1. Introduction

In the same way as pushdown automata generalize finite state automata, free trees generalize linear lists. This generalization relates important combinatorial objects as well as the abstract data types which 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) becomes a problem of maintaining "active" paths in a tree (with application to, say, communication networks). Since, after all, linear lists are far better understood than free trees, it is tempting to ask the question: can one use the wealth of techniques available for linear lists to solve problems on trees, or does the added generality of the latter type prevent any systematic "transfer"?

To be sure, difficult questions on trees have already been answered. For example, important work has been done recently on the problems of computing kth longest paths, p-centers, weighted centers, or lowest common ancestors [FJ,HT,Me,MT,T3]. To our knowledge, however, no attempt has yet been made to systematically map list-based query-answering methods to the more complex case of trees. The idea behind this approach is methodological in nature. It involves abstracting an algorithmic technique from a particular mathematical model and removing its unnecessary assumptions. This is useful because data structures are often discovered within a fairly restrictive context which may obscure a general principle.

Using a centroid theorem for divide-and-conquer purposes, we consider a number of data structuring techniques for linear lists (e.g. Yao's interval query scheme [Y], range tree [BS], segment tree [BW], interval tree [E,Mc]), and generalize them (i.e. map them) into techniques for free trees. Of course, we restrict our attention to mappings which preserve the complexity of the algorithms or at least keep the algorithms within the same complexity class. Here is a summary of our main results; in each case, we indicate the nature of the mapping used:

1. Let \((S,+)\) be a semigroup and let \(T\) be a free tree. Define a function \(F: \{\text{paths of } T\} \rightarrow S\) as follows: \(F(e)\) is given for each edge \(e \in T\), and \(F(t) = \sum_{1 \leq i \leq k} F(e_i)\), for any path \(t = e_1 \ldots e_k\) of \(T\) (think of \(F(t)\) as, say, the length of the path or the maximum edge-weight in \(t\), etc). In the arithmetic model (where only operations and not memory retrievals are charged), we establish an optimal class of complexity trade-offs for coinputing \(F\), by generalizing a technique of Yao [Y]. We prove that if \(m\) memory cells are available, \(F(t)\) can be computed in time \(O(\alpha(m, n) + \frac{n}{m-n+1})\), where \(\alpha\) is the functional inverse of Ackermann's function, as defined by Tarjan [T1].

2. We look at the previous problem in the pointer machine model (where no address arithmetic is allowed). We show that with only linear storage, \(F(t)\) can be evaluated in \(O(\log n)\) time (mapping from range trees).

3. We use a case-study to develop our mapping of segment trees. We consider the problem of maintaining a communication network where edges are assigned
vertex-count. In the following, we use the term "subtree" of \( T \) to refer to any connected subgraph of \( T \). We define edges in Lemma 1.

The following results are well-known (see [K] for Lemma 1; apply it recursively to obtain Lemma 2).

**Lemma 1.** In \( O(|T|) \) time and space, it is possible to decompose \( T \) into two trees \( T_1, T_2 \) such that \( \frac{1}{2} |T| \leq |T_1| \leq |T_2| \leq \frac{3}{2} |T| \).

**Lemma 2.** Let \( k \) be an arbitrary integer \( (1 \leq k \leq |T|) \). There exists a partition of \( T \) into subtrees \( T_1, \ldots, T_p \), such that for each \( i \) \( (1 \leq i \leq p) \) we have \( k/3 < |T_i| \leq k \).

**Lemma 3.** 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 \( T \). Each edge \( e \) is visited twice, once at step \( f(v) \) and the second time at step \( g(v) \), with \( 1 \leq f(v) \leq g(v) \leq 2|T| \). Label each vertex \( v \) by \( l(v) = \lfloor \min f(e) \rfloor \) if \( e \) adjacent to \( v \). To ensure that each vertex has a distinct label, 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 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 \( l(u) \notin [f(e), g(e)] \).

**Remark:** while Lemmas 1, 2 imply that lists and trees have similar separator theorems, Lemma 3 shows an important discrepancy between these two types of objects. Indeed, whereas only three indices are involved in testing whether an element lies in between two others in a linear list, four indices are used in the case of trees. This will cause some loss of performance in some of the mappings described later on, specifically because \( k \)-dim query problems over lists will map into \((k+1)\)-dim problems over free trees.

### 3. Decomposable Functions in the Arithmetic Model

Let \((S, +)\) be a semigroup and let \( T \) be a free tree. A function \( F : \{\text{paths of } T\} \to S \) is called decomposable if: 1) \( F(e) \) is given for each edge \( e \in T \); 2) \( F(t) = \sum_{1 \leq i \leq k} F(e_i) \), for any path \( t = e_1, \ldots, e_k \) of \( T \). In the model of computation considered in this section, the memory consists of \( m \geq n \) units of information \( q_1, \ldots, q_m \), each of the form \( q_i = \sum_{1 \leq i \leq k} \lambda_{i,j} F(e_j) \) \((\lambda_{i,j} \text{ integral } \geq 0)\). Let \( F(u,v) \) denote the value of \( F \) at the path of \( T \) from \( u \) to \( v \). An algorithm \( A \) for computing \( F \) is called a \((t, \alpha)\)-scheme if \( m \leq \alpha n \), and \( A \) computes \( F(u,v) \) by evaluating a sum of the form \( F(u,v) = \sum_{1 \leq k \leq \mu} \mu_k q_{\mu} \) for \( \mu_k \) integral \( \geq 0 \). The quantities \( t \) and \( m \) represent respectively the time of the computation and the storage used. The following result is a nontrivial generalization of a technique of Yao for computing interval queries [Y].

