[12] H. M. Martinez (ed.), Mathematical and computational problems in the analysis of molecular sequences, *Bull. Math. Biol.*, **46**, 4 (1984).
[13] W. J. Masek and M. S. Paterson, A faster algorithm for computing string editing distances, *J. Comput. System Sci.*, **20** (1980), 18–31.
[14] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, EATCS Monographs on TCS, Springer-Verlag, Berlin, 1984.
[15] D. Sankoff and J. B. Kruskal (eds.), *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparisons*, Addison-Wesley, Reading, MA, 1983.
[16] P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.*, **6** (1977), 80–82.
[17] R. A. Wagner and M. J. Fischer, The string to string correction problem, *J. Assoc. Comput. Mach.*, **21** (1974), 168–173.

Algorithmica (1987) 2: 337–361

Algorithmica
© 1987 Springer-Verlag New York Inc.

Computing on a Free Tree via Complexity-Preserving Mappings[1]

Bernard Chazelle[2]

Abstract. The relationship between linear lists and free trees is studied. We examine a number of well-known data structures for computing functions on linear lists and show that they can be canonically transformed into data structures for computing the same functions defined over free trees. This is used to establish new upper bounds on the complexity of several query-answering problems.

Key Words. Linear list, Free tree, Query-answering, Range tree, Interval tree, Ackermann's function.

1. Introduction. In the same way as pushdown automata generalize finite state automata, free trees generalize linear lists. This generalization relates these combinatorial objects together as well as the abstract data types that are built upon them. For example, consider the obvious similarity between an *interval* in a list and a *path* in a tree. The fact that in both cases two pieces of data fully specify the object in question allows us to rephrase many problems on linear lists by mere substitution of the word tree (resp. path) for the word list (resp. interval). Maintaining intervals dynamically (a typical task in IC design-rule checking, for example) becomes the problem of maintaining "active" paths in a tree (with application to, say, communication networks). Since, after all, linear lists are better understood than free trees, it is tempting to ask the question: can one use techniques available for linear lists to solve problems on trees, or does the added generality of the latter type prevent any systematic "transfer"?

Let us say now that we concern ourselves only with *query-answering* problems, that is, problems where a function is to be evaluated repeatedly for different values of its arguments, presented as *queries*. We show that many data structuring techniques defined in the context of linear lists can be generalized to work also on free trees. Moreover, the generalizations can be defined in a *canonical* fashion, that is, by using a small number of elementary transformations, quite independent of the problems to be solved.

A different approach to the problem of computing over free trees has been developed by Sleator and Tarjan [12], [13]. It can be used for some of the problems addressed later on; being particularly efficient when certain dynamic

[1] This work was started when the author was at Brown University, Providence, RI. It was partly supported by NSF Grant MCS 83-03925. A preliminary version of this work has appeared in the *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, West Palm Beach, FL, October 1984, pp. 358–368.
[2] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA.

Received April 24, 1985; revised May 8, 1986, and December 19, 1986. Communicated by D. T. Lee.

capabilities are needed (such as linking or cutting trees) and amortized complexity is acceptable. Roughly, Sleator and Tarjan's method involves rewriting a rooted tree as a "balanced" interconnection of paths.

Our approach is quite different. Instead of developing original data structures for free trees, we build a framework within which algorithms for linear lists can be ported to free trees with little effort. In the process, we establish new upper bounds on the complexity of several query-answering problems. One of the interesting findings of this work is that many optimal data structures can be indeed generated in that way. Here is a summary of our main results:

In Section 2 we make several basic observations and establish a few technical lemmas to be used throughout.

In Section 3 we show how Cartesian trees (a data structure due to Vuillemin [16]) can be adapted to operate on free trees. This leads to efficient methods for solving problems of the kind: preprocess a weighted free tree so as to be able to compute the minimum weight over an (arbitrary) query path. This, we show, can be done in constant time and linear space. In the same section we also consider the problem of storing a collection of distinguished paths in a tree, so that given a query edge all the paths passing through it can be reported efficiently. We solve this problem by generalizing the notion of an interval tree [5], [10]. By studying two simple examples this section illustrates some of the basic tools used later.

In Section 4 we return to the first problem of the previous section, but look at it in a more general setting. Now, weights are elements of an arbitrary (fixed) semigroup and queries ask for the cumulated weight of a given path. We describe an optimal solution to this problem by generalizing a technique developed by Yao for the case of linear lists [18]. As an application, we rederive a result of Tarjan [15] stating that a minimum spanning tree of a graph of n vertices and m edges can be verified in time $O(m\alpha(m, n))$, where α is the inverse of Ackermann's function. Our method is interesting because it presents the relation between minimum spanning tree verification and Ackermann's function in a completely new light.

In Section 5 we consider the following variant of the problem: each edge is assigned a real number (its weight), and a query specifies a path P as well as an interval on the real line; the problem is to determine how many edges in P have their weights in the given interval. We give an optimal solution to this problem, which we develop in two stages: first, we generalize the notion of a range tree [2]; then we apply a compaction scheme [3] to reduce the size of the data structure.

In Section 6 we close this paper with concluding remarks and open problems.

Throughout this paper, we assume that the underlying model of computation is a RAM, as defined in (Aho *et al.* [1]).

2. Preliminaries.

We begin with a few technical results which we shall use repeatedly later. Let T be a free tree, defined as an undirected, connected, acyclic graph [8], and let v be one of its vertices. If v has at least two adjacent vertices, then split v into two disconnected vertices v_1 and v_2, and reconnect to either v_1 or v_2 each vertex formerly adjacent to v. We assume that in the process both v_1

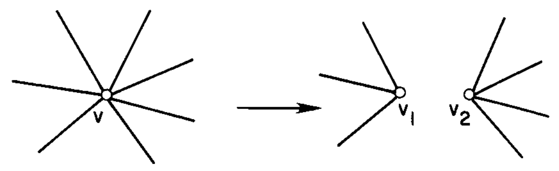


Fig. 1. Decomposing a free tree.

and v_2 get reconnected to at least one vertex (Figure 1). This decomposes T into two subtrees T_1 and T_2, partitions the edge-set of T, and adds one to the total vertex-count. We use the term *subtree* to refer to any connected subgraph of T. We define a partition of T as any set of subtrees of T which forms a partition of its edge-set. Finally, let $|T|$ denote the number of edges of T. The following lemmas are standard facts about free trees. We include proofs for the sake of completeness, aiming at simplicity rather than optimality.

LEMMA 1. *It is possible to preprocess a free tree T in linear time so that for any pair of vertices u, v and any edge e in T, one can determine in constant time whether e is an edge of the path between u and v.*

PROOF. Perform a depth-first traversal of T. Label the first edge traversed 1, the second edge 2, etc. Since each edge e is traversed twice, first forward and then backward, it will be labeled twice, the labels being $f(e)$ and $g(e)$, with $1 \leq f(e) < g(e) \leq 2|T|$ (Figure 2). Label each vertex v by $l(v) = \min\{f(e) | e \text{ adjacent to } v\}$; informally, $l(v)$ denotes the first time v is visited. To ensure that each vertex has a distinct label, we set $l(w) = 0$, where w is the first vertex visited. Clearly, edge e lies on the path from u to v if and only if exactly one of u or v is visited between the two visits of e, that is, either $l(u) \in [f(e), g(e)]$ and $l(v) \notin [f(e), g(e)]$ or $l(v) \in [f(e), g(e)]$ and $l(u) \notin [f(e), g(e)]$. Note that no vertex can be labeled $g(e)$ for any e. $\square$

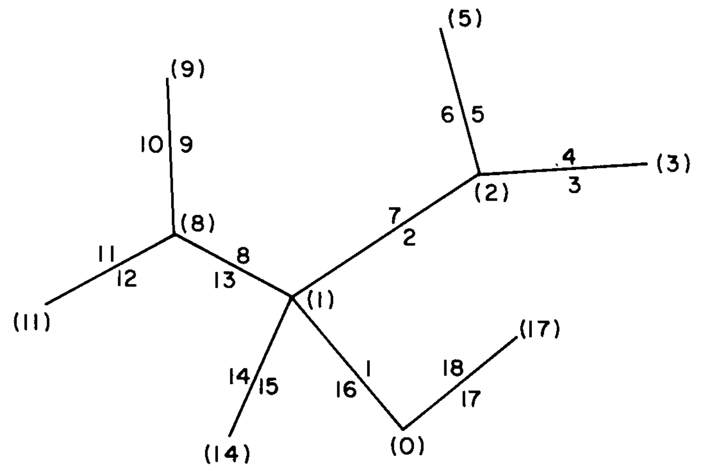


Fig. 2. Labeling edges and vertices.

Lemma 2. *If T is a free tree of at least two edges then, in linear time, it is possible to partition T into two subtrees T_1 and T_2 such that $\frac{1}{3}|T| \le |T_1| \le T_2| \le \frac{2}{3}|T|$.*

Proof. The existence part of the lemma is the standard centroid theorem. To implement the partitioning in linear time, we can use the labeling of Lemma 1. It suffices to notice that, for each edge e of T, $g(e)-f(e)-1$ represents twice the size of one of the subtrees adjacent to e. This allows us to discover the centroid(s) in linear time. Since none of the subtrees adjacent to the centroid(s) can contain more than $\lceil |T|/2 \rceil \le 2|T|/3$ edges of T, completing the proof is routine. □

Lemma 3. *Let T be a free tree and let k be any integer $(1 \le k \le |T|)$. In linear time it is possible to partition T into subtrees $T_1, \ldots, T_p$, such that for each i $(1 \le i \le p)$ we have $k/3 < |T_i| \le k$.*

Proof. Choose an arbitrary vertex r of T and represent T as a tree rooted at r, with a pointer from each node $(\ne r)$ to its parent. We also keep a separate list of the leaves. In addition, we associate two registers, $b(v)$ and $c(v)$, with each node v: initially, $b(v)=0$ and $c(v)$ holds the number of children of v. We assume that we have an unbounded supply of colors to choose from to color nodes and edges. At the outset, each node and each edge is given a distinct (but arbitrary) color. At the end, the edge-coloring indicates the desired partition. The algorithm iterates on the following process.

