February 1989 Report No. STAN-G-89-1248

Efficiency of the Network Simplex Algorithm for the Maximum Flow Problem

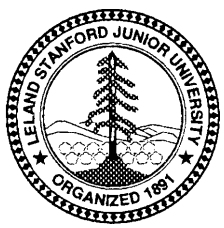
by

Andrew V. Goldberg, Michael D. Grigoriadis, and Robert E. Tarjan

Department of Computer Science

Stanford University

Stanford, California 94305



February 1989

Efficiency of the Network Simplex Algorithm for the Maximum Flow Problem

Andrew V. Goldberg[1]
Michael D. Grigoriadis[2]
Robert E. Tarjan[3]

[1] Department of Computer Science, Stanford University, Stanford, CA 94305. Research partially supported by a Presidential Young Investigator Award from the National Science Foundation, Grant No. CCR-8351097, an IBM Faculty Development Award, and AT&T Bell Laboratories.

Department of Computer Science, Rutgers University, New Brunswick, NJ 08903. Research partially supported by the Office of Naval Research, Contract No. N00014-87-K-0467.

[3] Department of Computer Science, Princeton University, Princeton, NJ 08544 and AT&T Bell Laboratories, Murray Hill, NJ 07974. Research partially supported by the National Science Foundation, Grant No. DCR-S605961, and the Office of Naval Research, Contract No. N00014-87-K-0467.

Efficiency of the Network Simplex Algorithm for the Maximum Flow Problem

Andrew V. Goldberg'
Michael D. Grigoriadis[2]
Robert E. Tarjan[3]

February, 1989

Abstract. Goldfarb and Hao have proposed a. network simplex algorithm that will solve a maximum flow problem on an n-vertex, m-arc network in at most *nm* pivots and 0(n^2m) time. In this paper we describe how to implement their algorithm to run in *O(nm* log n) time by using an extension of the dynamic tree data structure of Sleator and Tarjan. This bound is less than a logarithmic factor larger than that of any other known algorithm for the problem.

Key Words and *Phrases*: algorithms, complexity, data structures, dynamic trees, graphs, linear programming, maximum flow, network flow, network optimization.

1. The Maximum Flow Problem

Let $G = (V, E)$ be an undirected graph with vertex set V of size n and edge set E of size *m*. We regard each edge {v, w} as consisting of two oppositely-directed arcs, (v, w) and (w, v). For any vertex v we denote by E(v) the set of vertices w such that {v, w} is an edge. We assume that G is connected and that $n \geq 2$. Let each arc (v, w) of G have a nonnegative real-valued *capacity* $u(v, w)$. Finally, let s and t be two distinguished vertices of G; s is the *source* and t is the *sink*. A (feasible) *flow* on G is a real-valued function f on the arcs satisfying the following constraints:

[1] Department of Computer Science, Stanford University, Stanford, CA 94305. Research partially supported by a Presidential Young Investigator Award from the National Science Foundation, Grant No. CCR-8858097, an IBM Faculty Development Award, and AT&T Bell Laboratories.

[2] Department of Computer Science, Rutgers University, New Brunswick, NJ 08903. Research partially supported by the Office of Naval Research, Contract No. N00014-87-K-0467.

[3] Department of Computer Science, Princeton University, Princeton, N.J OS544 and AT&T Bell Laboratories, Murray Hill, N.J 07974. Research partially supported by the National Science Foundation, Grant No. DCR-8605961, and the Office of Naval Research, Contract No. N00014-87-K-0467.

For every arc (v, w), $f(v, w) = -f(w, v)$ (antisymmetry constraints) (1.1)

For every arc (v, w), $f(v, w) \leq u(v, w)$ (capacity constraints) (1.2)

For every vertex v $\notin \{s, t\}$, $\sum_{w \in E(v)} f(v, w) = 0$ (conservation constraints) (1.3)

The *value* of a flow f is value (f) = $\sum_{v \in E(s)} f(s, v)$. The *maximum flow problem* is that of finding a flow of maximum value.

To date, the asymptotically fastest known algorithms are those of Goldberg and Tarjan [S] and of Ahuja, Orlin, and Tarjan [1]. The former runs in $O(nm \log(n^2/m))$ time. The latter requires integer capacities; it runs in $O(nm \log(n(\log U)^{1/2}/m + 2))$ if no capacity exceeds U. Both of these algorithms are based on the $O(n^3)$-time algorithm of Goldberg [5]. Extensive discussions of the problem, its applications, and classical algorithms for it can be found in [5], [13], [14], [17].

