DT.

procedure update-cut(i)

begin

if $(i \neq cc_{d'}(j))$ **then**

remove i from the list of children of j in the DT;

else $\{i = cc_{d'}(j)\}$

$w' := cc^{+}_{d'}(j)$;

if $(w' = nil)$ **then**

DT-flag$_{d'}(j) := 1$;

unvisited$_{d'}(j) := nil$;

backtrack([$j, \ldots$,D-root(j)]);

else

$cc_{d'}(j) := w'$;

ReplaceUnvisited(j, w', d');

remove i from the list of children of j in the DT;

end if

end if

end

As before, if the D-tree containing node i is of size q, then the complexity of update-cut is $O(\log q)$.

This completes the proof that the labels can be maintained in time dominated by the complexity of the main algorithm, $O(mn \log \frac{n^2}{m})$.