**Theorem 1.** Given a decomposable function \( F \) defined over a free tree \( T \) with \( n \) edges, it is possible to precompute \( m \) values of \( F \) so that evaluating \( F \) at any path of \( T \) can be done in time \( O\left(\alpha(m,n) + \frac{n}{m-n+1}\right) \). This trade-off is optimal.

The remainder of this section is devoted to proving Theorem 1. From Lemma 2, we derive the two relations,

\[
k/3 < |T_i| \leq k,
\]
Informally speaking, the subtrees of the first case can be done by a recursive call, while in the other case, precomputed values of \( P \) associated with each super node will be sufficient to evaluate \( F \) at \( P \). The remainder of this section is devoted to developing this argument in more detail and analyzing the complexity of the algorithm.

Let \( T = \{ T_1, \ldots, T_p \} \) denote the partition of Lemma 2. We call the set of vertex-pairs joined during the process an interconnection pattern for \( T \). Let \( C(T) \) denote the set of all interconnection patterns for \( T \). For any \( R \in C(T) \), we have a unique resulting tree, \( T_R \) which can thus designate by the functional notation, \( T_R \). 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 \( T_j (j \neq i) \). This set, called the fringe of \( T_i \), induces a subtree of \( T_i \), denoted \( Q_i \). 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 isomorphic 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 \( T_R^* \). Again, note that \( T_R^* \) is isomorphic to \( T_R \).

We are now ready to describe the underlying data structure, denoted \( DS(T) \). From now on, let \( T = \{ T_1, \ldots, T_p \} \) denote the partition of \( T \) given in Lemma 2 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 \( DS(T) \) recursively, as follows: \( DS(T) = (A, B, C) \), with \( A = \{ DS(T_1), \ldots, DS(T_p) \} \), \( B = DS(T_R^*), \) 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 \( T \) that contains the edge adjacent to \( u \) (resp. \( v \)) on the path between \( u \) and \( v \). If \( i = j \), recurse on \( DS(T_i) \). Otherwise, let \( f_u \) (resp. \( f_v \)) be the first (resp. last) fringe vertex on the directed path from \( u \) to \( v \). The relation \( F(u, v) = F(f_u, f_v) + F(f_v, f_u) \) 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 \( DS(T_R^*) \). Let \( q(T) \) denote the maximum number of arithmetic operations necessary to compute \( F(u, v) \), for any two vertices \((u, v)\) in \( T \). We have

\[
q(T) \leq 5q(T_R) + 2q(T), \ldots, q(T_p). \tag{3}
\]

Let's now examine the storage \( |DS(T)| \) required by the data structure. Let \( d_i \) be the number of vertices in the fringe of \( T_i \). Each such vertex contributes \( d_i - 1 \) 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

\[
\sum_{1 \leq i \leq p} (|T_i| + 1 - d_i) d_i + d_i(d_i - 1) = \sum_{1 \leq i \leq p} d_i|T_i|.
\]

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 added number of fringe vertices in all the \( T_i \)'s. We can show by induction that this number is at most \( 2(p - 1) \) — to do so, first show that at least one \( T_i \) has only one vertex common to another \( T_j \); details are omitted. It follows that \( |C| \leq 2pk \), which from (2) leads to \( |C| < 6|T| \). We conclude that

\[
|DS(T)| \leq \sum_{1 \leq i \leq p} |DS(T_i)| + |DS(T_R^*)| + 6|T|. \tag{4}
\]

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

\[
|T_R^*| \leq 4p. \tag{5}
\]

360
Keeping the parameter \( k \) 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 pre-specified complexity can always be found. We introduce the function \( D(t, \alpha) \), defined for all integers \( t \geq 1, \alpha \geq 0 \) as follows: \( D(t, 0) = 0 \); for \( \alpha > 0 \), \( D(t, \alpha) = \max\{n | \text{free tree } T | T| \leq n, \exists (t, \alpha) \text{-scheme for } T \} \). 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 pre-specified performance. Let \( p(t, \alpha) \) be the maximum value of \( p \) such that for any \( T = \{T_1, \ldots, T_p\} \) with \( |V_i| (1 \leq i \leq p) \), \( \frac{D(t, \alpha)}{3} < |T_i| \leq D(t, \alpha) \) and for any \( R \in C(T) \), there exists a \((t, \alpha + 18)\)-scheme for \( T_R \). The growth of \( D(t, \alpha) \) follows two key inequalities.

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

**Proof:** It suffices to show that there exists a \((t, \alpha + 18)\)-scheme for any tree \( T \) with at most \( \frac{p(t, \alpha)}{3} D(t, \alpha) \) edges. We can assume that \( |T| > D(t, \alpha) \), otherwise the result is obvious. This allows us to follow the lines of Lemma 2, and partition \( T \) into \( p \) subtrees \( T_1, \ldots, T_p \) such that, for each \( i \) \((1 \leq i \leq p)\), we have \( \frac{D(t, \alpha)}{3} < |T_i| \leq D(t, \alpha) \). 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) \( \frac{1}{3} D(t, \alpha) p(t, \alpha) \leq p(t, \alpha) \), 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 \( D(t - 2, D(t, \alpha)) \geq \frac{4}{15} D(t - 2, D(t, \alpha)) \).