The above statement of the maximum flow problem simplifies notation by avoiding explicit mention of "forward" and "backward" residual arcs. It is completely equivalent to the usual formulation on directed graphs. The case where two oppositely-directed arcs (v, w) and (w, v) have nonnegative capacities $u(v, w)$ and u (w , v) and zero lower bounds can be represented by an undirected edge $\{v, w\}$ haying lower bound $-u(w, v)$ and capacity $u(v$, w). We also assume for simplicity that no pair of vertices in G is connected by more than one edge, but allowing G to be a multigraph does not in any way affect our results.

2. The Network Simplex Algorithms

The network simplex algorithm is a specialization of the revised simplex method that uses an appropriate data structure and a pivot selection rule for its implementation. It is based on an early observation by Fulkerson and Dantzig [6] and Dantzig [4] that any basis matrix of a vertex-edge incidence matrix of G corresponds to a rooted spanning tree and can be permuted to an upper triangular matrix with a ± 1 diagonal. (For a description of the method see e.g. the books of Chvatal [2], and Kennington and Helgason [12]; for an implementation see Grigoriadis [1 1].)

We state the network simplex algorithm for the maximum flow problem in a form suitable for our implementation; we omit, for example, the return arc (t, s) that is added in the standard treatment. Given a flow f, an arc (v, w) has *residual capacity* $u_f(v, w)$ =

$u(v, w) - f(v, w)$. Arc (u, w) is *saturated* if $u_f(v, w) = 0$ and *residual* if $u_f(v, w) > 0$. An edge $\{v, w\}$ is *saturated* if either $(v$, co) or (w, v) is saturated, and residual otherwise. A *basic flow* is a flow f such that the set of residual edges forms a forest (a set of trees) with s and t in different trees. Given a basic flow f, a *basis* is a pair of trees S, Z that are subgraphs of G, such that $s \in$ S, $t \in Z$, and every vertex and every residual edge is in either S or Z. Given a basic flow f and a. basis S, Z, an edge (or arc) is a *tree edge* (or tree arc) if it is in S or in Z, and a *nontree edge* (or *nontree arc*) if not. A basic flow f is called *degenerate* if there is a saturated tree edge and *nondegenerate* otherwise.

The network simplex algorithm maintains a basic flow f and a corresponding basis S, Z. Starting from such a flow f and basis S, Z, the algorithm consists of repeating the following step until there is no residual arc $(v$, rv) with v $\in$ S, $t \in Z$:

Pivot. Select a residual arc (v, w) with $v \in$ S, $w \in Z$. Add {v, w} to S **U** 2, forming a single spanning tree *T.* This tree contains a unique simple path p of tree arcs from s to t. Let δ be the minimum capacity of an arc on p. Add δ to the flow of every arc on *p.* Delete from *T* some edge {x, y} such that (x, y) is a saturated arc of *p.* This produces two trees that form a basis for the new basic flow.

Arc (v, w) is called the *entering* arc of the pivot and (x, y) the leaving arc; the pivot is said to be on (v, w). It is possible for 6 to equal zero if the basic flow is degenerate; then the pivot is said to be *degenerate.* A degenerate pivot does not change the flow but does change the basis. A nondegenerate pivot changes the flow, increases the flow value, and may or may not change the basis.

If a basic flow and a corresponding basis are not available initially, they can be computed in $O(nm)$ time in several ways. One way is as follows. Let f = 0 and compute a. spanning tree of *T* of G. Then, select a nontree residual edge, identify the unique simple cycle it forms in *T,* and push flow around this cycle so that at least one of its edges is saturated. Repeat this step until there are no nontree residual edges. Finally, push enough flow from s to t along the unique (s, t) path in *T* so that at least one additional edge is saturated. Deleting from *T* this edge yields a basic flow and a basis S, Z. The running time for this computation can be reduced to to $O(m \log n)$ by using the dynamic tree data structure [15], [16], [17], but this does not improve the running time of the overall algorithm.

3. A Refinement of the Algorithm with a Polynomial Number of Pivots

The algorithm of the previous section need not terminate unless an anti-cycling rule, such as Cunningham's [3], is used for breaking ties in selecting the leaving arc. For integer data, such an implementation solves the maximum flow problem in at most n^2U pivots and in $O(n^2mU)$ time using a simple rooted tree data structure to represent the basis. Goldfarb and Grigoriadis [9] proposed a rule that pivot's on a residual arc (v, w) with v $\in$ S, $w \in Z$, for which the number of residual arcs in the paths from s to v in S and from w to t in Z is minimum over all nontree residual arcs from S to Z. This variant. works better in practice than others, but it does not improve the pseudopolynomial bound on the total number of pivots.