Pick any leaf v; if $v=r$ then stop. Otherwise, let w be the parent of v in the tree. Decrease $c(w)$ by 1 and increase $b(w)$ by $1+b(v)$. Also, assign the color of v to (v, w) as well as to every edge colored like w. Prune v from the tree. If $b(w) \le k/3$, then give w the color of v; otherwise set $b(w)$ to 0 and color w with a new, as yet unused, color.

The algorithm is straightforward. At any time, $b(w)$ holds the size of the subtree rooted at w, colored the same way. This size never increases by more than $1+b(v) \le 1+k/3$, therefore $b(w) \le 1+2k/3$. (Note that since the lemma is obviously true for $k<3$, we can assume that $k \ge 3$, which implies $b(w) \le k$.) If at the end we find that $b(r) \le k/3$, then the last subtree is too small. Since $k/3 < |T|$ it cannot be the only subtree produced, so we can merge it with any of the adjacent ones to form another tree t. Note that $|t| \le k+k/3$. If $|t| \ge k+1$ then we can apply Lemma 2 and decompose t into two subtrees t_1, t_2, with $(k+1)/3 \le |t_1| \le |t_2| \le 8k/9$, which then satisfies the conditions of this lemma.

A few remarks concerning the implementation: $c(v)$ is used to detect when a node becomes a leaf; the recoloring of subtrees can be performed in $O(1)$ time by implementing each color as pointers to the head and the tail of a separate linked list (of edges). This ensures the claimed linear time. □

3. Two Simple Generalizations: Cartesian Trees and Interval Trees.

We begin with two examples of data structures developed in the context of linear orders, which it is particulary easy to *port* to free trees. The first one is the *Cartesian tree*, a data structure invented by Vuillemin [16] as a geometric representation

of a permutation. Let $A_1, \ldots, A_n$ be n distinct real values. We build the Cartesian tree $T(1, n)$ of this set by using the following rewriting rules: if $i>j$, then $T(i,j)$ is null. Otherwise, let A_k be the largest value among $A_i, \ldots, A_j$; the left (resp. right) subtree of the root of $T(i,j)$ is $T(i, k-1)$ (resp. $T(k+1, j)$). Gabow *et al.* [6] have observed that one-dimensional range searching for maximum can be reduced to computing a nearest common ancestor in a Cartesian tree. We apply these techniques to the following problem: let T be a free tree where each edge is assigned a weight taken from a totally ordered set; given a query path P, compute the maximum edge-weight in P.

The data structure $\mathcal{T}(T)$ associated with T is built recursively as follows: if T is reduced to a single vertex then $\mathcal{T}(T)$ is null. If not, let e be an edge of T of maximum weight, and let T_1 and T_2 be the two trees created by the removal of e (note that T_1 and T_2 consist of at least one vertex). We define $\mathcal{T}(T)$ as a binary tree whose root is associated with e, and whose two subtrees hanging from the root correspond respectively to T_1 and T_2 (the order is immaterial). Let v_1 and v_2 be the two nodes of $\mathcal{T}(T)$ corresponding to the end-edges of the query path P. Obviously, the edge of T associated with the nearest common ancestor of v_1 and v_2 has maximum weight among the edges of P. It can be found in constant time and linear preprocessing, using a technique due to Harel and Tarjan [7]. One last detail concerns v_1 and v_2. If P is represented by its two end-vertices (w_1, w_2), finding the end-edges might not be immediate (see Lemma 15). What we can do instead is keep a correspondence table between each vertex of T and the smallest-weight edge adjacent to it. In this way we may not have the right end-edges, but certainly the new path will have the same maximum edge-weight as the original one. Of course, the scheme may fail if not only the maximum weight of P is sought, but also the name of an edge carrying that weight. In that case it might then be better prior to preprocessing, to attach a dummy edge of weight $-\infty$ to each vertex of degree larger than one, and keep a correspondence table between each such vertex and the dummy edge adjacent to it. In this way the path actually processed will not be P, but its edge of maximum weight will be the same (assuming that $-\infty$ is any value less than any edge-weight).

We shall now apply the same method to *interval trees* and enable them to handle paths in a free tree, as opposed to intervals in a linear list. The problem which we wish to solve is the following: given a collection of paths in a free tree, report all the paths which pass through an arbitrary query edge. We briefly recall the definition of an interval tree, a data structure discovered independently by Edelsbrunner [5] and McCreight [10]. Given a set of intervals in $\{1, \ldots, n\}$, set up a complete binary tree of n nodes and store each interval I in the highest node of the tree whose rank in symmetric order falls within I. The generalization to free trees is immediate. Let T be a free tree and let C be a collection of paths in T; for convenience let m denote the size of T or C, whichever is larger. The data structure $\mathcal{T}(T)$, as usual, is defined recursively. If T consists of a single edge then $\mathcal{T}(T)$ is a tree with a single node, associated with the edge of T. Otherwise, let v be the vertex of T common to T_1 and T_2, as defined in Lemma 2. We *replace* v by a dummy edge, associated with the root of $\mathcal{T}(T)$. The replacement of v by a dummy edge (along with the recursive occurrences of this process) has the effect of creating a new tree T', whose edges are in one-to-one correspondence with the nodes of $\mathcal{T}(T)$. Note that the internal nodes correspond to dummy edges, whereas the leaves map to the edge-set of T. The second observation is that each vertex of T' can be associated with a unique vertex of T. Since this correspondence is in general not one-to-one, there might be more than one way to specify T'. Indeed, let T_1' and T_2' be the new trees derived from T_1 and T_2, respectively. We can form T' by connecting T_1' and T_2' via *any* edge (v_1, v_2), provided that v_1 (resp. v_2) is a vertex of T_1' (resp. T_2') corresponding to the vertex v of T. Clearly, there might be several ways to do so. We shall examine the correspondence between T and T' in greater detail in Section 5. We have enough at our disposal at this point for solving our current problem. Any edge of T is mapped into a unique edge of T', hence query edges in T can be interpreted as query edges in T'. Also, a vertex of T is mapped into one of the (possibly several) vertices of T' associated with it. This allows us to transform C into a collection C' of paths in T'. We can easily verify that the new problem—defined with respect to T' and C'—is exactly the same as the former one (despite the flexibility in the mapping of vertices from T to T'). Next, following the approach of the interval tree, we *insert* each path of C' into the highest node of $\mathcal{T}(T)$ whose associated edge lies on the path. Applying the labeling of Lemma 1 to T' allows us to classify the paths stored at each node v of $\mathcal{T}(T)$. Let T_1' and T_2' be the two subtrees of T' associated with the children of v (assuming that v is not a leaf). We form the lists $L_1(v)$ and $L_2(v)$ consisting respectively of the end-vertices in T_1' and T_2' of the paths stored at v. Both lists are sorted by the labels of the end-vertices. If v is a leaf then $L_1(v)$ contains the names of the single-edge paths of C' stored at v and $L_2(v)$ is not defined. The data structure requires linear storage.

Let e be a query edge. To retrieve all the paths of C containing e, we first map e into its corresponding edge e' in T'. Then we observe that by binary search we can identify a path of $\mathcal{T}(T)$ from the root to a leaf, whose lists $L_i(v)$ are guaranteed to contain the end-vertices of the paths sought. We can implement this first stage of the query-answering in $O(\log m)$ time. Our next observation is that within each list $L_i(v)$ thus identified the desired entries can be retrieved, each in constant time, after a binary search in $L_i(v)$ with respect to one of the two labels associated with e'. This leads to a query time of $O(k+\log^2 m)$, which we improve to $O(k+\log m)$ by using *fractional cascading*. This data structuring technique, due to Chazelle and Guibas [4], is a postprocessing method to optimize the search for a given key in a set of dictionaries connected together by a graph of bounded degree. It increases the preprocessing by at most a constant factor. This closes our informal discussion of generalized intervals trees.

The main purpose of this section has been to illustrate, in their simplest form, some of the mappings used later. Whether our generalization of an interval gives the solution of choice for the problem considered is unclear. Other solutions can be contemplated. One is to use Lemma 1 to express the problem as a specific instance of range searching, and then use McCreight's *priority search tree* [11]; we omit the details. Another solution is to recast the approach followed here in the framework proposed by Sleator and Tarjan [12].

We are now ready for a more complex illustration of the *transformational* approach which constitutes the main theme of this work.

4. Summing Up Weights Along a Query Path.

Let T be a free tree of n edges and let w be a weight function mapping every edge e of T into an element $w(e)$ of a semigroup $(\mathcal{S}, +)$. The problem which we consider here is computing the function $\sum_{e\in P} w(e)$, for an arbitrary query path P of T. We first look at the problem from a purely combinatorial point of view. We show that if m values can be computed in preprocessing, then any sum of the form $\sum_{e\in P} w(e)$ can be evaluated in $O(\alpha(m, n)+n/(m-n+1))$ additions (in the sense of $\mathcal{S}$), where α is the inverse of Ackermann's function defined by Tarjan [14]. Then we show how to implement this combinatorial scheme on a RAM. Finally, we give an application of this result to the verification of minimum spanning trees.

4.1. A Combinatorial Solution

THEOREM 1. *Let T be a free tree of n edges with an edge-weight function w. For any integer $m \geq n$ there exists a set W of m semigroup values such that, for any path P of T, the quantity $\sum_{e\in P} w(e)$ can be expressed as the sum of $O(\alpha(m, n)+n/(m-n+1))$ values in W.*

To prove this theorem we generalize a technique of Yao [18] for computing partial sums in a linear list. We need some terminology. Let $W=\{q_1, \ldots, q_m\}$ be a set of m values in the semigroup and let $F(u, v)$ denote the quantity $\sum_{e\in P} w(e)$, where P is the path between the vertices u and v. The set W provides a (t, α)-scheme (α real >0) if $m \leq \alpha n$ and any $F(u, v)$ can be expressed as $\sum_{1\leq k\leq t'} q_{i_k}$, where $t' \leq t$.