**Proof:** Let \( T = \{T_1, \ldots, T_p\} \) be a set of free trees such that \( \frac{1}{15} D(t - 2, D(t, \alpha)) \). Assume furthermore that for each \( i \) \((1 \leq i \leq p)\), we have \( \frac{D(t, \alpha)}{3} < |T_i| \leq D(t, \alpha) \). Let \( R \) be an arbitrary interconnection pattern of \( C(T) \); we will show that \( T_R \) 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. \( A = \{DS(T_1), \ldots, DS(T_p)\} \). For each \( T_i \), we set up a \((t, \alpha)\)-scheme, which is certainly possible since \( |T_i| \leq D(t, \alpha) \).

2. \( B = DS(T_R^+) \). From (5) we know that \( T_R^+ \) has no more than \( 4p \) edges. Since \( p \leq \frac{1}{4} D(t - 2, D(t, \alpha)) \), we have \( |T_R^+| < D(t - 2, D(t, \alpha)) \); therefore there exists a \((t - 2, D(t, \alpha))\)-scheme for \( T_R^+ \).

3. \( C = \{V_1, \ldots, V_p\} \). Defined as previously.

From (3), it follows directly that the scheme proposed involves at most \( t \) arithmetic operations for any evaluation of \( F \) over \( T_R \). Let \( S \) represent the amount of storage required by the algorithm. From (4) we have \( S \leq \sum_{1 \leq i \leq p} \alpha |T_i| + 4p |D(t, \alpha) + 6 |T| \). From the hypothesis that \( \frac{D(t, \alpha)}{3} < |T_i| \), we derive \( \sum_{1 \leq i \leq p} |T_i| > \frac{D(t, \alpha)}{3} p \) hence \( p < \frac{3 |T|}{D(t, \alpha)} \). This implies that \( 4p |D(t, \alpha) < 12 |T| \), which yields \( S < |T| + 12 |T| + 6 |T| = |(\alpha + 18) |T| \). We have thus shown that \( T_R \) admits of a \((t, \alpha + 18)\)-scheme, which completes the proof.

**Lemma 6.** For any \( t \geq 2 \) and any \( \alpha \geq 1 \), we have \( D(t, \alpha + 18) \geq \frac{4}{15} D(t - 2, D(t, \alpha)) \).

**Proof:** A direct consequence of Lemmas 4 and 5.

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

**Lemma 7.**

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

**Proof:** \( B(0,j) = \left[ \frac{1}{18} D(2, 18j) \right] \). Assume that \( j > 0 \); a simple solution for achieving time \( \leq 2 \) 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 \( \left( \frac{T}{2} \right)^{-1} + |T| \), so there is a \((2, 18j)\)-scheme for all \( T \) such that \( \left( \frac{T}{2} \right)^{-1} + |T| \leq 18j |T| \), i.e., \( |T| \leq 36j + 1 - \frac{|T|}{2} \), from which we easily derive (1). It is clear that the function \( B(i,j) \) is monotone (i.e., non-decreasing with respect to \( i \) and \( j \)), therefore for all \( i \geq 1 \) we have \( B(i,1) \geq B(0,1) \geq 2 \). Also, from Lemma 6, we derive

\[
B(i,j) \geq \left[ \frac{18}{12} B(i, j-1)B(i-1, B(i, j-1)) \right],
\]
from which (3) follows, since \( B(i,j-1)B(i,B(i,j-1)) \geq B(0,j-1)B(0,B(0,j-1)) \geq 8(j-1)^2 > 2. \)

We will 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 [T1]. Let \( A(i,j) \) be the function defined recursively as follows:

\[
A(0,j) = 2j, \text{ for any } j \geq 0.
\]

\[
A(i,0) = 0 \text{ and } A(i,1) = 2, \text{ for any } i \geq 1.
\]

\[
A(i,j) = A(i-1, A(i,j-1)), \text{ for any } i \geq 1 \text{ and } j \geq 2.
\]

For any \( m \geq n \geq 1 \), we define the function \( \alpha(m,n) \) by

\[
\alpha(m,n) = \min\{i \mid i \geq 1, A(i,4\lfloor m/n \rfloor) > \log_2 n\}, \]

and, for any real number \( x \) and integer \( j \geq 0 \), we define \( a(x,j) \) by

\[
a(x,j) = \min\{i \mid i \geq 1, A(i,j) > x\}. \tag{6}
\]

**Lemma 8.** The function \( A(i,j) \) is monotone.

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

**Proof:** Equation (11) in [T1] states that for any \( i \geq 0, j \geq 1 \), we have \( A(i+1,j+1) \geq A(i,2j') \). Applying this inequality iteratively, we complete the proof by observing that for any \( i \geq 4, j \geq 3 \), we have \( A(i,j) \geq A(1,2i-1) \geq A(1,2^{2i-1}+2i-1) \geq \ldots \geq A(1,4,300j) \).