The key to making the network simplex algorithm run fast is to choose pivots more carefully. Goldfarb and Hao [10] proposed a pivot rule such that at most nm pivots occur. Explaining their rule requires a few extra definitions. We call an arc (v, w) *pseudoresidual* if it is residual or a tree arc[1]. For any vertex v, we define the label $d(v)$ of v to be the minimum number of pseudoresidual arcs on a path of pseudoresidual arcs from s to v, or infinity if there is no such path. Every vertex label remains finite, and indeed less than n, until after the last pivot. Goldfarb and Hao's pivot rule, which we call the *smallest label rule,* is:

Among all residual arcs (v, w) with $v \in$ S and w $\in Z$, pivot on one with $d(v)$ minimum.

Efficient implementation of this rule requires a reformulation of it, also proposed by Goldfarb and Hao: Repeat the following step until d(t) = ∞:

Choose a vertex $w \in Z$ with $d(w)$ minimum. Pivot on any residual arc (v, w) with v $\in$ S. (Such an arc will have *d(v) = d(w) − 1.)*

4. Efficient Implementation of the Smallest Label Rule

We shall describe a way to implement the smallest label rule so that the running time of the resulting network simplex algorithm is $O(nm \log n)$. This improves Goldfarb and Hao's bound of $0(n^2m)$, and is within less than a logarithmic factor of the bound of any other known algorithm.

[1] If the basic flow is nondegenerate, every pseudoresidual arc is also a residual arc.

Our implementation consists of two main parts. The first part, described in this section, is a way to maintain vertex labels in a total time of 0(nm). The second and more complicated part, explained in the next three sections, is a dynamic tree data structure used to choose pivots and to maintain the basis. The amortized time [2] per pivot with this data structure is $O(\log n)$, resulting in the claimed $O(nm \log n)$ overall time bound.

To maintain vertex labels, we use the method proposed by Goldberg and Tarjan for maintaining exact distance labels in their maximum flow algorithm (see [S], Section 7). For each vertex w, we maintain a pointer into a fixed list $A(w)$ of the arcs (v, w). This pointer indicates a pseudoresidual arc (v, w) with *d(v)* = *d(w)* − 1. That is, arc (v, w) is on some pseudoresidual path of fewest arcs from s to w. We call (v, w) the *current arc* of w. For each vertex w, we also maintain a list $L(w)$ of those vertices x such that the current arc of x is (w, x). Initializing this information at the beginning of the maximum flow computation can be done by a single breadth-first search from s, taking O(m) time.

Goldfarb and Hao proved that vertex labels can never decrease, only stay the same or increase, as the algorithm proceeds. Fur thermore, once a pseudoresidual arc (v, w) becomes a saturated nontree arc, it cannot become pseudoresidual again until at least one of *d(v)* and *d(w)* increases.

We need to update vertex labels after each pivot; the leaving arc (x, y) may no longer be pseudoresidual. If (x, y) is indeed no longer pseudoresidual, and if in addition (x, y) is the current arc of y, we delete y from $L(x)$ and initialize a set $R=\{y\}$ of vertices to be relabeled. Then we repeat the following step until *R* is empty:

Relabel. Select a vertex w ∈ *R* and delete it from *R*. Let (v, w) be the current arc of w. (Since w was on *R,* either (v, w) is no longer pseudoresidual or d(v) ≥ d(w).) Scan the arcs after (v, w) on *A(w)* until finding one, say (x, w), such that (x, w) is pseudoresidual and $d(x)$ = *d(w)* − *1,* or reaching the end of *A(* w). In the former case, make (x, w) the current arc of w and add w to *L(x);* the relabeling is complete. In the latter case, scan all of *A(* w) to find the first pseudoresidual arc (y, w) on A(w) with *d(y)* minimum. Make (y, w) the current arc of w, add w to L(y), set *d(w)* = *d(y)* + 1, add all vertices on *L(w)* to *R,* and set $L(w)$ = ∅, completing the relabeling. If there is no such arc (y, w), then *d(w)* = ∞;

[2] By *amortized time* we mean the time per operation averaged over a worst-case sequence of operations. See [18].

the maximum flow computation is complete.

