Mathematical Programming 50 (1991) 277-290
North-Holland

Use of dynamic trees in a network simplex algorithm for the maximum flow problem

Andrew V. Goldberg*
Department of Computer Science, Stanford University, Stanford, CA 94305, USA

Michael D. Grigoriadis**
Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, USA

Robert E. Tarjan***
Department of Computer Science, Princeton University, Princeton, NJ 08544, USA, and AT&T Bell Laboratories, Murray Hill, NY 07974, USA

Received 22 December 1988
Revised manuscript received 1 October 1989

Goldfarb and Hao (1990) have proposed a pivot rule for the primal network simplex algorithm that will solve a maximum flow problem on an n-vertex, m-arc network in at most nm pivots and $O(n^2m)$ time. In this paper we describe how to extend the dynamic tree data structure of Sleator and Tarjan (1983, 1985) to reduce the running time of this algorithm to $O(nm \log n)$. This bound is less than a logarithmic factor larger than those of the fastest known algorithms for the problem. Our extension of dynamic trees is interesting in its own right and may well have additional applications.

Key words: Algorithms, complexity, data structures, dynamic trees, graphs, linear programming, maximum flow, network flow, network optimization.

1. Introduction

Goldfarb and Hao [12] have proposed a pivot rule for the primal network simplex algorithm that will solve a maximum flow problem on an n-vertex, m-arc network in at most nm pivots and $O(n^2m)$ time. Although this bound is polynomial, it is worse by a factor of almost n than that of the fastest known algorithms for the problem. We consider the question of whether this algorithm can be made more efficient through the use of appropriate data structures and algorithmic techniques. A data structure that suggests itself as being possibly useful is the *dynamic tree*

* 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.
** Research partially supported by the Office of Naval Research, Contract No. N00014-87-K-0467.
*** Research partially supported by the National Science Foundation, Grant No. DCR-8605961, and the Office of Naval Research, Contract No. N00014-87-K-0467.

structure, which Sleator and Tarjan [17, 18] invented for the purpose of speeding up the maximum flow algorithm of Dinic [6]. We show that an appropriate extension of dynamic trees can indeed be used to speed up the Goldfarb-Hao version of the network simplex algorithm, reducing its running time to $O(nm \log n)$. The main novelty in our extension is a way to combine vertex values over subtrees; the original structure only supported combining over paths. Our data structure may have other applications; we leave the discovery of such applications for the future.

Our paper contains six sections in addition to this introduction. Section 2 defines the maximum flow problem. Section 3 describes the generic primal network simplex algorithm and Section 4 reviews the Goldfarb-Hao pivot selection rule. Section 5 describes an efficient way to maintain the vertex labels needed in the Goldfarb-Hao pivot rule, and Sections 6-8 develop an extension of dynamic trees suitable to implement pivot selection. We assume some familiarity with both the Goldfarb-Hao paper and with the version of dynamic trees presented in [18].

2. 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:

$$\text{For every arc } (v, w), \quad f(v, w) = -f(w, v) \quad \text{(antisymmetry constraints)}. \tag{2.1}$$

$$\text{For every arc } (v, w), \quad f(v, w) \leq u(v, w) \quad \text{(capacity constraints)}. \tag{2.2}$$

$$\text{For every vertex } v \notin \{s, t\}, \quad \sum_{w \in E(v)} f(v, w) = 0 \quad \text{(conservation constraints)}. \tag{2.3}$$

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 [10] and of Ahuja et al. [2]. The former runs in $O(nm \log(n^2/m))$ time. The latter, an improvement of an earlier algorithm of Ahuja and Orlin [1], 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 [9]. Extensive discussions of the problem, its applications, and algorithms for it can be found in [7, 15, 16, 19].

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. We 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.

3. A network simplex algorithm

The primal network simplex algorithm that we study is a specialization of the revised simplex method. It is based on an early observation by Fulkerson and Dantzig [8] and Dantzig [5] that any basis matrix of a vertex-arc incidence matrix of a directed graph 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, for example, the books of Chvatal [3] and Kennington and Helgason [14]; for an implementation see Grigoriadis [13].)

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. Our presentation is deliberately terse; for further details, see Goldfarb and Hao's paper [12], which gives a somewhat different formulation. Given a flow f, an arc (v, w) has *residual capacity* $u_f(v, w) = u(v, w) - f(v, w)$. Arc (v, 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, w) 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 *nondegenerate* if there are exactly $n-2$ residual edges and *degenerate* if there are fewer than $n-2$ residual edges.