**Lemma 10.** For any \( x, j \) with \( z \geq 1 \) 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 \( \alpha(m,n) = a(\log_2 n, 4\lfloor m/n \rfloor) \).

**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 fewer than \( m \) words 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, \( f(m,n) \leq \min\{i \mid n \leq D(i,\lfloor m/18n \rfloor)\} \). Since \( D \) is monotone, we also have \( f(m,n) \leq \min\{2i+2 \mid 18n \leq D(2i+2,18\lfloor m/18n \rfloor)\} \). Now, because \( \lfloor m/18n \rfloor \leq \lfloor m/18 \rfloor \), we have \( \lfloor m/18 \rfloor \leq \lfloor m/18n \rfloor \), which implies \( f(m,n) \leq \min\{2i+2 \mid 18n \leq D(2i+2,18\lfloor m/18n \rfloor)\} \). We derive

\[
f(m,n) \leq \min\{2i+2 \mid n \leq B(i,\lfloor m/18n \rfloor)\}. \tag{7}
\]

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

\[
g(m,n) \leq a(n,\lfloor m/18n \rfloor). \tag{8}
\]

Next we apply Lemma 9, with \( j = \lfloor m/18n \rfloor \) and \( j' = 4\lfloor m/n \rfloor \). We can easily check that since \( m \geq 72n \), we have \( 3 \leq j \leq j' \leq 300j \), which directly implies that \( g(m,n) \leq a(n,4\lfloor m/n \rfloor) + 4 \), and from Lemmas 10,11, we derive \( g(m,n) = O(a(\log_2 n, 4\lfloor m/n \rfloor)) = O(\alpha(m,n)) \). 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:** \( 1 \leq m \leq 72n \).

The idea is to break up the tree \( T \) into \( p \) subtrees \( T = \{T_1, \ldots, T_p\} \), applying the results of Lemma 2, 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 \( T_R \), and the two others in some of the subtrees of \( T \). The difference with the previous method is that the procedure will now recur only with respect to the path in \( T_R \), 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 \( T \) and use the \( m - n \) remaining words of storage to apply the technique of the first case to \( T_R \). We wish to prove that in all cases we can achieve a space-time trade-off of the type

\[
f(m,n) = O(n(\alpha(m,n) + \frac{n}{m - n + 1})). \tag{9}
\]
In order to apply the previous scheme to $T_R^T$, we must assume that the available storage for $T_R^T$ is at least 72 times the number of edges in $T_R^T$. For this reason, we will 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 $T_R^T$ (see Lemma 2).

**Lemma 14.** If $k = \lceil \frac{822n}{m-n} \rceil$, the decomposition of Lemma 2 leads to $|T_R^T| < \frac{1}{2}(m-n)$.

**Proof:** We may assume that $k = \lceil \frac{822n}{m-n} \rceil \leq n$, since otherwise Relation (9) is trivially satisfied. This assumption allows us to apply (2,5). From these relations we derive $|T_R^T| \leq 4p < 122n$, hence $|T_R^T| < 12n \frac{m-n}{m-n} = \frac{1}{2}(m-n)$.  

We are now back to Case 1 with respect to $T_R^T$. We use $n$ words of storage for all the trees $T_1, \ldots, T_p$ and $m-n$ words for the tree $T_R^T$. From Lemma 14 we derive an upper bound on $f(m,n)$, the maximum number of precomputed operands needed in any given evaluation of $F$; $f(m,n) \leq O(\alpha(72|T^2| + |T_R^T|)) + 2 \max_{1 \leq i \leq p} |T_i|$. Using Relation (1), Lemma 12, $|T_R^T| \leq n$, and $m < 22n$, we conclude that $f(m,n) \leq 2k + O(\alpha(72n,n)) = \lceil \frac{1228n}{m-n} \rceil + O(\alpha(m,n))$, which establishes Relation (9).

In all cases, we have shown that $f(m,n) = O(\alpha(m,n) + \frac{n}{m-n+1})$. Optimality follows from a lower bound by Yao [Y] for the special case of a linear list. The proof of Theorem 1 is now complete.

4. Mapping Range Trees

In this section, we look at the problem of computing decomposable functions in a different (perhaps more realistic) model of computation: pointer machines. This model of computation is tailored for algorithms that gain access to memory via pointer manipulation [T2]. This rules out address arithmetic, and hence operations like hashing, radix sort, or random access into dense matrices. To evaluate $F$, we preprocess $T$ so that any path can be rewritten as the concatenation of $O(\log |T|)$ canonical subpaths. The linear list equivalent of our technique is the notion of one-dimensional range tree [BS].

Let $T_1$ and $T_2$ be the two subtrees of Lemma 1. We encode $T$ by means of a virtual binary tree $\bar{T}$ as follows: associate the root of $\bar{T}$ with $T$, and its two children with $T_1$ and $T_2$, respectively. In general each node $v$ of $\bar{T}$ will be associated with a subtree of $T$, denoted $T(v)$. Recursive application of Lemma 1 completes the definition of $\bar{T}$, with the understanding that every leaf $l$ of $\bar{T}$ has its associated tree, $T(l)$, made up of a single edge. Note that each node $v$ of $\bar{T}$ thus corresponds to a unique partitioning vertex of $T$, denoted $\sigma(v)$ ($\sigma$ is in general non-injective). If $v$ is an internal node of $\bar{T}$, $\sigma(v)$ is the vertex of $T$ whose removal leads to the decomposition associated with node $v$. If $v$ is a leaf of $\bar{T}$, let $uu'$ denote the unique edge of $T(v)$ and let $w$ be the parent of $v$. Observing that $\sigma(uw)$ is a vertex of $uu'$, say $u$, we define $\sigma(v)$ to be $u'$. $\bar{T}$ is clearly near-balanced, with a height $\leq \log_{1,4} |T| + O(1)$.