It is straightforward to verify by induction the correctness of this method of maintaining vertex labels and current arcs. Each arc list *A(w)* for $w \neq$ s is scanned at most $2n - 2$ times, twice for each possible value of $d(w)$ (from 1 to $n - 1$). The total time needed to maintain vertex labels is thus *O(nm).*

5. The Use of Dynamic Trees

To choose pivots and maintain the basis, we use an extension of the dynamic tree data structure of Sleator and Tarjan [15], [16], [17]. This data structure will represent a collection of vertex disjoint rooted trees, each vertex of which has an integer label, and each edge $\{v, w\}$ of which has two associated real values, $g(v, w)$ and $g(w, v)$. We denote by parent(v) the parent of vertex v in its dynamic tree; if v is a tree root, $parent(v)$=null. We adopt the convention that every tree vertex is both an ancestor and a descendant of itself. The data structure supports the following ten operations on dynamic trees. Each operation takes $O(\log k)$ amortized time, where k is the total number of tree vertices.

m&e-tree(v): Make vertex v into a one-vertex dynamic tree. Vertex v must be in no other tree.

find-parent(v): Return the parent of vertex v, or null if v is a tree root.

find-value(v): Compute and return $g(v,$ parent(v)); if v is a tree root, return infinity.

find-min-vulue(v): Find and return an ancestor w of vertex v such that $g(w,$ *parent(w))* is minimum; if v is a tree root, return v.

find-min-label(v): Find and return a descendant w of v that has minimum label.

change-label(v, l): Set the label of v equal to *1.*

change-value(v, δ): Add real number δ to $g(w,$ *parent(w))* and subtract δ from $g(parent(w), w)$ for every nonroot ancestor w of v.

link(v, w, α, β): Combine the trees containing v and w by making w the parent of v. Define $g(v,$ w) = α and $g(w,$ v) = β. Before the *link* operation, vertices v and w must be in different trees, with v the root of its tree.

cut(u): Break the tree containing vertex v in two by deleting the edge joining v and its parent. Before the *cut* operation, vertex v must be a nonroot.

evert(v): Reroot the tree containing vertex v by making v the root.

To implement the network simplex algorithm, we maintain the basis S, Z as a pair of dynamic trees. Tree Z is permanently rooted at t; the root of S changes as the algorithm proceeds. Initialization of the two trees requires n *make-tree,* $n - 2$ *link,* and n *change-label* operations at the beginning of the algorithm. Each time a vertex label, as computed by the method in Section 4, changes, we perform the corresponding *change-label* operation.

To determine which pivot to do next during the computation, we perform fin&&-*label(t),* which returns a vertex in Z, say w, of smallest label. We pivot on the current asc (v, w) of w, as defined in Section 4. To actually carry out the pivot, we first perform *evert*(v), to root S at v. Then we perform *link*(v, *w, a,* β), where $\alpha = u_f(v, w)$ and $\beta = u_f(w,$ v). We compute the leaving arc (x, y) of the pivot by letting x be the vertex returned by *find-min-value*(s) and then letting y be the vertex returned by *find-parent(x).* The amount of flow to be moved from s to t is the amount, say δ, returned by *find-value(x).* To complete the pivot, we perform *change-value*(s, $-\delta$) and then *cut(x).* At the end of the maximum flow computation, we compute the flow on all the tree arcs by using $n - 2$ *find-value* operations.

With this implementation, each pivot takes O(1) tree operations. The amortized time per pivot is O(log *n),* so the overall running time of the network simplex algorithm is 0($nm \log n$), as desired.

6. Representation of Dynamic Trees by Phantom Trees

It remains for us to discuss how to implement dynamic trees so that the amortized time per tree operation is O(log n). Obtaining such an implementation requires extending the Sleator-Tarjan data structure. An extension designed to maintain edge values and to support a.11 the operations except *find-min-label* and *change-label* appears in [19] and can be used without modification here. The novel part of our implementation lies in the handling of vertex labels; whereas the original dynamic tree data structure was designed to compute combinations of values over tree paths, the operation *find-min-lube1* requires combining values over subt rees. We shall describe a data structure that supports the

operations *make-tree, find-parent, find-min-label, change-label, link, cut,* and *evert.* For the other operations, we can either use a separate data structure of the kind described in [19], or we can combine the two structures into one. This can be done by adding information representing edge values to the data structure described below. (See [19].)