From Lemma 3, we derive the two relations,

$$\frac{k}{3} < |T_i| \leq k, \tag{1}$$

$$\frac{|T|}{k} \leq p < \frac{3|T|}{k}. \tag{2}$$

Let $\mathcal{T}=\{T_1, \ldots, T_p\}$ denote the partition of Lemma 3. Informally speaking, the subtrees of $\mathcal{T}$ can be regarded as the *super nodes* of a free tree. This allows us to classify any path P in T as either falling entirely within a super node or stretching over several super nodes. Computing F in the first case can be done by a recursive call, while in the other case, precomputed values of F associated with each super node will be sufficient to evaluate F at P. The remainder of this discussion is devoted to developing this argument in more detail and analyzing the complexity of the algorithm.

Distancing ourselves a little from Lemma 3, let us look at $\mathcal{T}$ as an arbitrary set of trees, $\{T_1, \ldots, T_p\}$, with distinct vertices. Suppose that we join together a number of vertices chosen among the T_i, so as to produce a single tree T. This process involves starting with T empty, and then picking each T_i, one at a time, and identifying one of their vertices with a vertex of T. Note that $\mathcal{T}$ will thus constitute a partition of T. We call the set of vertex-pairs joined during the process an *interconnection pattern* for $\mathcal{T}$. Let $C(\mathcal{T})$ denote the set of all interconnection patterns for $\mathcal{T}$. For any $R \in C(\mathcal{T})$, we have a unique resulting tree, T, which we can thus designate by the functional notation, $\mathcal{T}_R$. For any i $(1 \leq i \leq p)$, consider the subset of vertices in T_i which contribute an entry to R, i.e. those vertices merged to some others in T_j $(j \neq i)$. The set, called the *fringe* of T_i, induces a subtree of T_i, denoted Q_i (i.e, the Steiner minimal tree of the fringe). Unfortunately, Q_i may contain an excessive number of redundant edges. To remedy this flaw, we consider each vertex of degree 2 in Q_i which is not in the fringe, and merge its two adjacent edges. The resulting tree, Q_i^*, is homomorphic to Q_i, and every edge of Q_i^* maps to a path of Q_i. Note that we may easily extend the domain of F to include the paths of Q_i^* so that, for any two vertices u, v in the fringe of Q_i, the value of $F(u, v)$ can be computed directly from Q_i^*. Note that this is in general not true if u and v are arbitrary vertices of Q_i. The union of all Q_i^*, for $i = 1, \ldots, p$, is easily shown to form a free tree, denoted $\mathcal{T}_R^*$.

We are now ready to describe the underlying data structure, denoted $\mathrm{DS}(T)$. From now on, let $\mathcal{T} = \{T_1, \ldots, T_p\}$ denote the partition of T given in Lemma 3 and R designate the underlying interconnection pattern. Let V_i be the set of values $F(u, v)$, for all pairs (u, v) where u is any vertex of T_i and v is a vertex on the fringe of T_i. We define $\mathrm{DS}(T)$ recursively, as follows $\mathrm{DS}(T) = (A, B, C)$, with $A = \{\mathrm{DS}(T_1), \ldots, \mathrm{DS}(T_p)\}$, $B = \mathrm{DS}(\mathcal{T}_R^*)$, and $C = \{V_1, \ldots, V_p\}$. The algorithm for computing $F(u, v)$ is now clear. Let T_i (resp. T_j) be the unique tree of $\mathcal{T}$ that contains the edge adjacent to u (resp. v) on the path between u and v. If $i = j$, recurse on $\mathrm{DS}(T_i)$. Otherwise, let f_u (resp. f_v) be the first (resp. last) fringe vertex on the directed path from u to v. The relation $F(u, v) = F(u, f_u) + F(f_u, f_v) + F(f_v, v)$ allows us to evaluate $F(u, v)$ recursively by computing $F(u, f_u)$ and $F(f_v, v)$, using V_i and V_j, respectively, and then computing $F(f_u, f_v)$, using $\mathrm{DS}(\mathcal{T}_R^*)$. Let $q(T)$ denote the maximum number of arithmetic operations necessary to compute $F(u, v)$ for any pair of vertices (u, v) in T. We have

$$q(T) \leq \max\{q(\mathcal{T}_R^*) + 2, q(T_1), \ldots, q(T_p)\}. \tag{3}$$

Let us now examine the storage $|\mathrm{DS}(T)|$ required by the data structure. Let d_i be the number of vertices in the fringe of T_i. These vertices contribute $\binom{d_i}{2}$ precomputed values to V_i, while any other vertex in T_i contributes d_i values. The storage, $|C|$, required by item C is therefore

$$\leq \sum_{1 \leq i \leq p} \left[(|T_i| + 1 - d_i) d_i + \binom{d_i}{2} \right] \leq \sum_{1 \leq i \leq p} d_i |T_i|.$$

From (1) we derive $|C| \leq k \sum_{1 \leq i \leq p} d_i$. The quantity $\sum_{1 \leq i \leq p} d_i$ represents the total number of fringe vertices in all the T_i's. To bound this quantity above is not

difficult. Starting from any vertex of T, follow any path with this rule in mind: when entering T_i, aim toward another fringe-vertex of T_i, if any, and continue the traversal in the other subtree. Since T is a tree, this process must stop at some point, and thus identify a tree T_i with a single fringe-vertex. The contribution of T_i to $\sum_{1 \leq i \leq p} d_i$ is at most 2, therefore, by induction $\sum_{1 \leq i \leq p} d_i \leq 2(p-1)$. It follows that $|C| < 2pk$, which from (2) leads to $|C| < 6|T|$. We conclude that

$$|\mathrm{DS}(T)| < \sum_{1 \leq i \leq p} |\mathrm{DS}(T_i)| + |\mathrm{DS}(\mathcal{T}_R^*)| + 6|T|. \tag{4}$$

Let us relate the size of the tree $\mathcal{T}_R^*$ to p, the cardinality of $\mathcal{T}$. The contribution of T_i to the number of edges of $\mathcal{T}_R^*$ is easily seen to be dominated by $2d_i$ (this bound is not tight, but chosen for convenience). We derive $|\mathcal{T}_R^*| \leq \sum_{1 \leq i \leq p} (2d_i)$, hence

$$|\mathcal{T}_R^*| < 4p. \tag{5}$$

Keeping the quantity k as a parameter allows us to establish a class of optimal space-time trade-offs. Instead of looking for algorithms with some given performance, we reverse our perspective and set out to characterize the size of trees within which a scheme with a prespecified complexity can always be found. We introduce the function $D(t, \alpha)$, defined for all integers $t \geq 0$, $\alpha \geq 0$ as follows:

1. $D(t, 0) = D(0, \alpha) = 0$;
2. $D(t, \alpha) = \max\{n \mid \forall \text{ free tree } T \text{ s.t. } |T| \leq n, \exists (t, \alpha)\text{-scheme for } T\}$ for $t > 0$, $\alpha > 0$.

To establish a recurrence relation on $D(t, \alpha)$, we must introduce another function, $p(t, \alpha)$ related to the maximum cardinality of interconnection patterns yielding a prespecified performance. Let $p(t, \alpha)$ be the maximum value of p such that for any p' $(1 \leq p' \leq p)$, any $\mathcal{T} = \{T_1, \ldots, T_{p'}\}$ with

$$[\forall i \, (1 \leq i \leq p'), \, D(t, \alpha)/3 < |T_i| \leq D(t, \alpha)]$$

and any $R \in C(\mathcal{T})$, there exists a $(t, \alpha + 18)$-scheme for $\mathcal{T}_R$. The growth of $D(t, \alpha)$ follows two key inequalities.

Lemma 4. *For any $t \geq 1$ and any $\alpha \geq 1$, we have $D(t, \alpha + 18) \geq (p(t, \alpha)/3) D(t, \alpha)$.*

Proof. It suffices to show that there exists a $(t, \alpha + 18)$-scheme for any tree T with at most $(p(t, \alpha)/3) D(t, \alpha)$ edges. We can assume that $|T| > D(t, \alpha)$, otherwise the result is obvious. This allows us to follow the lines of Lemma 3, and partition T into p subtrees $T_1, \ldots, T_p$ such that, for each i $(1 \leq i \leq p)$, we have $D(t, \alpha)/3 < |T_i| \leq D(t, \alpha)$. Note that this is always possible since for any $t \geq 1$ and $\alpha \geq 1$, we have $D(t, \alpha) \geq 1$. Since $|T| \leq \frac{1}{3} D(t, \alpha) p(t, \alpha)$, we have from relation (2) $p < 3|T|/D(t, \alpha) \leq p(t, \alpha)$, at which point the proof follows directly from the definition of $p(t, \alpha)$. □

Lemma 5. *For any $t \geq 2$ and any $\alpha \geq 1$, we have $p(t, \alpha) \geq \frac{1}{4} D(t-2, D(t, \alpha))$.*

Proof. The lemma is clearly true for $t = 2$, so let us assume that $t > 2$. Let $\mathcal{T} = \{T_1, \ldots, T_p\}$ be a set of free trees such that $p \leq \frac{1}{4} D(t-2, D(t, \alpha))$. Assume furthermore that for each i $(1 \le i \le p)$ we have $D(t, \alpha)/3 < |T_i| \le D(t, \alpha)$. Let R be an arbitrary interconnection pattern of $C(\mathcal{T})$; we shall show that $\mathcal{T}_R$ always admits of a $(t, \alpha+18)$-scheme. We follow the divide-and-conquer strategy developed previously. This involves setting up the following data structures:

1. $A = \{\mathrm{DS}(T_1), \ldots, \mathrm{DS}(T_p)\}$. For each T_i, we set up a (t, α)-scheme, which is certainly possible since $|T_i| \le D(t, \alpha)$.
2. $B = \mathrm{DS}(\mathcal{T}_R^*)$. From (5) we know that $\mathcal{T}_R^*$ has no more than $4p$ edges. Since $p \le \frac{1}{4}D(t-2, D(t, \alpha))$ and $t > 2$, we have $|\mathcal{T}_R^*| < D(t-2, D(t, \alpha))$, therefore there exists a $(t-2, D(t, \alpha))$-scheme for $\mathcal{T}_R^*$.
3. $C = \{V_1, \ldots, V_p\}$. Define as previously.