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, w) with $v \in S$, $w \in Z$:

Pivot. Select a residual arc (v, w) with $v \in S$, $w \in Z$. Add $\{v, w\}$ to $S \cup Z$, 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 residual 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 δ 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 and add it to T, forming a unique simple cycle. Increase the flow on this cycle so that at least one of its edges becomes saturated; delete the saturated edge from T. 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 this edge from T yields a basic flow and a basis S, Z. The running time for this computation can be reduced to $O(m \log n)$ by using the dynamic tree data structure [17, 18, 19], but this does not improve the running time of the overall algorithm.

4. 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 [4], is used for breaking ties in selecting the leaving arc. For integer data, the algorithm with Cunningham's rule 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 [11] proposed a rule that pivots 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 [12] 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 minimum $d(v)$.

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

Choose a vertex $w \in Z$ with minimum $d(w)$. Pivot on any pseudoresidual arc (v, w) with $d(v)=d(w)-1$. (There always must be such an arc. Such an arc will have $v \in S$ and hence will be residual.)

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

5. Efficient implementation of the smallest label rule

We 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 $O(n^2m)$, and is within less than a logarithmic factor of the bounds of the fastest known algorithms.

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 $O(nm)$. 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 [10, 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. Furthermore, 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) is no longer pseudoresidual. If (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 $w \in R$ and delete it from R. Let (v, w) be the current arc of w. (Since w was in R, either (v, w) is no longer pseudoresidual or $d(v) \geq d(w)$.) 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 relabelling 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 $L(w) = \emptyset$, completing the relabelling. If there is no such arc (y, w), then $d(w) = \infty$; the maximum flow computation is complete.

[2] By *amortized time* we mean the time per operation averaged over a worst-case sequence of operations. See [20].

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 $O(nm)$.

6. The use of dynamic trees

To choose pivots and maintain the basis, we extend the dynamic tree data structure of Sleator and Tarjan [17, 18, 19]. 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.

make-tree(v, l): Make vertex v into a one-vertex dynamic tree, with v having label l. 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-value(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 l.

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 $g(v, w) = \alpha$ and $g(w, v) = \beta$. Before the *link* operation, vertices v and w must be in different trees, with v the root of its tree.

cut(v): 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* and $n-2$ *link* 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 *find-min-label*(t), which returns a vertex in Z, say w, of smallest label. We pivot on the current arc (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, α, 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. We show in Sections 7 and 8 how to obtain an O(log n) amortized time bound per tree operation. Then the amortized time per pivot is O(log n), so the overall running time of the network simplex algorithm is O(nm log n), as desired.

7. 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 k). Obtaining such an implementation requires extending the Sleator–Tarjan data structure. Two extensions are needed. The first is to allow each tree edge $\{v, w\}$ to have two associated values, $g(v, w)$ and $g(w, v)$, rather than just one. This extension is reasonably straightforward and is described in detail in [21]; we can use the data structure described there to handle all the operations except *find-min-label* and *change-label*. The more interesting extension and 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-label* requires combining values over subtrees. 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 use a separate data structure of the kind described in [21].

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 vertex in a phantom tree, henceforth called a *p-vertex*, has a *label*, which may change, and a fixed *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: *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:

make-tree(v, l, γ): Make vertex v into a one-vertex phantom tree; set the label of v equal to l and the color of v equal to γ.

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

find-top(v): Find the ancestor w of v closest to the tree root such that all vertices on the path between v and w (including v and w) have the same color.

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. Such a path consists in general of two vertices x and y, their least common ancestor z, and every ancestor of either x or y that is a descendant of z. 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 ∞ 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.)