To perform *find-min-label* operations efficiently, we need to impose a constant upper bound on the valence of each tree vertex. Thus we represent each dynamic tree D by a rooted *phantom tree P.* Tree *P* contains all the vertices of D and possibly some additional *dummy vertices.* Each vertes in a. phantom tree, henceforth called a *p-vertex,* has a *label* and a. *color.* In the simulation of a dynamic tree D by a phantom tree *P,* the colors are vertices in D. Every phantom tree has maximum valence three. The following operations are allowed on phantom trees: *make-tree, find-parent, find-min-label, change-label, link, cut,* and *evert,* with the added constraint on *link* operations that *link*(v, w) cannot be performed unless v and w both have valence at most two. (The third and fourth parameters of a link operation are unnecessary, since edges do not have values in phantom trees.) Phantom trees also support three additional operations:

find-children(v): Find and return the set of children of v.

find-top(v): Find the ancestor of v closest to the tree root that has the same color as v.

change-color(v, γ): Set the color of v equal to γ.

The precise correspondence between dynamic trees and phantom trees is as follows. In a phantom tree *P* corresponding to a dynamic tree D, there is a path *p(v)* of vertices colored v corresponding to each vertex v of D.. One of the vertices of *p(v)* is identified with v and has the same label as v; the remaining vertices of *p(v)* are dummy vertices, each of which has label oo and valence exactly three. Each edge {v, w} of D corresponds to an edge {v', w'} of *P* with v' colored v and w' colored w . That is, if each path *p(v)* in *P* is condensed into a single vertex v and loops (edges of the form {x,x}) are deleted, the result, is tree D. (See Figure 1.)

[Figure 1]

We simulate each of the dynamic tree operations by a constant number of phantom tree operations, as follows:

make-tree(v):

make- tree(v); change- color(v, v).

find-parent(v):

find-color&d-parent(find-top(v*)))*.

find-min-label(v):

find-min-label(find-top(u)).

change-label(v):

change-label(v).

link(v, w):

Step 1. Let u=*find-top*(v). Perform find-children(u). If u has two or fewer children, go to Step 2. Otherwise, find a child q of u (if any) colored v; if there is no such child, let q be any child of u. Let r be new vertex (not in any phantom tree). Perform *make-tree (r); change-color@, v);* *change-label*(r, ∞); *cut*(q); *link*(q, r); *link*(u, r). Replace u by r and go to Step 2.

Step 2. Perform *find-parent*(w); *find-children(w).* If w has valence two or less, let x = w and go to Step 3. Otherwise, choose a child y (if any) colored w; if there is no such child, let y be any child of w. Create a new vertex z (not in any phantom tree). Perform *make-tree(z); change-color+, w); change-label@,* ∞); *cut(y);* *link*(y, z); *link*(w, z). Let x = z; go to Step 3.

Step 3. Perform *link*(u, x).

cut(v):

Step 1. Let u=*find-top*(v) and x=*find-parent*(u). Perform *cut(u).* if u = v, go to Step 2. Otherwise, perform *find-children,(u).* Let q and r be the children of u. Perform *cut(q); cut (r);* *link*(q, r). Destroy dummy vertex u.

Step 2. If 2 = w, stop. Otherwise, find the two vertices y and z adjacent to x by performing *find-parent(x)* and *find-children(x).* If one of y and z, say y, is the parent of x, perform *cut(z); cut(x);* *link*(z, y). Otherwise, perform *cut(z); cut(y);* *link*(z, y). In either case, destroy dummy vertex x and stop.

evert(v):

evert(v).

Each dynamic tree operation consists of O(1) phantom tree operations and 0(1) additional work. Since each dummy vertex in a phantom tree has valence exactly three a. dynamic tree containing k vertices corresponds to a phantom tree containing at most $3k/2$ vertices.

7. Representation of Phantom Trees by Virtual Trees

We implement *phantom trees* by using the method of Sleator and Tarjan [16], modified only as necessary to deal with vertex labels and colors. We assume some familiarity with [16]; we shall merely sketch the details of the implementation, highlighting the changes needed for our purpose. (See also [17], Chapter 5.)