From (3) it follows directly that the scheme proposed involves at most t operands for any evaluation of F over $\mathcal{T}_R$. Let S represent the amount of storage required by the algorithm. From (4) we have

$$S \le \sum_{1 \le i \le p} \alpha |T_i| + 4pD(t, \alpha) + 6|T|.$$

From the hypothesis that $D(t, \alpha)/3 < |T_i|$, we derive $\sum_{1 \le i \le p} |T_i| > (D(t, \alpha)/3)p$, hence $p < 3|T|/D(t, \alpha)$. This implies that $4pD(t, \alpha) < 12|T|$, which yields $S < \alpha|T| + 12|T| + 6|T| = (\alpha+18)|T|$. We have thus shown that $\mathcal{T}_R$ admits of a $(t, \alpha+18)$-scheme, which completes the proof. □

LEMMA 6. *For any* $t \ge 2$ *and any* $\alpha \ge 1$, *we have* $D(t, \alpha+18) \ge \frac{1}{12}D(t, \alpha)D(t-2, D(t, \alpha))$.

PROOF. A direct consequence of Lemmas 4 and 5. □

For convenience, we introduce the function $B(i, j)$, defined as follows: for all $i, j \ge 0$, we have $B(i, j) = \lfloor \frac{1}{18}D(2i+2, 18j) \rfloor$. We can assess the (fast) growth of $B(i, j)$ by means of a recurrence relation.

LEMMA 7.

1. $\forall j \ge 0$, $B(0, j) \ge 2j$.
2. $\forall i \ge 1$, $B(i, 0) = 0$ *and* $B(i, 1) \ge 2$.
3. $\forall i \ge 1$, $\forall j \ge 2$, $B(i, j) \ge B(i, j-1)B(i-1, B(i, j-1))$.

PROOF. $B(0, j) = \lfloor \frac{1}{18}D(2, 18j) \rfloor$. Assume that $j > 0$; a simple solution for achieving time ≤ 2 is to remove from T one edge adjacent to a leaf, and store the value of F over each edge of T and over the path between each pair of edges in the reduced tree. The space used will be

$$\binom{|T|-1}{2} + |T|,$$

so there is a $(2, 18j)$-scheme for all T such that

$$\binom{|T|-1}{2} + |T| \le 18j|T|,$$

i.e. $|T| \le 36j + 1 - 2/|T|$, from which we easily derive (1). We omit the proof that the function $B(i, j)$ is monotone (nondecreasing with respect to i and j), therefore for all $i \ge 1$ we have $B(i, 1) \ge B(0, 1) \ge 2$. Also, from Lemma 6, we derive

$$B(i, j) \ge \lfloor \tfrac{18}{12} B(i, j-1)B(i-1, B(i, j-1)) \rfloor,$$

for $i \ge 1$ and $j \ge 2$, from which (3) follows, since

$$B(i, j-1)B(i-1, B(i, j-1)) \ge B(0, j-1)B(0, B(0, j-1)) \ge 8(j-1)^2 > 2. \quad \square$$

We shall now relate the function $B(i, j)$ to Ackermann's function. Beforehand, we recall well-known results; some of them have trivial proofs, which are therefore omitted. For a thorough treatment of the numerical properties of Ackermann's function see Tarjan [14]. Let $A(i, j)$ be the function defined recursively as follows:

$$A(0, j) = 2j \quad \text{for any} \quad j \ge 0.$$
$$A(i, 0) = 0 \quad \text{and} \quad A(i, 1) = 2 \quad \text{for any} \quad i \ge 1.$$
$$A(i, j) = A(i-1, A(i, j-1)) \quad \text{for any} \quad i \ge 1 \quad \text{and} \quad j \ge 2.$$

For any $m \ge n \ge 1$, we define the function $\alpha(m, n)$ by $\alpha(m, n) = \min\{i \mid i \ge 1, A(i, 4\lceil m/n \rceil) > \log_2 n\}$, and, for any real number x and integer $j \ge 2$, we define $a(x, j)$ by

(6) $$a(x, j) = \min\{i \mid i \ge 1, A(i, j) > x\}.$$

LEMMA 8. *The function* $A(i, j)$ *is monotone.*

LEMMA 9. *For any* j, j' *with* $3 \le j \le j' \le 300j$, *we have* $a(x, j) \le a(x, j') + 4$.

PROOF. Equation (11) in Tarjan [14] states that for any $i \ge 0$, $j \ge 1$, we have $A(i+1, j+1) \ge A(i, 2^j)$. Applying this inequality iteratively, we complete the proof by observing that for any $i \ge 4$, $j \ge 3$, we have

$$A(i, j) \ge A(i-1, 2^{j-1}) \ge A(i-4, 2^{2^{2^{j-1}-1}-1}-1}) \ge \cdots \ge A(i-4, 300j). \quad \square$$

Lemma 10. *For any x, j with $x \geq 1$ and $j \geq 4$, we have $a(x, j) = O(a(\log_2 x, j))$.*

Lemma 11. *For any m, n with $m \geq n \geq 1$, we have $\alpha(m, n) = a(\log_2 n, 4\lceil m/n \rceil)$.*

Lemma 12. *For any $t \geq 1$, we have $\alpha(m, n) \leq \alpha(tm, tn)$.*

Lemma 13. *For any $i, j \geq 0$, $A(i, j) \leq B(i, j)$.*

Proof. Straightforward double induction. □

We are now ready to evaluate the performance of our algorithm. Let $f(m, n)$ denote the minimum value of t such that for any tree T with $|T| = n$, there exists an algorithm which uses at most m units of storage and computes any value of F over T in at most t steps. Note that if $m < n$, then $f(m, n) = \infty$. We distinguish between two cases.

Case 1: $m \geq 72n > 1$. Clearly, $f(m, n) \leq \min\{t \mid n \leq D(t, \lfloor m/n \rfloor)\}$. Since D is monotone, we also have

$$f(m, n) \leq \min\left\{2i+2 \,\middle|\, 18n \leq D\left(2i+2, 18\left\lfloor \frac{\lfloor m/n \rfloor}{18} \right\rfloor\right)\right\}.$$

Now, because $18\lfloor m/18n \rfloor \leq \lfloor m/n \rfloor$, we have $\lfloor m/18n \rfloor \leq \lfloor \lfloor m/n \rfloor / 18 \rfloor$, which implies

$$f(m, n) \leq \min\left\{2i+2 \,\middle|\, 18n \leq D\left(2i+2, 18\left\lfloor \frac{m}{18n} \right\rfloor\right)\right\}.$$

We derive

$$f(m, n) \leq \min\left\{2i+2 \,\middle|\, n \leq B\left(i, \left\lfloor \frac{m}{18n} \right\rfloor\right)\right\}. \tag{7}$$

Let $g(m, n)$ be the function defined by $g(m, n) = \min\{i \mid n \leq A(i, \lfloor m/18n \rfloor)\}$. From (6) and the fact that A is monotone we find that

$$g(m, n) \leq a\left(n, \left\lfloor \frac{m}{18n} \right\rfloor\right). \tag{8}$$

Next we apply Lemma 9, with $j = \lfloor m/18n \rfloor$ and $j' = 4\lceil m/n \rceil$. We can easily check that since $m \geq 72n$, we have $3 \leq j \leq j' \leq 300j$, which directly implies that $g(m, n) \leq a(n, 4\lceil m/n \rceil) + 4$, and, from Lemmas 10 and 11, we derive $g(m, n) = O(a(\log_2 n, 4\lceil m/n \rceil)) = O(\alpha(m, n))$. Lemma 13 and relation (7) show that $f(m, n) \leq 2 + 2g(m, n)$, from which we conclude that for any m, n ($m \geq 72n > 1$) we have $f(m, n) = O(\alpha(m, n))$.

Case 2: $n \leq m < 72n$. The idea is to break up the tree T into p subtrees $\mathcal{T} = \{T_1, \ldots, T_p\}$, applying the results of Lemma 3, and computing the function F in the usual way. Recall that in the general case the path over which F is to be evaluated is decomposed into three paths, one in $\mathcal{T}_R^*$, and the two others in some of the subtrees of $\mathcal{T}$. The difference with the previous method is that the procedure will now recurse *only* with respect to the path in $\mathcal{T}_R^*$, the two other paths being handled naively by examining all of their edges in turn. We use n words of storage to store the data structures associated with all the trees of $\mathcal{T}$ and use the remaining $m - n$ words of storage to apply the technique of the first case to $\mathcal{T}_R^*$. We wish to prove that in all cases we can achieve a space–time trade-off of the type

$$f(m, n) = O\left(\alpha(m, n) + \frac{n}{m - n + 1}\right). \tag{9}$$

In order to apply the previous scheme to $\mathcal{T}_R^*$, we must assume that the available storage for $\mathcal{T}_R^*$ is at least 72 times the number of edges in $\mathcal{T}_R^*$. For this reason, we shall assume in the following that we have $m > n + 72$. Note that if this inequality is not satisfied, relation (9) holds trivially. The next result expresses the relationship between k, the maximum size of each T_i, and the storage available for $\mathcal{T}_R^*$ (see Lemma 3).