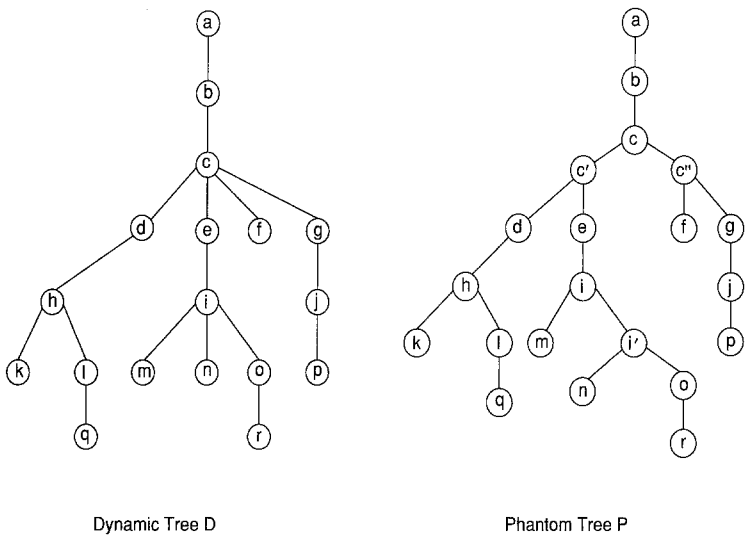

Fig. 1. A dynamic tree and a corresponding phantom tree. Labels inside the vertices of P are colors. Primes are added to distinguish vertices.

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

make-tree(v, l):
  make-tree(v, l, v).
find-parent(v):
  find-color(*find-parent*(*find-top*(v))).

find-min-label(v):
  find-min-label(*find-top*(v)).
change-label(v, l):
  change-label(v, l).
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 a new vertex (not in any phantom tree). Perform *make-tree*(r, ∞, v); *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, ∞, w); *cut*(y); *link*(y, z); *link*(z, w). 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, with r colored v. Perform *cut*(q); *cut*(r); *link*(q, r). Destroy dummy vertex u.

Step 2. If $x = w$, 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, at least one of y and z, say y, is colored w; 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 O(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 $2k$ vertices.

8. Representation of phantom trees by virtual trees

We implement *phantom trees* by using the method of Sleator and Tarjan [18], modified only as necessary to deal with vertex labels and colors. We assume some familiarity with [18]; we shall merely sketch the details of the implementation, highlighting the changes needed for our purpose (see also [19, 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, any y is a *solid ancestor* of x, if x is a descendant of y and the path from x to y consists of solid edges.

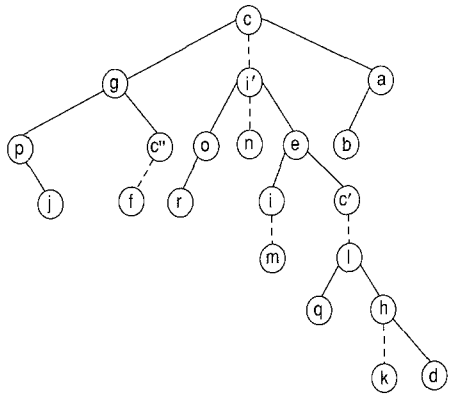

Fig. 2. A virtual tree corresponding to the phantom tree in Figure 1.

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 two pieces of cumulative information, *min-label*(x), which is the minimum label of any descendant of x in V, and *unicolor*(x), which is a bit equal to 1 if x and all its descendants in the same solid subtree are the same color and equal to 0 otherwise. 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-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 O(1) time after a rotation or a splice.

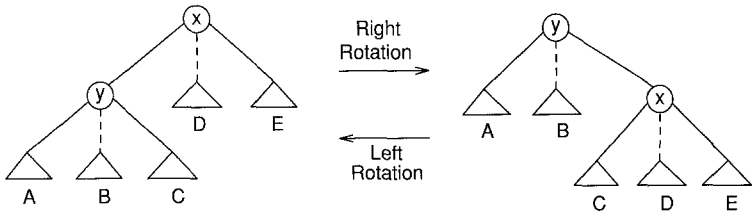

Fig. 3. A rotation in a virtual tree (triangles denote subtrees).

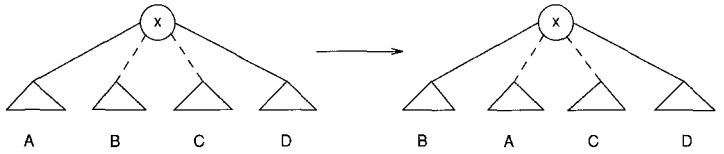

Fig. 4. A splice in a virtual tree.

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 $O(\log k)$ if the tree containing x has k vertices.

There are various ways to specify the details of a splay operation so as to achieve the $O(\log k)$ amortized bound. We shall describe the method of [18], which uses three bottom-up passes to move x to the root. The first and third passes do rotations exclusively, the second pass does splices exclusively. The first pass consists of initializing $a = x$ and repeating the following step until a is the tree root:

Rotation Step: Let b the parent of a and c the parent of b (if any). Apply the appropriate one of the following cases (see Figure 5):

Skip: If the edge joining a to b is dashed, replace a by b.