We represent each phantom tree P by a rooted *virtual tree* V, which contains the same vertices as P but has different structure. Each vertex of V has a *left child* and a *right child,* either or both of which can be missing, and at most three *middle children.* We call an edge of V *solid* if it joins a left or right child to its parent and *dashed* otherwise. Tree V consists of a collection of binary trees, its *solid subtrees,* connected by dashed edges. The parent in P of a vertex x is the symmetric-order successor of x in the solid subtree containing x in V, unless x is last in its solid subtree, in which case its parent in P is the parent in V of the root of its solid subtree. (See Figure 2.) That is. each solid subtree in V corresponds to a path in P, with symmetric order in the solid subtree corresponding to the order along the path from deepest to shallowest vertex. We *say a* vertex x is a *solid descendant* of a vertex y in V, and y is a *solid ancestor* of y, if x is a descendant of y and the path from x to y consists of solid edges.

[Figure 2.]

We represent the structure of V by storing with each vertex x pointers to its parent, its left and right children, and a list of its middle children. We also store with x its label and color. In addition, we store with x one piece of cumulative information, *min-label(m),* which is the minimum label of any descendant of x in V. Finally, we store with x a *reversal bit* $rev(x)$, used to handle the *evert* operation. The interpretation of reversal bits is as follows. Let $sum\text{-}rev(x)$ be the mod-two sum of the reversal bits of all solid ancestors

of x. If *sum-rev*(x) is 1, then the meanings of the left and right child pointers of x are reversed, i.e., the left pointer points to the right child and vice-versa.

We use two $O(1)$-time restructuring primitives on virtual trees. The first is *rotation*, in which two vertices x and y joined by a solid edge are interchanged while preserving symmetric order. (See Figure 3.) The second is *splicing,* in which the left child, if any, of a. vertex x is made a middle child, and possibly in addition some middle child is made the left child. (See Figure 4.) A splice can only be performed if x is the root of a solid subtree. It is straightforward to verify that all the values stored at each vertex can be updated in 0(**1)** time after a rotation or a splice.

[Figure 3]

[Figure 4]

The main restructuring operation on virtual trees is *splaying.* A splay at a vertex x consists of a specific sequence of rotations and splices along the path from x to the tree root. The effect of the splay is to restructure the tree, making x the root. The actual time required for a splay at x is proportional to the (original) depth of x; the amortized time is 0(log k) if the tree containing x has k vertices. See [16].

We can perform each of the phantom tree operations using at most two splay operations and 0(1) additional restructuring of the tree. We shall describe the implementation of three of the operations; implementation of the others is similar. (See [16].) To perform *evert*(v), we splay at v, make the left child of v (if any) a middle child, and flip the bit *rev*(v). To perform *find-min-label*(v), we choose a vertex x of minimum *min-label* among v and all its children except the right child. We search clown through descendants of x to find a vertex y such that *label(y)=min-label(x).* (This search is guided by *label* and *min-label* values.) Then we splay at y and return y. The splay at y pays for the search to reach y. We perform *find-top*(v) as follows. First, we splay at v. Then we let $i = 0$ and $v_0 = v$. We repeat the following step until v_i has no right child or v_i differs in color from v: search down from the right child of v_i through left children until reaching a vertex v_{i+1} of the same color as v or that has no left child; replace i by $i + 1$. Once this computation is completed, we splay at v_i and return v_i if it has the same color as v; if it does not, we return v_{i-1}. The splay at v_i pays for all the searching.

With this implementation, the amortized time per phantom tree operation is O(log n).

This implies by the discussion in Section 6 that the amortized time per dynamic tree operation is $O(\log n)$. By the discussion in Section 5, this implies in turn that the amortized time to choose a pivot and implement it in the network simplex algorithm is 0(log n). This gives the main result of our paper:

Theorem 1. The Goldfarb-Hao version of the primal network simples algorithm for the maximum flow problem can be implemented to run in $O(nm \log n)$ time.

8. References

[1] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan, "Improved time bounds for the maximum flow problem," *SIAM J. Comput.*, to appear.

[2] V. Chvatal, *Linear Programming,* W. H. Freeman, New York, 1983.

[3] W. H. Cunningham, "A network simplex method", *Mathematical Programming* 1 (1976), 105-116.

[4] G. B. Dantzig, *Linear Programming and Extensions,* Prince ton University Press, Princeton, NJ, 1963.

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

[6] D. R. Fulkerson and G. B. Dantzig, "Computations of maximal flows in networks", *Naval Research Logistics Quarterly 2* (1955), *277-283.*

[7] A. V. Goldberg, "A new max-flow algorit hm ," Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge. MA, 1985.

[8] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," *J. Assoc. Comput. Mach. 35* (1988), *921-940.*

[9] D. Golclfarb and M. D. Grigoriaclis, "A computational comparison of the Dinic and network simplex methods for maximum flow", *Annals of Operations Research* 13 (1988), 83-123.