Lemma 14. *If $k = \lceil 864n/(m-n) \rceil$, the decomposition of Lemma 3 leads to $|\mathcal{T}_R^*| < \frac{1}{72}(m - n)$.*

Proof. We may assume that $k = \lceil 864n/(m-n) \rceil \leq n$, since otherwise relation (9) is trivially satisfied. This assumption allows us to apply (2, 5). From these relations we derive $|\mathcal{T}_R^*| < 4p < 12(n/k)$, hence $|\mathcal{T}_R^*| < 12n((m-n)/864n) = \frac{1}{72}(m - n)$. □

We are now back to Case 1 with respect to $\mathcal{T}_R^*$. We use n words of storage for all the trees $T_1, \ldots, T_p$ and $m - n$ words for the tree $\mathcal{T}_R^*$. From Lemma 14 we derive an upper bound on $f(m, n)$, the maximum number of precomputed operands needed in any given evaluation of F:

$$f(m, n) \leq O(\alpha(72|\mathcal{T}_R^*|, |\mathcal{T}_R^*|)) + 2 \max_{1 \leq i \leq p} |T_i|.$$

Using relation (1), Lemma 12, $|\mathcal{T}_R^*| \leq n$, and $m < 72n$, we conclude that

$$f(m, n) \leq 2k + O(\alpha(72n, n)) = \left\lceil \frac{1728n}{m - n} \right\rceil + O(\alpha(m, n)).$$

Being now in the case $m > n + 72$, this establishes relation (9).

In all cases we have shown that $f(m, n) = O(\alpha(m, n) + n/(m - n + 1))$, from which Theorem 1 follows. Yao [18] showed that under reasonable assumptions this result was optimal if restricted to linear lists. Theorem 1 is therefore optimal.

4.2. A RAM Implementation. How difficult is it to implement the solution of Theorem 1 on a RAM? We begin with a technical result.

LEMMA 15. *It is possible to preprocess a free tree T in linear time and space, so that for any two vertices v and w of T the second and next-to-last vertices (if any) on the path from v to w can be found in constant time.*

PROOF. For conceptual simplicity we build two trees T' and T'' as follows. Pick any vertex of T to make T' into a rooted version of T. With each node of T' we keep a pointer to its parent (if any) and its preorder and postorder ranks (to check whether a node is a descendant of another in constant time). To build T'' we relink each internal node z of T' with its children $z_1, \ldots, z_k$: the edges $(z, z_2), \ldots, (z, z_k)$ are replaced by $(z_1, z_2), \ldots, (z_{k-1}, z_k)$. We call z_k the *link-node* of z. The root of T'' is the same as that of T'. Finally, we use the linear-time preprocessing of Harel and Tarjan [7] to allow the constant-time computation of nearest common ancestors in T''. The query-answering proceeds as follows. Check whether v is a descendant of w or vice-versa. If not, then the pointers of T' from v and w to their parents will give us the desired answer. If on the contrary, say, v is a descendant of w, it then suffices to find the unique child w' of w which is an ancestor of v. To do so, we observe that w' is precisely the nearest common ancestor of v and the link-node of w in T''. □

(A) The Preprocessing. We shall proceed in a bottom-up fashion and describe the implementation of the inner loop of the algorithm first. What is the complexity of implementing the nonrecursive part of DS(T)?

Recall that T is broken up into p subtrees $T_1, \ldots, T_p$, via Lemma 3, and that $\mathcal{T}_R^*$ and V_i $(1 \le i \le p)$ must be computed. From Lemma 3 the T_i's can be found in linear time. For each i $(1 \le i \le p)$ we compute V_i in time proportional to its size by a depth-first traversal of T_i, starting from each vertex on the fringe. We store V_i as a rectangular matrix with, say, columns representing fringe-vertices. To facilitate its use we keep a correspondence table which indicates the row- (resp. column-) index associated with a given vertex (resp. fringe-vertex) of T_i. Next, we build $\mathcal{T}_R^*$ by marking the fringe-vertices in each T_i, made into rooted trees for the occasion, and finding their nearest common ancestor (details omitted). Completing the computation, which also includes removing vertices of degree 2 and updating edge-weights, is routine. We also need a tree $\mathcal{T}^+$, defined as follows: for each T_i $(1 \le i \le p)$ create a new vertex z_i, and connect each z_i to the fringe-vertices of T_i: $\mathcal{T}^+$ is the tree formed by the new edges; its vertices are $z_1, \ldots, z_p$ and the fringe-vertices of T. Each edge of $\mathcal{T}^+$ is of the form z_iv, where v is a fringe-vertex of T_i. Next, split each edge z_iv into two edges z_iv' and $v'v$. The vertex v is called the *image* of v'. Finally, apply to $\mathcal{T}^+$ the preprocessing of Lemma 15. Each tree defined above (T, T_i's, $\mathcal{T}_R^*$, $\mathcal{T}^+$, etc.) has its vertices labeled consecutively: the vertices of tree t are labeled $1, 2, \ldots, |t|+1$. This means that a given vertex is likely to have several distinct labels, depending on which tree is considered. For this reason we need a few correspondence tables, so that given the label of a vertex v in tree X we can find its label in tree Y in constant time.

Specifically, we need correspondences from T to $T_1, \ldots, T_p$: given $v \in T$, we need the index i of the tree T_i which contains it (or any of them if there are several), as well as the labels of z_i in $\mathcal{T}^+$ and v in T_i. Also, given the label of a node in $\mathcal{T}^+$ adjacent to some z_i, we need constant-time access to the labels of its image in both T_i and $\mathcal{T}_R^*$. All the operations described above can be performed in linear time. Moreover, with these data structures in hand and the use of Lemma 15 with respect to $\mathcal{T}^+$, we can easily reduce the computation of $F(u, v)$ (assuming that u and v do not fall into the same tree T_i) to that of $F(f_u, f_v)$ in constant time—see the discussion leading to relation (3).

It appears that a fixed fraction of the storage is used for storing semigroup values. For this reason, the quantity m, as used in the following, will refer to this specific amount of storage. It must be understood that it only denotes the actual storage requirement up to within a constant factor.

(B) Implementing a (t, α)-Scheme. For convenience, we introduce $D'(t, \alpha)$ to provide an estimate on $D(t, \alpha)$. (Obviously, the latter should not be computed if preprocessing time is a concern.) For all $t = 2i > 0$ and $\alpha = 18j \ge 0$, we have $D'(2, \alpha) = 2\alpha$, $D'(t, 0) = 0$, and $D'(t, 18) = 36$. For all $t = 2i > 2$ and $\alpha = 18j \ge 18$, we have

$$D'(t, \alpha + 18) = \lfloor \tfrac{1}{12} D'(t, \alpha) D'(t-2, 18\lfloor D'(t, \alpha)/18 \rfloor) \rfloor.$$

Note that these relations form a well-founded definition of $D'(t, \alpha)$ for all $t = 2i > 0$ and $\alpha = 18j \ge 0$; the function is left undefined outside of these values. To begin with, we prove by induction that, as a function of i and j, D' is monotone. The cases $t = 2$ and $\alpha = 0$, 18 being obvious, it suffices to show that for any $t = 2i > 2$ and $\alpha = 18j \ge 18$ the quantity $D'(t, \alpha + 18)$ is greater than or equal to both $D'(t, \alpha)$ and $D'(t-2, \alpha + 18)$. The former inequality is easily established: by induction hypothesis we have $D'(t, \alpha) \ge D'(t, 18) = 36$, therefore

$$D'(t, \alpha + 18) \ge \lfloor \tfrac{1}{12} D'(t, \alpha) D'(2, 36) \rfloor = 6D'(t, \alpha) > D'(t, \alpha).$$

To handle the second inequality, we observe that by induction hypothesis we have $D'(t, \alpha) \ge D'(2, \alpha) = 2\alpha$, therefore $D'(t, \alpha + 18) \ge 3D'(t-2, 18\lfloor \alpha/9 \rfloor)$, which is at least $D'(t-2, \alpha + 18)$ since $\alpha = 18j \ge 18$. Next, we prove by induction that any free tree T with $n \le D'(t, \alpha)$ edges ($t = 2i > 0$ and $\alpha = 18j \ge 0$) admits a (t, α)-scheme which can be implemented (in the sense of (A)) in $O(\alpha n)$ time and with $\le \alpha n$ semigroup values. Following the definition of D', the proof is in four parts:

1. $n \le D'(2, \alpha) = 2\alpha$. See Case 1 in the proof of Lemma 7.
2. $n \le D'(t, 0) = 0$. Trivial.
3. $n \le D'(t, 18) = 36$. From (1) above a $(2, 18)$-scheme can be implemented for $n = D'(2, 18) = 36$.
4. $t = 2i > 2$ and $\alpha = 18j \ge 18$, with $n \le D'(t, \alpha + 18)$. We can immediately assume that $n > D'(t, \alpha)$, and therefore apply to T the preprocessing of (A) for

$k = D'(t, \alpha)$. From (2) we derive $p < 3D'(t, \alpha+18)/D'(t, \alpha)$, which is at most $\frac{1}{4}D'(t-2, 18\lfloor D'(t, \alpha)/18 \rfloor)$. At this point the proof of Lemma 5 can be used verbatim (substituting D' for D and $D'(t-2, 18\lfloor D'(t, \alpha)/18\rfloor)$ for $D(t-2, D(t, \alpha))$).

We can now put all these results together and derive an algorithm for the general case.

(C) The General Algorithm. *Let* T be a free tree of n edges and let m be the maximum number of semigroup values which we are allowed to precompute. Given the necessary overhead in storage we may assume that m/n is larger than a constant, say, 72. Let $j(i)$ be the largest j such that $D'(2i, 18j) \le n$, and let Δ_i be the sequence $D'(2i, 0)$, $D'(2i, 18), \ldots, D'(2i, 18j(i))$. We begin by computing the sequence $\Delta_1, \Delta_2, \ldots, \Delta_\tau$, where τ is the smallest i such that $18(j(i)+1) \le m/n$. Note that since $D'(t, 36)$ is strictly increasing with t and $m/n > 72$, the quantity τ is always well defined. Also, it is important to see that each value in any sequence Δ_i can be derived from values previously computed: this comes from the fact that while computing $D'(t, \alpha+18) \le n$ we have

$$D'(t-2, 18\lfloor D'(t, \alpha)/18\rfloor) \le D'(t, \alpha+18) \le n.$$

We are now ready to start the preprocessing proper, that is, the construction of a $(2\tau, 18j(\tau)+18)$-scheme for T, as discussed in (A) and (B). Next, we must show that the preprocessing time is proportional to the storage needed. The only difficulty is that we do not know what scheme to use for $\mathcal{T}_R^*$. Let j^* be the largest j such that $D'(2\tau-2, 18j) < |\mathcal{T}_R^*|$ (computed by trying out $j = 1, 2, \ldots$). Since $|\mathcal{T}_R^*| \le n$ this value of D' has been precomputed; we can then proceed recursively by constructing a $(2\tau-2, 18j^*+18)$-scheme for $\mathcal{T}_R^*$. Since $D(t, 18j)$ is strictly increasing in j, computing j^* adds another linear term to the preprocessing time, which thus still remains proportional to the storage (Lemma 5). More simply, because of (1) and the fact that each row increases at least geometrically, finding the proper entry in row 2τ for each T_i can be done in constant time by scanning the row from left to right from the initial position.