Observe that $\bar{T}$ induces a canonical decomposition of any path $P$ in $T$. We develop this notion by introducing additional edges. For any pair $u,v \in T$, let $P(u,v)$ designate the path of $T$ between $\sigma(u)$ and $\sigma(v)$. For each internal node $v \in \bar{T}$ in turn, apply the following procedure. Let $l_v$ (resp. $r_v$) be the left (resp. right) child of $v$. We describe the procedure with respect to $l_v$ only, with the understanding that it must also be applied to $r_v$. Let $l_1, \ldots, l_k$ be the sequence of nodes of $\bar{T}$ defined as follows. $l_{i+1}$ is the child of $li$ with $\sigma(v) \in T(l_{i+1})$, defined only if this child is unique. The sequence stops as soon as $li$ is a leaf or $\sigma(li) = \sigma(v)$, whichever happens first. Let $k$ be the largest value of $i$ achieved. We now add the edges $(v,l_1), \ldots, (v,l_k)$ to $\bar{T}$ (Fig.1A). Note that $(v,l_1)$ is already in $\bar{T}$. The sequence $((v,l_1), \ldots, (v,l_k))$ is denoted $L(v)$ and is kept in this order at node $v$. Each edge $(v,w) \in \bar{T}$ makes the path $P(v,w)$ canonical, i.e. part of the alphabet over which each path in $T$ can be expressed. We omit the proof that the augmented tree (still denoted $\bar{T}$ for convenience) has $O(|T|)$ (old and new) edges. Furthermore each node is adjacent to at most $O(\log |T|)$ edges.

**Lemma 15.** Any path of $T$ can be rewritten as the concatenation of canonical paths. The decomposition can be computed in $O(\log |T|)$ time, after preprocessing.
Proof: Let $P$ denote the path to be canonically 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 $T$, i.e. whether each subtree contains at least one node which, through $\sigma$, maps $i$ to a vertex of $P$. To set the conditions of Lemma 3, we conceptually duplicate each partitioning vertex of $T$ and replace it by a dummy edge. This will be accomplished by examining each node of $T$ level by level in a top-down fashion. Let $v$ be an internal node of $T$ with children $v_1, v_2$. Replace $\sigma(v)$ in $T$ by a dummy edge $(\sigma(v), \sigma'(v))$ with $T(v_1)$ attached to $\sigma(v)$ and $T(v_2)$ attached to $\sigma'(v)$. The dummy edge is canonically associated with node $v$ of $T$. If $v$ is not the root, note that it might be attached to other edges not in $T(v_1)$ or $T(v_2)$. For consistency each leaf $v$ also has an edge of $T$ associated with it, i.e. $(\sigma(v), \sigma(v))$, with $w$ the parent of $v$. In Fig.1B, dummy edges have been labelled with superscripts indicating their level in $T$.

For any path $P$, there is a single node in $T$, denoted $v(P)$, which plays a special role in the canonical decomposition of $P$. We define $\nu(P)$ (algorithmically) as follows: the previous preprocessing allows us to apply Lemma 3 to decide, in constant time, whether $P$ lies totally in $T(v_1)$ or $T(v_2)$, for any node $v \in T$ with children $v_1, v_2$. Starting with $v = \text{root of } T$, we recurse in $T(v_1)$ (resp. $T(v_2)$) if we fall in the former (resp. latter) case. Otherwise, set $\nu(P) = v$. Note that in general $\nu(P)$ is the highest-level node of $T$ such that $P$ can be rewritten as the concatenation of $(P_1, (\sigma(\nu(P)), \sigma'(\nu(P))), P_2)$, with at least one of the following conditions holding true: 1) $P_1$ is null; 2) $P_2$ is null; 3) $P_1$ (resp. $P_2$) has a vertex $\sigma(v)$ (resp. $\sigma(w)$) with $v$ and $w$ not in the same subtree rooted at $\nu(P)$. Note that, in all cases, $\nu(P)$ can be computed in $O(\log |T|)$ time. Once $\nu(P)$ is available, the search splits into two directions (or just one, if either $P_1$ or $P_2$ is null). Wlog, we outline the procedure for $P_1$, which we assume to be non-null. Since the procedure is recursive, a running variable $v$ is initially set to $\nu(P)$. From previous observations, we can determine in constant time whether the computation should proceed in the left or right subtree of $v$ (it will proceed in only one such subtree if $v \neq \nu(P)$). Wlog assume that it proceeds in the left. Let $L(v) = \{(v, i_1), \ldots, (v, i_k)\}$; compute $j$ the smallest index $i$ such that $P$ contains the dummy edge $i$. If none is found, set $j = k$. Collect the edge $(v, i_j)$ and replace $i_j$ by a dummy edge. This will be accomplished by adummy edge conceptually duplicate each partitioning vertex of $T$, using $O(n)$ space, so that a decomposable function $F$ can be evaluated at any point $(u, v)$ in $O(\log n)$ time.