Zig: If the edge joining a to b is solid and either c does not exist or the edge joining b to c is dashed, rotate the edge joining a to b.

Zig-Zig: If c exists, the edges joining a to b and b to c are solid, and a and b are both left or both right children, rotate the edge joining b to c and then rotate the edge joining a to b.

Zig-Zag: If c exits, the edges joining a to b and b to c are solid, and exactly one of a and b is a left child, rotatee the edge joining a to b and then rotate the edge joining b to c.

The first pass eliminates all solid edges on the path from x to the virtual tree root; that is, after the first pass the entire path from x to the virtual tree root consists of dashed edges. The second pass consists of splicing each edge along the path from x to the root, proceeding bottom-up, thereby making the entire path solid. (See Figure 6.) The third pass is identical to the first pass, but it does no "skip" steps because there are no longer any dashed edges on the path from x to the root.

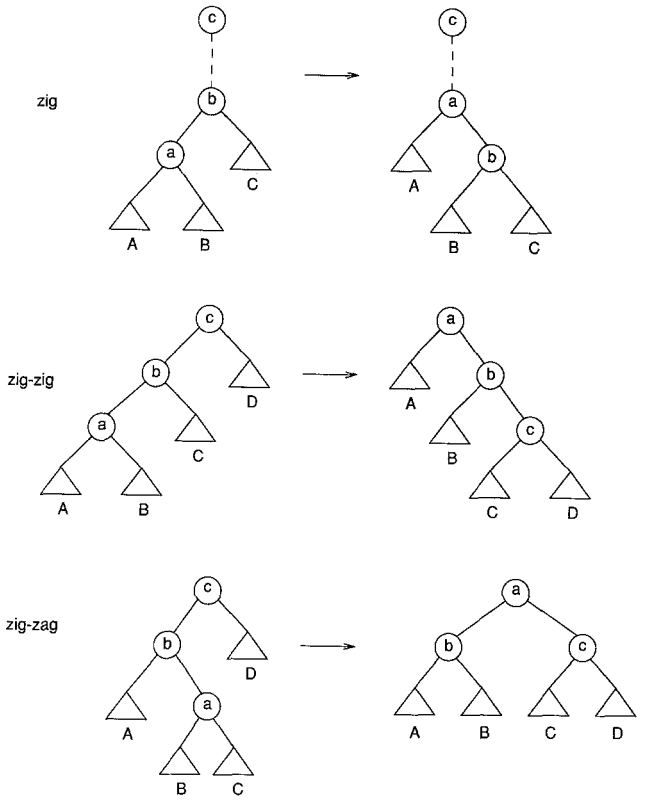

Fig. 5. The cases of a rotation step (not including *skip*).

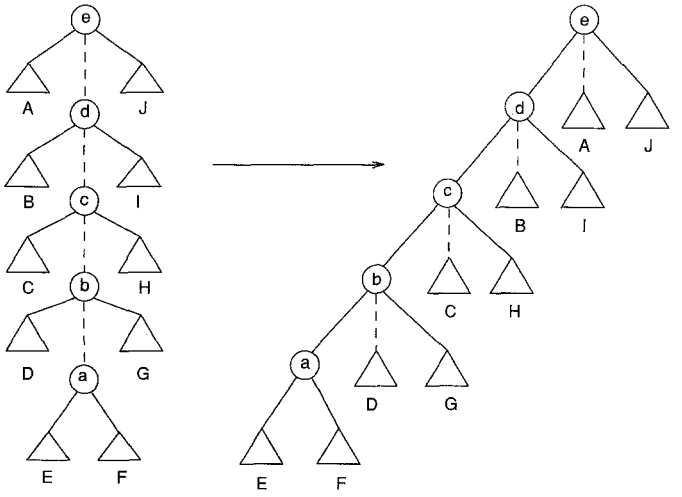

Fig. 6. The second pass of a splay operation, consisting of splicing repeated bottom-up.

At the end of the third pass, x is the root of the entire virtual tree. The details of the timing analysis of splaying are not straightforward; they can be found in [18].