The data structure can be regarded as a tree $\mathcal{D}$ (of high degree), each of whose nodes points to the root of another (distinct) tree of a similar nature. Each node of $\mathcal{D}$ is associated with a certain subset of vertices in T (of size bounded by a constant in the case of a leaf). Given a query path from v_1 to v_2, let l_1 and l_2 be the two leaves of $\mathcal{D}$ associated with v_1 and v_2, respectively. The first task must be to determine the nearest common ancestor of l_1 and l_2 in $O(1)$ time. (This is necessary to ensure relation (3).) Once again, we turn to the linear algorithm in Harel and Tarjan [7] for constant-time computations. In this manner our earlier remark about storage can be made now with respect to time. The actual query time will be at most proportional to the number of semigroup operations needed, in other words, proportional to $\tau = f'(m, n)$. To show that $f'(m, n)$ can be estimated just as its counterpart $f(m, n)$, it suffices to show that the function $B'(i, j) = \lfloor \frac{1}{18}D'(2i+2, 18j)\rfloor$, defined for all $i, j \ge 0$, satisfies all the conditions of Lemma 7. Since D' is monotone, so is B'. The first two conditions are easily checked. Regarding the last one, observe that $B'(i, j)$ is equal to

$$\lfloor \tfrac{1}{18} \lfloor \tfrac{1}{12} D'(2i+2, 18j-18) D'(2i, 18\lfloor D'(2i+2, 18j-18)/18\rfloor)\rfloor\rfloor,$$

which is larger than or equal to $\lfloor \frac{3}{2}B'(i, j-1)B'(i-1, B'(i, j-1))\rfloor$. Since B' is monotone, we have

$$B'(i, j-1)B'(i-1, B'(i, j-1)) \ge B'(0, j-1)B'(0, B'(0, j-1)) \ge 8(j-1)^2 \ge 8,$$

therefore

$$B'(i, j) > B'(i, j-1)B'(i-1, B'(i, j-1)).$$

This proves our claim and shows that for all $i, j \ge 0$, $A(i, j) \le B'(i, j)$. We conclude that the query time is $O(\alpha(m, n))$. Let $c > 1$ be the constant by which m must be multiplied to obtain the actual storage (or time) requirement. If we assume that m/n is larger than some appropriate constant, the proof of Lemma 9 can be easily adapted to show that $\alpha(m, n) = O(\alpha(cm, n))$. The time needed to build the data structure is proportional to m and the sum $|\Delta_1| + \cdots + |\Delta_\tau|$, which is easily shown to be $O(m + n + \alpha(m, n))$, which is also $O(m)$.

Theorem 2. *Let T be a free tree with n weighted edges. There exists a constant $c > 1$ such that, for any integer $m > cn$, it is possible to sum up weights along an arbitrary query path of T in time $O(\alpha(m, n))$. The data structure is of size at most m and can be constructed in time $O(m)$.*

4.3. *A Theoretical Application.*

Let G be a connected graph with m weighted edges and n vertices. Tarjan [15] has shown that verifying if a given subgraph of G forms a minimum spanning tree (MST) can be done in $O(m\alpha(m, n))$ time. We can rederive this result from Theorem 2. Interestingly, our method is completely different from Tarjan's.

Let T be a spanning tree of G. Recall that T is an MST if and only if the weight of every edge e in $G\backslash T$ is not smaller than that of any edge e along the path of T between the end-vertices of e. Since $m \ge n-1$ there exist a constant $d > 1$ and an integer $m' = dm$ such that $m' > cn$, where c is the constant of Theorem 2. Setting the amount of storage in Theorem 2 to be equal to m', one can check each edge of G in $O(\alpha(m', n))$ time, at a preprocessing cost of O(m). Thus, the verification is complete after $O(m\alpha(m', n)) = O(m\alpha(m, n))$ operations. One will observe that the method of Section 3 (Cartesian trees) would allow us to perform all the checks in $O(m)$ time; unfortunately, as is easily seen, the preprocessing might require $\Omega(n \log n)$ time.

Before closing this section, let us recall that Komlós [9] has shown that $O(m)$ comparisons suffice to decide whether a spanning tree is minimum. To implement an algorithm on a RAM which performs within the same time bound is still, to our knowledge, an open problem.

5. Generalizing Range Trees: The Emulation Dag of a Free Tree.

A balanced tree is a versatile tool to represent a linear list. In this section we develop the equivalent notion for a free tree, which we call an *emulation dag*. To motivate the discussion we look at a specific problem: let T be a free tree supplied with a real edge-weight function w; given a query consisting of a path P and a real interval $[x, y]$, compute $|\{e \in P | x \leq w(e) \leq y\}|$. This problem is a generalization of orthogonal range counting in $\mathfrak{R}^2$. We develop our solution in three stages: first, we describe the emulation dag of T and the canonical decomposition of a path; then we generalize the notion of a range tree to provide a near-optimal solution; finally, we use a compaction scheme to optimize the data structure. In light of the following discussion it will be clear that the same ideas can be used to solve the "free tree" generalizations of many other orthogonal range searching problems [2], [3], [17].

5.1. *The Emulation Dag.*

We encode T by means of a directed acyclic graph $\mathcal{T}$, called the *emulation dag* of T. We start the construction of $\mathcal{T}$ by calling the procedure of Lemma 2 recursively and putting this process in correspondence with the nodes of a binary tree. For this reason, we use binary tree terminology to refer to the nodes of $\mathcal{T}$. Note that since centroids are not always unique, the emulation dag may often not be uniquely defined. To avoid confusion between T and its emulation dag, we will refer to the *vertices* of T and the *nodes* of $\mathcal{T}$.

Let T_1 and T_2 be the two subtrees of Lemma 2. Associate the root of $\mathcal{T}$ with T, and its left (resp. right) child with T_1 (resp. T_2). In the general case, each node v of $\mathcal{T}$ is associated with a tree $T(v)$, which is a subgraph of T. Recursive application of Lemma 2 completes the definition of $\mathcal{T}$, with the understanding that every leaf l of $\mathcal{T}$ has its associated tree, $T(l)$, made up of a single edge. Note that each node v of $\mathcal{T}$ thus corresponds to a unique *partitioning* vertex of T, denoted $\sigma(v)$ (σ is in general many-to-one). If v_1 and v_2 are the two children of an internal node v of $\mathcal{T}$, then $\sigma(v)$ is the unique vertex common to $T(v_1)$ and $T(v_2)$. If v is a leaf of $\mathcal{T}$, let uu' denote the unique edge of $T(v)$ and let w be the parent of v. Observing that $\sigma(w)$ is a vertex of uu', say u, we define $\sigma(v)$ to be u'. Each edge of $\mathcal{T}$ is directed from parent to child. It is easily seen that $\mathcal{T}$ has height at most $\lceil \log_{1.5}|T| \rceil$. Ignoring the dashed edges, Figure 3(b) illustrates the construction of $\mathcal{T}$. Note that four nodes of $\mathcal{T}$ are mapped to the same vertex of T, labeled 2. This is because this vertex was picked as the centroid on three occasions, and is also in correspondence with a leaf of $\mathcal{T}$.

In order to show that $\mathcal{T}$ induces a natural (so-called *canonical*) decomposition of any path in T, we must first augment $\mathcal{T}$ with additional edges. Why is that necessary? Certainly, a path of T such as (7, 11) in Figure 3(a) can be expressed in $\mathcal{T}$ by the four edges connecting the root to the leaf labeled 11. But what about

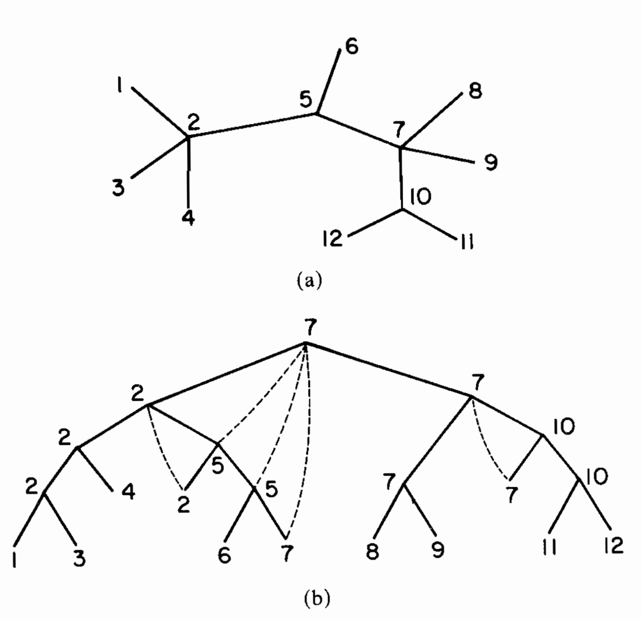


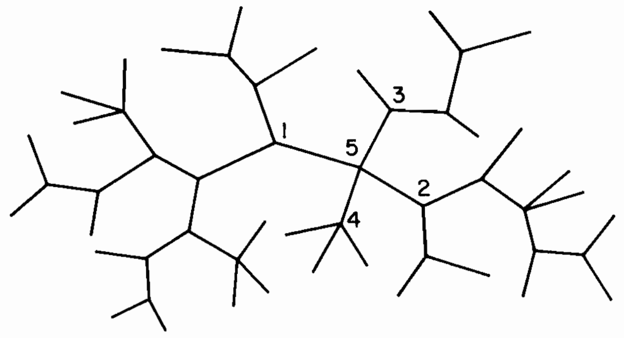
Fig. 3. A free tree and its emulation dag.

the path (7, 6)? We have no choice but to take the left edge from the root and then hope for the best. Unfortunately, this edge corresponds to a path of T, (7, 2), that is not a subpath of (7, 6). We need *shortcuts* in T to remedy this situation. The idea is to link any node v to any descendant whose associated subtree contains $\sigma(v)$. For each internal node $v \in \mathcal{T}$ in turn, apply the following procedure. Let l_1 (resp. r_1) be the left (resp. right) child of v. We describe the procedure with respect to l_1 only, with the understanding that it must also be applied to r_1.