Proof: Direct corollary of Lemma 15. Each edge $(v, w)$ of $T$ points to the precomputed value, $F(P(v, w))$.

5. Mapping Segment-Trees: A Case-Study

Suppose that the free tree $T$ is a communication network, where each edge $e$ is given a capacity $c(e)$ to designate, say, the maximum number of communications the edge $e$ can support simultaneously. At any given time the network has a number of active communication lines. A communication line is defined by a pair of vertices and is thus associated with a unique path in $T$; a given path may support several communication lines. The problem is to construct a data structure to grant or deny incoming communication requests according to the overall saturation of the requested path. A path is said to be saturated if any of its edges has achieved its full capacity. Each communication line contributes one to the saturation index of each of its edges. First, we describe an algorithm for the case where $T$ is a linear list, which we then extend to handle the general case. This extension involves a nontrivial mapping of the notion of segment-tree [BW].

Lemma 16. It is possible to preprocess a communication network consisting of $n$ consecutive links, using $O(n)$ storage, so that any communication request can be answered in $O(\log n)$ time.

Proof: Let $T$ be a chain of edges, $(e_1, \ldots, e_n)$. We set up a complete binary tree whose leaves are, from left to right,
$e_1, \ldots, e_n$. Let $C = \{P_1, \ldots, P_k\}$ be the multiset of communication lines. We represent $C$ by means of a segment tree $D$ but instead of storing each interval $P_i$ in the appropriate node-lists, we only keep track of the cardinality of each node-list. In other words, each node $v$ of $T$ stores the number $n(v)$ of intervals $P_i$ for which it contributes a canonical part. We wish to distinguish between canonical parts and communication parts. Any path in $T$ is decomposed into canonical parts, while any communication line is decomposed into canonical lines. Thus, there is a non-injective correspondence between canonical lines and canonical parts. Let $D(v)$ be the subtree of $D$ rooted at $v$. Considering $D(v)$ in isolation from the rest of $D$, let $e_i$ be the leaf of $D(v)$ that is the closest to saturation. More formally put, for each canonical line, we only keep track of the cardinality of each node-list. Thus, there is a non-injective correspondence between canonical lines and canonical parts. Let $D(v)$ be the subtree of $D$ rooted at $v$. Considering $D(v)$ in isolation from the rest of $D$, let $e_i$ be the leaf of $D(v)$ that is the closest to saturation. More formally put, for each leaf $e_j$ of $D(v)$, we define $d_j$ as the number of canonical lines in $D(v)$ that cover $e_j$. Note that $d_j$ represents the added cardinality of the node-lists on the path from $v$ to the leaf $e_j$. The edge $e_i$ is the one that minimizes the quantity $c(e_j) - d_j$. Along with the cardinality of the node-list, we will store in $v$ the pair $p(v) = (e_i, c(e_i) - d_i)$. Since each node of $D$ keeps only a constant amount of information, the size of the data structure is $O(n)$, which is optimal. Wlog, assume that $k$ is bounded above by a polynomial of $n$, i.e. the number of bits needed to count the number of communication lines is of the same order of magnitude as the length of any pointer in $D$.

The tree $D$ acts both as a segment-tree (without the node-lists) and a priority queue. For this reason, updating $D$ can be made very efficient. Suppose that we wish to terminate a communication $P_i$; we simulate a search in the segment tree to retrieve the canonical decomposition of $P_i$. For each node $v$ of the decomposition, decrement $n(v)$ by one and reset the pair $p(v) = (e_i, c(e_i) - d_i)$ to the value $(c_i, c(e_i) - d_i + 1)$. This may upset the priority structure of $D$, in that the pair $p(w)$ associated with the parent, $w$, of $v$, might no longer be relevant. We fix this anomaly by comparing the new value of $p(v)$ against the value $p(z)$ of its brother $z$; the pair with the smaller second term will "win" and determine the updating of $p(w)$. This in turn may upset the father of $w$, which we update in a similar fashion. This procedure will propagate all the way from $v$ to the root of $D$, mimicking the updating of a heap. Each of these updates will take $O(\log n)$ time, so the overall time cost of the deletion will be $O(\log^2 n)$. We can reduce this cost to $O(\log n)$ by batching the updates together. This is fairly straightforward, so we will omit the details. The main observation is that the nodes visited in all the updates forms a subtree of $D$, denoted $t$, (in the graph-theoretical sense) of size $O(\log n)$. Let's make $t$ into a directed tree by oriented each edge towards the root. We can then carry out all the necessary updates by pebbling the tree $t$, which will take time linear in its size.

Inserting a new communication proceeds in exactly the same manner, substituting "increments" for "decrements".

The last point to examine concerns the question of granting or denying an incoming request. To do so, we decompose the requested communication path into its canonical parts, at a cost of $O(\log n)$ steps. For each node $z$ visited, we maintain the sum $\sum w n(w)$, taken over each ancestor $w$ of $z$. This ensures that when reaching any node $v$ associated with a canonical part of the requested path, we can check in constant time whether the edge $e_i$ from the pair $p(v) = (e_i, A)$ is saturated. Once again, we batch the computation by storing intermediate values in the searched subtree, which ensures an $O(\log n)$ run time.