[10] D. Goldfarb and J. Hao, "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time," manuscript. Department, of Industrial Engineering and Operations Research, Columbia University, New York, NY, 1988.

[11] M. D. Grigoriadis, "An efficient implementation of the primal simplex method", *Mathematical Programming Study* 26 (1986), 83-111.

[12] J. L. Kennington and R. V. Helgason, *Algorithms for Network Programming,* Wiley, New York, NY, 1980.

[13] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Reinhart, and Winston, New York, NY, 1976.

[14] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall. Englewood Cliffs, NJ, 1982.

[15] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *J. Computer and System Sciences 26* (1983), 362-391.

[16] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J.* Assoc. *Comput. Mach. 32* (1985), 652-686.

[17] R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[18] R. E. Tarjan, "Amortized computational complexity," *SIAM J. Algebraic and Discrete Methods 6* (1985), 306-318.

[19] R. E. Tarjan, "Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem," to appear.

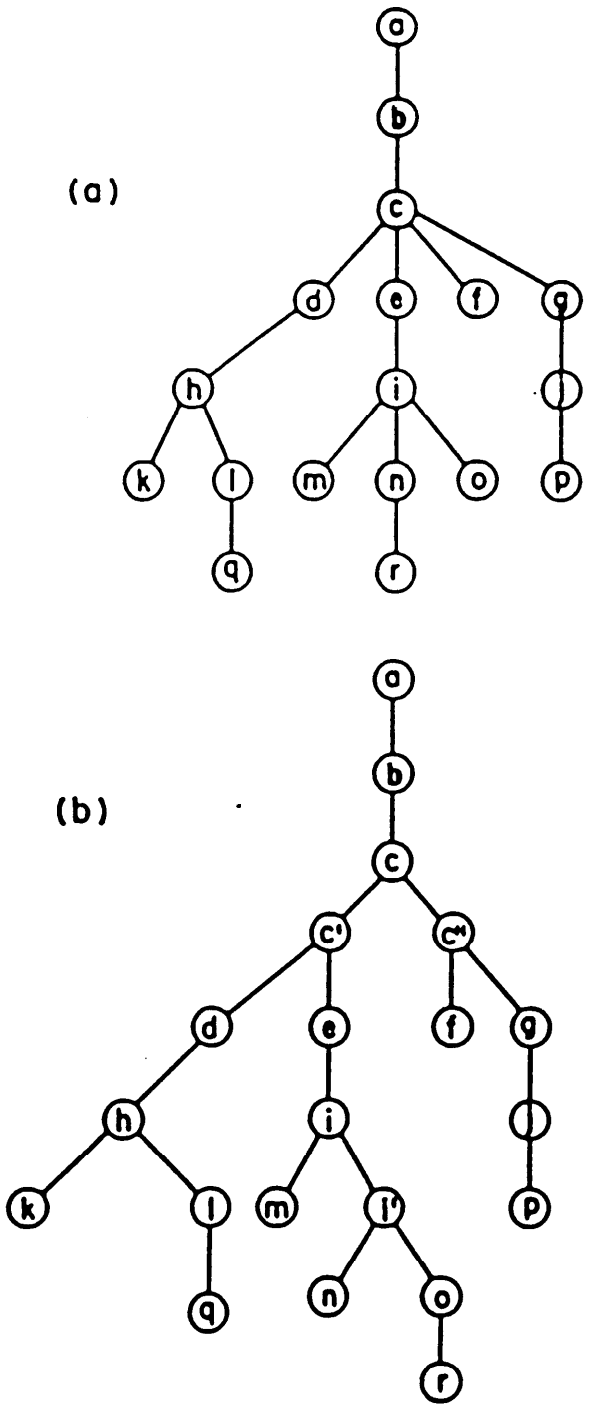


***Figure** 1.* **A dynamic tree and a corresponding phantom tree**
(a) Dynamic Tree D. (b) Phantom Tree P.
Labels inside the vertices of *P* are colors. Primes are added to distinguish vertices.