We perform each of the phantom tree operations using at most two splay operations on the virtual tree(s) and O(1) additional restructuring, as follows:

make-tree(v, l, γ): Make v into a one-vertex virtual tree, with label l and color γ.

find-parent(v): Splay at v. If v has no right child, return null. Otherwise, follow left child pointers from the right child of v until reaching a vertex w with no left child. Splay at w and return w.

find-children(v): Splay at v. If v has a left child, follow right child pointers from the left child of v until reaching a vertex w with no right child. Return w if it exists and also return all middle children of v. If w exists, splay at w.

find-min-label(v): Splay at v. Let l be the minimum among the label of v and the *min-label* values of the left and middle children of v. If $l = label(v)$, return v. Otherwise, let w be a left or middle child of v with $l = min\text{-}label(w)$. Walk down from w through the tree along a path of vertices all of whose *min-label* values are l, until reaching a vertex x with $l = label(x)$. Splay at x and return x.

find-top(v): Splay at v. Let $i = 0$ and $v_0 = v$. Repeat the following step until v_i has no right child or v_i differs in color from v:

Descend: Follow left child pointers down from the right child of v_i until reaching a vertex v_{i+1} such that either v_{i+1} has no left child, or v_{i+1} has a left child, both v_{i+1} and its left child have the same color as v, and the unicolor bit of the left child of v_{i+1} is 1. Replace i by $i+1$.

Complete the *find-top* operation by splaying at v_i and returning v_i if it has the same color as v or v_{i-1} if it does not.

change-label(v, l): Splay at v. Set the label of v equal to l and recompute *min-label*(v) by using the *min-label* values of the children of v.

evert(v): Splay at v. Make the left child of v (if any) a middle child by doing a splice. Flip the bit $rev(v)$.

cut(v): Splay at v. Break the edge between v and its right child. Recompute *unicolor*(v) and *min-label*(v).

link(v, w): Splay at v. Splay at w. Make v a middle child of w. Recompute *unicolor*(w) and *min-label*(w).

Verifying the correctness of each of these implementations is straightforward. The most interesting operation is *find-top*. Its correctness follows from the invariant (proved by induction on i) that every vertex in the same solid subtree as v and between v and v_{i+1} (possibly not including v_{i+1}) has the same color as v, and if v_{i+1} has the same color as v then either some descendant of v_{i+1} has a different color than that of v, or v_{i+1} is a left child and its parent has a different color than that of v.

The time for the splay operations in *find-parent*, *find-children*, *find-min-label* and *find-tree* dominates the time for the searching in each of those operations (to within

a constant factor). The amortized time per phantom tree operation is $O(\log k)$. This implies by the discussion in Section 7 that the amortized time per dynamic tree operation is $O(\log k)$. By the discussion in Section 6, this implies in turn that the amortized time in the network simplex algorithm to choose a pivot and to perform it is $O(\log n)$. This gives the main result of our paper:

Theorem 1. *A primal network simplex algorithm for the maximum flow problem that uses the Goldfarb–Hao pivot selection rule can be implemented to run in* $O(nm \log n)$ *time.* □

References

[1] R.K. Ahuja and J.B. Orlin, "A fast and simple algorithm for the maximum flow problem," *Operations Research* 37 (1989) 748–759.

[2] R.K. Ahuja, J.B. Orlin and R.E. Tarjan, "Improved time bounds for the maximum flow problem," *SIAM Journal on Computing* 18 (1989) 939–954.

[3] V. Chvatal, *Linear Programming* (Freeman, New York, 1983).

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

[5] G.B. Dantzig, *Linear Programming and Extensions* (Princeton University Press, Princeton, NJ, 1963).

[6] E.A. Dinic, "Algorithm for solution of a problem of maximum flow in networks with power estimation," *Soviet Mathematics Doklady* 11 (1970) 1277–1280.

[7] L.R. Ford, Jr. and D.R. Fulkerson, *Flows in Networks* (Princeton University Press, Princeton, NJ, 1962).

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

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

[10] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *Journal of the Association of Computing Machinery* 35 (1988) 921–940.

[11] D. Goldfarb and M.D. Grigoriadis, "A computational comparison of the Dinic and network simplex methods for maximum flow," *Annals of Operations Research* 13 (1988) 83–123.

[12] 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,", *Mathematical Programming* 47 (1990) 353–365.

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

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

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

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

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

[18] D.D. Sleator and R.E. Tarjan, "Self-adjusting binary search trees," *Journal of the Association of Computing Machinery* 32 (1985) 652–686.

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

[20] R.E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic and Discrete Methods* 6 (1985) 306–318.

[21] R.E. Tarjan, "Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem," to appear in: *Mathematics of Operations Research* (1991).