Next we turn to the general case where $T$ is an arbitrary free tree. Our first task is to preprocess $T$ along the lines of Section 4. To simplify the exposition we introduce some terminology. As before, the term "edge of $T" refers to any edge of $T$ (added or not). We will reserve the use of the term "edge" to $T$, and in the following the edges of $T$ will be called links. Recall that for any edge $(u,v)$ in $\mathcal{T}$, either $u$ is a descendent of $v$, or $v$ is a descendent of $u$. For simplicity, we will always denote an edge as an ordered pair $(u,v)$, with $v$ the descendent of $u$. We define a partial order $\preceq$ on the set of edges of $T$. Given two edges $(u_1,v_1)$ and $(u_2,v_2)$, we say that $(u_1,v_1) \preceq (u_2,v_2)$ iff 1) $u_1 = u_2$ and $v_1$ is a descendant of $v_2$, or 2) $u_1$ is a descendant of $u_2$. The entire algorithm is based on the following simple result, whose proof is omitted. Recall that $P(u,v)$ is the path in $T$ between $\sigma(u)$ and $\sigma(v)$. We assume in the fol-
Lemma 17. If two paths \( P(u_1, v_1) \) and \( P(u_2, v_2) \) have at least one link in common, then the two edges \((u_1, v_1)\) and \((u_2, v_2)\) are comparable, i.e. \((u_1, v_1) \leq (u_2, v_2)\) or \((u_2, v_2) \leq (u_1, v_1)\) (or both). Furthermore, if we regard \( P(u_1, v_1) \) and \( P(u_2, v_2) \) as chains of links, their intersection always constitutes a single subchain.

The main idea of the algorithm is to set up the structure of Lemma 16 for each canonical path. For each edge \((u, v)\) in \( T \), let \( D(u, v) \) denote the data structure defined with respect to the chain \( P(u, v) \). Next we specify which communications should be represented in \( D(u, v) \). Let \( \mathcal{C} = \{C_1, \ldots, C_p\} \) be the set of communication lines in \( T \) at some arbitrary time \( t \) and let \( n = |T| \). The tree \( T \) induces a decomposition of each communication line into \( O(\log n) \) canonical sub-lines. Let \( \mathcal{C}' \) denote the set of all sub-lines in \( T \) at time \( t \). Let \( V \) be the multiset consisting of all the edges \((w_1, w_2)\) of \( T \), such that \( 1) P(w_1, w_2) \in \mathcal{C}' \), \( 2) P(w_1, w_2) \cap P(u, v) \neq \emptyset \), \( 3) (w_1, w_2) \leq (u, v) \). \( V \) is precisely the set of communications represented in \( D(u, v) \) at time \( t \). \( D(u, v) \) provides us with a means to assess the saturation of any subchain of \( P(u, v) \), taking only into consideration the sub-lines \( P(w_1, w_2) \) with \((w_1, w_2)\) in the subtree rooted at \( u \) containing \( v \).

We complete our description of the data structure by defining the set \( I(u, v) \). Informally, \( I(u, v) \) indicates the interaction between \( P(u, v) \) and paths of the form \( P(u', v') \), with \((u, v) \leq (u', v')\). From Lemma 17, we know that the intersection of \( P(u, v) \) and \( P(u', v') \), if non-empty, is a chain of the form \((e_1, \ldots, e_j)\). We define \( I(u, v) \) as the set of pairs \([u', v'], (i, j)\] , taken over all edges \((u', v')\) such that \((u, v) \leq (u', v')\) and \( P(u') \cap P(u, v) \neq \emptyset \). Note that since for a fixed \( u' \) there are at most \( O(\log n) \) edges of the form \((u', v')\), the cardinality of \( I(u, v) \) is \( O(\log^2 n) \). We omit the proof that the overall size of the data structure is \( O(n \log n) \) (assuming as usual that \( k \) is bounded by a polynomial of \( n \)).

**Theorem 3.** Using \( O(n \log n) \) storage, it is possible to preprocess a communication network consisting of \( n \) links arranged in a free tree, so that any communication request can be answered in \( O(\log^4 n) \) time \((c \leq 3)\).

**Proof:** Let \( P \) be a communication line to be inserted. Using Lemma 15, we decompose \( P \) into its \( O(\log n) \) canonical sub-lines, and proceed to insert each sub-line into the data structure, one at a time. Let \( l \) be the current sub-line under consideration, and let \((u, v)\) be its associated edge in \( T \). Obviously, only structures \( D(w_1, w_2) \) such that \((u, v) \leq (w_1, w_2)\) need be updated. But the set of all edges \((w_1, w_2)\) to be considered coincides exactly with the set of first elements in \( I(u, v) \). Let \([([w_1, w_2], (i, j)]\) be a pair of \( I(u, v) \). From Lemma 16, we can insert the line \((i, j)\) into \( D(w_1, w_2) \) in \( O(\log^4 n) \) time. Iterating on this process for each element in \( I(u, v) \) and for each sub-line \( l \) leads to an \( O(\log^4 n) \) insert-time. We observe that deadlines can be handled in exactly the same manner, so we turn directly to the question of deciding whether an incoming request should be granted or denied.