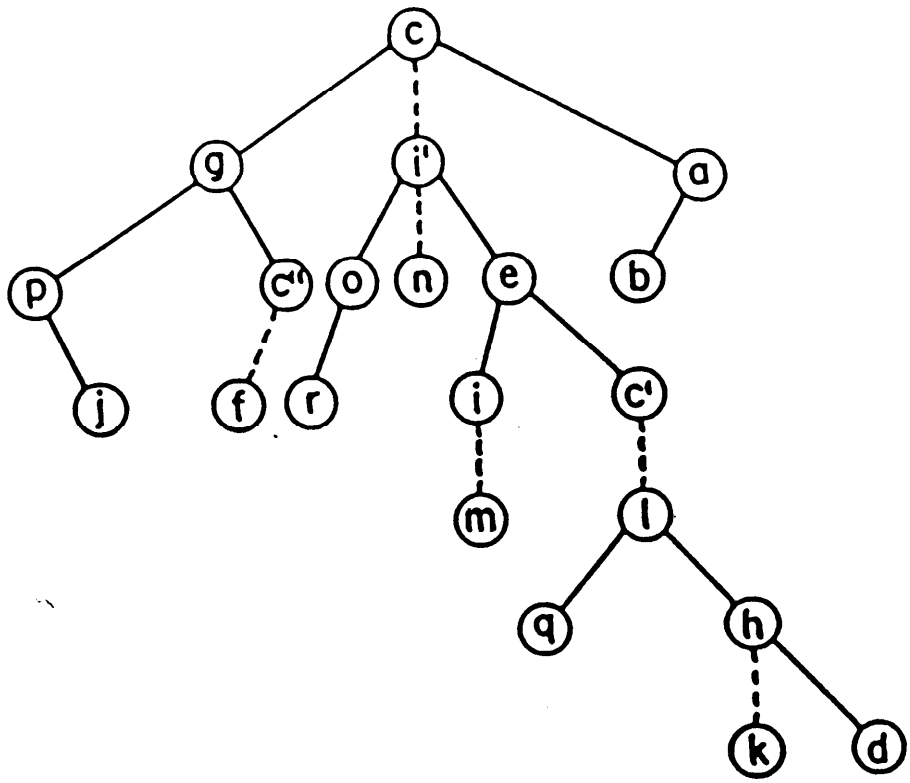


Figure 2. **A virtual tree corresponding to the phantom tree in Figure 1 Solid edges are solid; dashed edges axe dashed.**

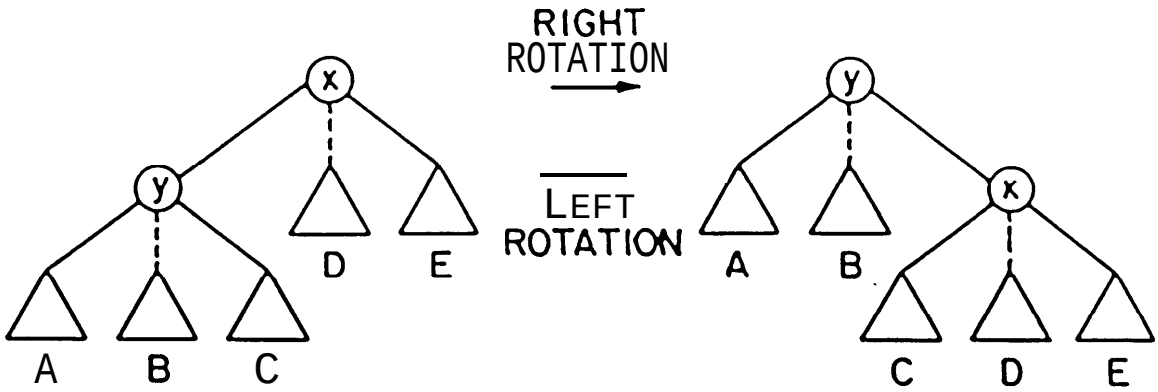


Figure 3. **A rotation in a virtual tree. Triangles denote subtrees.**

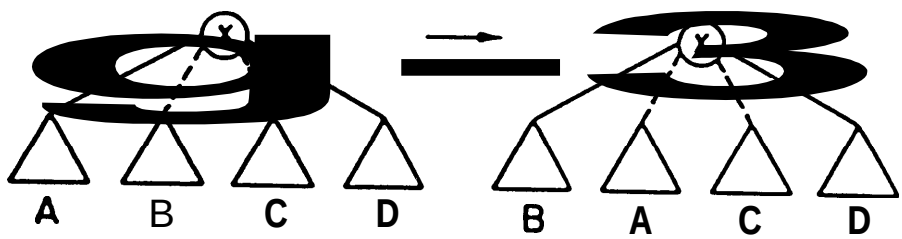


Figure 4. **A splice in a virtual tree.**