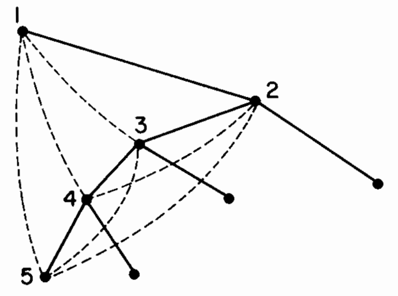
Of the children of l_1, at least one of them, call it l_2, is such that $T(l_2)$ contains $\sigma(v)$. If this property is satisfied by both children (hence $\sigma(v) = \sigma(l_1)$) or if l_1 has no children, then no additional link is provided. Otherwise, we provide a direct edge from v to l_2 and iterate on this process by considering the two children of l_2 (if any). Specifically, let $l_1, \ldots, l_k$ be the sequence of nodes of $\mathcal{T}$ defined as follows (initially, $i = 1$):

1. If l_i is a leaf or $\sigma(l_i) = \sigma(v)$, then set $k = i$ and stop.
2. Else, let l_{i+1} be the unique child of l_i such that $\sigma(v)$ is a vertex of $T(l_{i+1})$; set i to $i+1$ and go to 1.

Next, add to $\mathcal{T}$ the directed edges $(v, l_2), \ldots, (v, l_k)$. The augmentation edges of $\mathcal{T}$ are shown as dotted lines in Figure 3(b). Note that (v, l_1) is already in $\mathcal{T}$. Let $L_1(v) = \{(v, l_1), \ldots, (v, l_k)\}$, and $L_2(v)$ be the sequence defined similarly with respect to v and r_1, the right child of v. These two sequences form the adjacency-list associated with v in the representation of $\mathcal{T}$. For any pair $u, v \in \mathcal{T}$, let $P(u, v)$ designate the path of T between $\sigma(u)$ and $\sigma(v)$. Each edge $(v, w) \in \mathcal{T}$ is in correspondence with the *canonical* path $P(v, w)$: by construction, this path lies in $T(w)$ and is possibly reduced to a single vertex. Incidentally, it is interesting



(a)



(b)

Fig. 4. A case of nonplanarity.

to observe that the emulation dag is not necessarily planar (nodes 1–5 form a K_5 in Figure 4(b)).

LEMMA 16. *Let T be a free tree of n edges. An emulation dag of T has $O(n)$ edges and each of its nodes has indegree and outdegree $O(\log n)$.*

PROOF. The outdegree of each node cannot exceed twice its height. An upper bound on $|\mathcal{T}|$ is thus given by the recurrence relation $U(n) = U(n_1) + U(n_2) + O(\log n)$, with $U(1) = O(1)$, $n_1 + n_2 = n$, and $1 \le n_1, n_2 \le \frac{2}{3}n$, which gives $U(n) = O(n)$. That the indegree and outdegree of each node are $O(\log n)$ follows directly from the $O(\log n)$ height of $\mathcal{T}$. □

LEMMA 17. *Any path of T can be partitioned into $O(\log |T|)$ canonical paths. The decomposition can be computed in $O(\log |T|)$ time, after preprocessing.*

PROOF. Let P denote the path to be decomposed. One crucial primitive in the decomposition procedure is the ability to check in constant time whether P *stretches* over both subtrees of a node v of $\mathcal{T}$, that is, whether each subtree contains at least one node which σ maps into a vertex of P. Lemma 1 provides

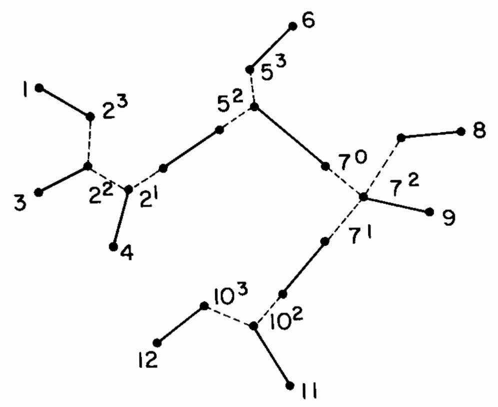


Fig. 5. A tree in normal form.

almost the answer to this question. We say *almost* because the lemma recognizes edges and not vertices as queries. To overcome this difficulty, we express T in *normal form* by creating a dummy edge for each partitioning vertex of T. This is accomplished by examining each node of $\mathcal{T}$ level by level in a top-down fashion. Let v be an internal node of $\mathcal{T}$ with children v_1, v_2. Replace $\sigma(v)$ in T by a dummy edge $\delta(v) = (\sigma'(v), \sigma''(v))$: $T(v_1)$ is now adjacent to $\sigma'(v)$ and $T(v_2)$ to $\sigma''(v)$. If v is not the root, then $\delta(v)$ might be adjacent to other edges which are neither in $T(v_1)$ nor in $T(v_2)$: in this case these edges are now adjacent to either $\sigma'(v)$ or $\sigma''(v)$ indifferently. For consistency, if v is a leaf of $\mathcal{T}$, we define $\delta(v)$ as the edge of T' corresponding to $(\sigma(v), \sigma(w))$ in T, with w the parent of v. Figure 5 represents the tree of Figure 3(a) in normal form: dummy edges have been labeled with superscripts indicating their level in $\mathcal{T}$. Each vertex of T is now associated with one or several vertices of T' forming a connected subgraph of T'. Note that from the definition itself the normalization is in general not unique: the dummy edges associated with a given vertex of T form a subtree that is not defined uniquely.

Let T' be the tree T in normal form. The analogue of $T(v)$ in T' is denoted $T'(v)$. A path P of T can be mapped into a path P' of T' in constant time. Since the mapping is not one-to-one, arbitrary conventions can be made to make the conversion deterministic. The ambiguity comes from the end-vertices of P. The path (1, 11) in Figure 3(a) causes no difficulty, but the path (2, 11) can be mapped into a path that contains any of the edges, 2^1, 2^2, 2^3, indifferently. Any such path will do. With T in normal form, we are now ready to describe the decomposition of P. Initially v is the root of $\mathcal{T}$ and V is the empty list. At the outset of the computation, V will contain each edge of $\mathcal{T}$ in the canonical decompostion of P. We assume that P contains at least one edge.

1. If v is a leaf, let w be its parent and $V = \{(v, w)\}$, then stop. Otherwise, let v_1, v_2 be the two children of v. If P' lies completely in $T'(v_1)$ and does not contain $\delta(v)$, then set $v = v_1$ and iterate recursively through step 1. Otherwise, check whether P' lies completely in $T'(v_2)$ without containing $\delta(v)$. If this is

the case, then set $v = v_2$ and iterate recursively through step 1. Else set $c = d = 1$ and proceed to step 2.

2. Let v_1 (resp. v_2) be the left (resp. right) child of v, and let p_1 (resp. p_2) be the end-vertex of P' in $T'(v_1)$ (resp. $T'(v_2)$).
3. From the list $L_d(v) = \{(v, w_1), \ldots, (v, w_k)\}$, compute j the smallest index i such that $\sigma(w_i) = \sigma(v)$, or the path of T' between p_c and $\sigma'(v)$ contains the edge $\delta(w_i)$, or $\sigma(w_i)$ is the end-vertex of P corresponding to p_c. If none is found, then go to step 4. Otherwise, add the edge (v, w_j) to the list V, provided that $\sigma(w_j) \neq \sigma(v)$. Proceed to step 4 if p_c corresponds to $\sigma(w_j)$. Otherwise, set $v = w_j$ and let v_1 (resp. v_2) be the left (resp. right) child of v. If p_c is a vertex of $T'(v_1)$, then set $d = 1$, else set $d = 2$. Iterate recursively through step 3.
4. Stop if already passed through step 4. If not, reset v to its value at step 2, then set $c = d = 2$ and go to step 3.

With T in normal form, the labeling technique of Lemma 1 applied to P' allows us to execute step 1 in constant time. It suffices to store the relevant labels with the nodes of $\mathcal{T}$. In steps 2 and 3 we can check whether the edge $\delta(w_i)$ lies between p_c and $\sigma'(v)$ in constant time. Checking whether p_c is a vertex of $T'(v_1)$ is also easily done in constant time, again using the technique of Lemma 1. Each list $L_d(v)$ is scanned from the beginning until the index j is found. If none is found, the whole list will have been examined but then we will either stop or go to step 4. In this way, at most two nodes of $\mathcal{T}$ per level are visited during the computation, and each visit takes constant time. This leads to the claimed $O(\log|T|)$ running time. □

The reader familiar with segment trees will recognize similar features in the emulation dag. At the end of step 2, node v is the highest-level node of $\mathcal{T}$ such that P' can be rewritten as the concatenation of $P_1 = P' \cap T'(v_1)$, $\delta(v)$, and $P_2 = P' \cap T'(v_2)$. Paths P_1 and P_2 either are null or lie on distinct sides of the dummy edge $\delta(v)$. Step 3 is applied successively to P_1 and P_2. Its role is iteratively to identify an edge of $\mathcal{T}$ whose associated path in T is a subpath of P. By construction, except at the end, one of the edges adjacent to v always satisfies this property. The edge (v, w_j) with highest-level node w_j is chosen, since it then allows the recursion to proceed with respect to one of its subtrees. Which one is determined by the end-vertex of P' under consideration. Of course, there is no need adding the edge (v, w_j) to the current list V if the corresponding path in T is reduced to a single vertex, that is, $\sigma(w_j) = \sigma(v)$. Checking whether $\sigma(w_i) = \sigma(v)$ during the scan of step 3 prevents a premature jump to step 4; note that in that case P' might not always pass through $\delta(w_j)$. This is immaterial with regards to T, however. To illustrate this point, consider the path of T' formed by the edges labeled 7^0, 7^2, and 8 in Figure 5. The second iteration through step 3 proceeds "as though" 7^1 was in the path.