To do so, we must provide some efficient scheme for checking the saturation index of each link on the requested path \( P \), without looking at each of them exhaustively. Once again, let \( l \) be the generic canonical sub-line of \( P \), with \((u, v)\) its edge in \( T \). We first check in \( D(u, v) \) whether the entire chain \( P(u, v) \) can support an additional line stretching all across the chain. If yes, we consider each element \([([w_1, w_2], (i, j)]\) of \( I(u, v) \), and check whether \((i, j)\) can be inserted into \( D(w_1, w_2) \). If all these tests are positive, we claim that \( T \) can accept the sub-line \( l \) (we postpone the proof of this claim for later on). If this is the case, it suffices to proceed in a similar manner for all the sub-lines of \( P \), accepting the request iff not a single test fails. As observed previously, Lemma 16 shows that the entire operation will be complete after \( O(\log^4 n) \) steps. We next prove the major claim of this argument.

First of all, we observe that if \( P \) should indeed be accepted, none of the tests can possibly fail. Assume now that the link \( e \) of \( P \) is saturated. This is caused by the presence of exactly \( c(e) \) communication lines passing through \( e \).
Let \((u_i, v_i), \ldots, (u_{c(e)}, v_{c(e)})\) be the edges associated with the sub-lines (passing through \(e\)) of these \(c(e)\) communication lines. From Lemma 17, all these edges are comparable (with respect to \(\leq\)) so we turn our attention to \((u_i, v_i)\), with \((u_j, v_j) \leq (u_i, v_i)\) for all \(j\) \((1 \leq j \leq c(e))\). By definition, \(D(u_i, v_i)\) must give evidence that \(e\) is saturated, i.e. any request to insert a line containing \(e\) into \(P(u_i, v_i)\) will be denied. The crux is now that since both \(P(u_i, v_i)\) and \(P(u, v)\) contain \(e\), the edges \((u, v)\) and \((u_i, v_i)\) are comparable. If \((u, v) \leq (u_i, v_i)\), the edge \((u_i, v_i)\) will appear as the first element of some pair in \(I(u, v)\), in which case one of the tests will be negative, hence causing denial. If now \((u_i, v_i) \leq (u, v)\), the \(c(e)\) sub-lines passing through \(e\) will be represented in \(D(u, v)\), therefore the first test in the series will be negative. A more refined analysis of the algorithm shows that the running time is \(O(\log^3 m)\). No effort has been made to improve this bound, but we believe that it can be substantially lowered.

The previous data structure can be easily adapted to solve the following problem: preprocess a set of \(m\) paths in \(T\) so as to allow for insertions and deletions and so that all paths intersecting a query path can be reported efficiently. This problem is much easier than the one treated in Theorem 3, so it is actually not difficult to achieve a better asymptotic performance. Note that the same problem on linear lists motivated the introduction of segment trees.

6. Mapping Interval Trees

We briefly outline the technique used to map the interval tree (a data structure discovered independently by Edelsbrunner [E] and McCreight [Mc]). Let \(S = \{P_1, \ldots, P_m\}\) be a set of \(m\) paths in \(T\). To simplify the notation, assume wlog that \(m = \Omega(n)\), with \(n = |T|\). Given a query edge \(e \in T\), count the number of paths in \(S\) that pass through \(e\). Using Lemma 3, we easily derive an \(O(m \log m)\) space, \(O(\log m)\) query time solution by formulating the problem as a 2d range search problem. One can do better, however, by mapping the notion of interval trees via a centroid-based decomposition. Borrowing notation from Lemma 18, keep a pointer to \(P_i\) in node \(v(P_i)\) of \(T\) \((1 \leq i \leq m)\). Let \(T_1, T_2\) be the two subtrees of \(T\) associated with node \(v \in T\). Keep two sorted lists of each path stored at \(v\) one with the labels of the end-vertices in \(T_1\) (resp. \(T_2\)). Using Lemmas 3, 15, it is now easy to answer a query. Start a binary search in \(T\) relative to edge \(e\), and for each node \(v\) visited, retrieve all paths containing \(e\) and stored at \(v\) by performing a 1d range search in one of the lists sorted at \(v\). The space required is \(O(m)\) and the query time is \(O(\log^2 m)\). As it turns out, a recent technique [CG] can be added to the algorithm to cut down the query time to \(O(\log m)\) (details omitted).

7. A Few Closing Words

It appears from our previous discussion that in a number of cases computing over free trees is no more difficult than computing over linear lists. In the arithmetic model, we have established a strict equivalence between the interval query problem and its generalization on a tree structure. In the reference machine model, the concept of efficiency is traditionally captured by the class of retrieval problems which can be solved in \(O(n \log \log n)\) space and \(O(\log \log n)\) time. We have shown that in a number of examples this class is closed under transformations from lists to trees. Characterizing the set of problems and techniques for which this holds is an interesting open problem. Also, is there some way to extend the mappings of this paper to arbitrary graphs? Perhaps planar graphs, which have a nice separator theorem, are good candidates. Dealing with the multiplicity of paths between two vertices might be done by introducing optimization criteria (e.g. shortest paths) or treating paths as regular expressions.

REFERENCES


Figure 1: the virtual tree $T$

This work was supported in part by NSF grant MCS 83-03025 and the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786.