To illustrate the previous discussion further, consider the decomposition of the path of T from 6 to 11 in Figure 3(a). Referring now to the labels of the nodes of $\mathcal{T}$ in the figure, we obtain (7, 5)-(5, 6) and (7, 10)-(10, 11). The node v found in step 2 is in this case the root of $\mathcal{T}$. The decomposition can be expressed

as the sequence [(6, 5)-(5, 7)-(7, 10)-(10, 11)]. As a sequence of edges this does not form a path in $\mathcal{T}$ but its image under σ does form the path (6, 11) of T.

5.2. Generalizing Range Trees. We return to our original problem: generalized range counting. Following the idea of Bentley's *range tree* [2], we associate with each edge (u, v) of the emulation dag $\mathcal{T}$ an array $C(u, v)$ containing the edges of $P(u, v)$, sorted by weight. To answer a query (P, x, y), we start with a decomposition of the path P into its canonical parts, using Lemma 17. For each edge (u, v) of the decomposition we search $C(u, v)$ for the keys x and y. Using binary search this gives us the number of weights between x and y in $O(\log n)$ time. Repeating this operation for each edge of the decomposition allows us to compute the desired answer, $|\{e \in P | x \leq w(e) \leq y\}|$, in $O(\log^2 n)$ time. Since the edges of $P(u, v)$ lie in $T(v)$ (recall that v is a descendant of u), the added size of the arrays $C(u, v)$ for fixed u amounts to $O((1 + \frac{2}{3} + (\frac{2}{3})^2 + \cdots)|T(u)|) = O(|T(u)|)$. Therefore the size $S(n)$ of the data structure satisfies a recurrence of the form $S(1) = O(1)$ and $S(n) = S(n_1) + S(n_2) + O(n)$ $(n > 1)$, where $n = n_1 + n_2$ and $1 \leq n_1, n_2 \leq 2n/3$. It follows that $S(n) = O(n \log n)$.

Next, we reduce the storage to $O(n)$ and the query-time to $O(\log n)$ by adapting a compaction technique originating in Chazelle [3]. To begin with, for each node v of $\mathcal{T}$ we define the list $B(v)$ as follows: let $C(v)$ be the list of edges of $T(v)$ sorted by weight. For simplicity, we shall assume that all weights are distinct. If v is either the root of $\mathcal{T}$ or a leaf, then $B(v) = C(v)$. Otherwise, let v_1 (resp. v_2) be the left (resp. right) child of v. Since $C(v_1)$ and $C(v_2)$ partition $C(v)$, we can form $B(v)$ by replacing each entry of $C(v)$ by 0 if it is a member of $C(v_1)$, and 1 if not. The $B(v)$'s are boolean arrays, accessed via the function $z(v, k)$, which gives the number of zeros among the first k bits of $B(v)$. For the time being, we assume that this function can be computed in constant time.

For each edge (u, v) of $\mathcal{T}$, the list $C(u, v)$ being a subset of $C(v)$, we can define the boolean array $B(u, v)$ by replacing each entry of $C(v)$ by 1 if it lies in $C(u, v)$, and 0 otherwise. We define the function $z(u, v, k)$ similarly to $z(v, k)$.

The data structure is almost complete. The missing part will be best motivated by discussing the query-answering algorithm now. With $B(u, v)$ substituted for $C(u, v)$ the problem is now to search for x and y in arrays whose keys are implicitly represented by individual bits. Given a real z, let $r(v, z)$ be the position of the smallest entry in $C(v)$ at least as large as z (or $|C(v)|$ if none); by convention the first entry has position 0. Let $\{(u_1, v_1), \ldots, (u_m, v_m)\}$ be the edges of the decomposition of the query path P. It is easy to see that the entire query-answering process can be completed in $O(\log n)$ steps if, for each i $(1 \leq i \leq m)$, $r(v_i, x)$ and $r(v_i, y)$ can be computed in $O(1)$ time. Indeed, it suffices to sum up the number of 1's between positions $r(v_i, x)$ and $r(v_i, y)$ in $B(u_i, v_i)$, for each $i = 1, \ldots, m$. This can be done with the function $z(u_i, v_i, k)$. Let V be the set of nodes visited during the decomposition of P (Lemma 17); V includes, among others, the root v_0 of $\mathcal{T}$ and each u_i, v_i $(1 \leq i \leq m)$. We compute $r(v_0, x)$ in $O(\log n)$ steps by performing a binary search in $B(v_0) = C(v_0)$. Then we deal with each $v \in V$ as follows. If v is not the root then its parent u is in V, and $k = r(u, x)$ is computed before $r(v, x)$. Then we derive $r(v, x) = z(u, k)$ if v is the left child of u, and

$r(v, x) = r(u, x) - z(u, k)$, otherwise. This allows us to compute $r(v_i, x)$ (and similarly $r(v_i, y)$) for each i $(1 \le i \le m)$ in $O(\log n)$ time.

REMARK. A curious effect of the compaction process is that the array $B(u, v)$ is not accessible from u but only from the lower node v. To allow constant-time access from u is possible but it necessitates too much storage.

To complete the description of the algorithm we must show how to implement the function $z(v, k)$ (the case of $z(u, v, k)$ is similar). Let $\lambda = \lceil \log_2 n \rceil$ be the size of a computer word (or a lower bound on it). We represent all the boolean arrays by filling up λ bits per word. In addition, we keep a companion array $Z(v)$ for each node v of $\mathcal{T}$: its ith word indicates the number of zeros among the first i words of $Z(v)$. We also keep some auxiliary arrays: $A_1[i]$ indicates how many zeros the binary representation of i contains $(1 \le i \le n)$ and $A_2[i]$ gives the value of 2^i $(1 \le i < \lambda)$. With these arrays, it is elementary to show that $z(\,)$ can be computed in constant time. (Note that the array A_2 is used to truncate a word by a given number of bits, via a division.)

To summarize, we have obtained a compact representation of our earlier data structure which allows us to answer any query in $O(\log n)$ time. For each internal node u distinct from the root the added size of all the arrays $B(u, v)$ is $O(|C(u)|/\lambda + \text{depth of } u)$. Consequently, if N is the total number of arrays used in the data structure, the new storage requirement is easily shown to be $O(N + (n \log n)/\lambda)$, which is $O(n)$. To show that the entire data structure can be built in $O(n \log n)$ time is left as an exercise.

THEOREM 3. *Let T be a free tree with n weighted wedges. Given an arbitrary query path it is possible to find in* $O(\log n)$ *time how many of its edges have their weights between two query values. The data structure is linear in size and can be built in time* $O(n \log n)$.

Once again, our main objective was to extend techniques developed for linear lists to the case of free trees. As mentioned earlier, the more specifically tree-oriented data structure of Sleator and Targan [12], [13] can be substituted for the emulation dag (applying to it the same compaction technique). This might be the solution of choice if dynamic operations such as linking or cutting trees are needed. A similar reasoning to the one above can be applied to adapt these techniques.

6. A Few Closing Words.

We have shown that it is possible to go from linear lists to free trees without forsaking all the efficient techniques developed in the context of lists. Moreover, the extensions can be defined in a fairly automatic manner by applying certain elementary transformations. It might be interesting to characterize formally the domain of problems and techniques amenable to that *transformational* approach. Also, one might wonder what happens if arbitrary

graphs are substituted for free trees. Difficulties arise because of the nonuniqueness of a path between two vertices. Restricting oneself to shortest paths or to planar graphs, or perhaps representing paths as regular expressions, are possible approaches to the general problem, which we believe deserves further study.

Acknowledgments. I wish to thank Bob Tarjan for his valuable comments on this work. I also wish to thank the referees for their useful suggestions to improve the presentation of this paper.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. L. Bentley, Multidimensional divide and conquer, *Comm. ACM*, **23** (1980), 214–229.

[3] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, *SIAM J. Comput.* (in press). Preliminary version in *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 165–174.

[4] B. Chazelle and L. J. Guibas, Fractional cascading: I. A data structuring technique, *Algorithmica*, **1** (1986), 133–162.

[5] H. Edelsbrunner, A new approach to rectangle intersections, Part II, *Internat. J. Comput. Math.*, **13** (1983) 221–229.

[6] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, Scaling and related techniques for geometry problems, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, 1984, pp. 135–143.

[7] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, **13** (1984), 338–355.

[8] D. E. Knuth, *The Art of Computer Programming*, Vol. I, 2nd edn., Addison-Wesley, Reading, MA, 1973

[9] J. Komlós, Linear verification for spanning trees, *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 1984, pp. 201–206.

[10] E. M. McCreight, Efficient algorithms for enumerating intersecting intervals and rectangles, Technical Report CSL-80-9, Xerox Parc, 1980.

[11] E. M. McCreight, Priority search trees, *SIAM J. Comput.*, **14** (1985), 256–276.

[12] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. System Sci.*, **26** (1983), 362–391.

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

[14] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.*, **22** (1975), 215–225.

[15] R. E. Tarjan, Applications of path compression on balanced trees, *J. Assoc. Comput. Mach.*, **26** (1979), 690–715.

[16] J. Vuillemin, A unifying look at data structures, *Comm. ACM*, **23** (1980), 229–239.

[17] D. E. Willard, New data structures for orthogonal range queries, *SIAM J. Comput.*, **14** (1985), 232–253.

[18] A. C. Yao, Space-time tradeoff for answering range queries, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, 1982, pp. 128–136.