Finding an Optimal Tree Searching Strategy in Linear Time

Shay Mozes
Brown University
shay@cs.brown.edu

Krzysztof Onak*
MIT, CSAIL
konak@mit.edu

Oren Weimann
MIT, CSAIL
oweimann@mit.edu

Abstract

We address the extension of the binary search technique from sorted arrays and totally ordered sets to trees and tree-like partially ordered sets. As in the sorted array case, the goal is to minimize the number of queries required to find a target element in the worst case. However, while the optimal strategy for searching an array is straightforward (always query the middle element), the optimal strategy for searching a tree is dependent on the tree's structure and is harder to compute. We present an $O(n)$-time algorithm that finds the optimal strategy for binary searching a tree, improving the previous best $O(n^3)$-time algorithm. The significant improvement is due to a novel approach for computing subproblems, as well as a method for reusing parts of already computed subproblems, and a linear-time transformation from a solution in the form of an edge-weighed tree into a solution in the form of a decision tree.

1 Introduction

The binary search technique is a fundamental method for finding an element in a sorted array or a totally ordered set. If we view the sorted elements as a line of vertices connected by edges, then searching for the target element is done by querying edges such that a query on edge e tells us which endpoint of e is closer to the target. It is well known that in such a search, the optimal way to minimize the number of queries in the worst case is to perform a binary search (see Knuth's book [10]). This technique repeatedly queries the middle edge of the searched segment and eliminates half of the segment from further consideration. Binary search can therefore be described by a complete decision tree where every decision node corresponds to a query on some edge and has degree two, associated with the two possible outcomes of the query.

The problem of locating an element in a sorted array naturally generalizes to the problem of locating a vertex in a tree [2, 11, 15]. Again, we are allowed to query an edge to find out which of each endpoints is closer to the required vertex. Another natural generalization is searching in partially ordered sets (posets) rather than totally ordered sets [2, 3, 12, 13]. When searching a poset for a target element x, the queries are of the form "$x \leq y$?" for some member y of the poset. A negative answer to a query means that either $x > y$ or that x and y are incomparable. These two generalizations are equivalent when the partial order can be described by a forest-like diagram.

Both search problems can be formalized as follows. *Given a tree (or a partially ordered set), construct a decision tree of the lowest possible height that enables the discovery of every target element. A decision node corresponds to a query and has degree two associated with the two possible outcomes of the query.* Unlike searching in a sorted array or a totally ordered set, the optimal decision tree is now not necessarily complete and depends on the structure of the input tree (or the partial order diagram). This is illustrated in Figure 1 for the case of searching a tree. A searching strategy based on this decision tree is called the *optimal strategy* and is guaranteed to minimize the number of queries in the worst case.

Carmo *et al.* [3] showed that finding an optimal strategy for searching in general posets is NP-hard, and gave an approximation algorithm for random posets. For trees and forest-like posets, however, an optimal strategy can be computed in polynomial time as was first shown by Ben-Asher, Farchi, and Newman [2]. Ben-Asher *et al.* gave an $O(n^4 \log^3 n)$-time algorithm that finds an optimal strategy. This was recently improved to $O(n^3)$ by Onak and Parys [15] who introduced a general machinery for bottom-up constructions of optimal strategies. Laber and Nogueira [11] gave an $O(n \log n)$-time algorithm that produces an additive $\lg n$-approximation. This yields a 2-multiplicative approximation, since the depth of a valid decision tree is always at least $\lg n$.

Our Results. In this paper, we follow [2, 11, 15] and focus on trees and forest-like posets. That is, we are interested in computing the optimal strategy for searching a tree where querying an edge tells us which endpoint of the edge is closer to the target. We present a worst-case $O(n)$-time algorithm for this problem, im-

*Supported in part by NSF grant 0514771. Part of the research was done during a summer internship with Google.

proving the previous best $O(n^3)$-time algorithm. Our result requires a novel approach for computing subproblems in the bottom-up framework of [15]. In addition to proving the correctness of this approach, we introduce two new ideas that are crucial for obtaining a linear-time algorithm. The first is a method for reusing parts of already computed subproblems, and the second is a linear-time transformation from an edge-weighed tree into a decision tree. Our result improves the running time of algorithms for searching in forest-like partial orders (cf. [15]) as well.

Applications. One practical application of our problem is file system synchronization. Suppose we have two copies of a file system on two remote servers and we wish to minimize the communication between them in order to locate a directory or a file at which they differ. Such a scenario occurs when a file system or database is sent over a network, or after a temporary loss of connection. The two servers can compare directory or file checksums to test whether two directories or files differ. Such a checksum test can detect if the fault is in a subdirectory or in a parent directory. Directories are normally structured as rooted trees and a checksum test on a rooted subtree is equivalent to an edge query on an unrooted subtree. This paper assumes edge queries on unrooted trees but this is equivalent to subtree queries on rooted trees.

Software testing or "bug detection" is another motivation for studying search problems in posets (and in particular in trees). Consider the problem of locating a buggy module in a program where dependencies between modules constitute a tree. For each module we have a set of exhaustive tests that verify correct behavior of the module. Such tests can check, for instance, whether all branches and statements in a given module work properly. Minimizing the worst-case number of modules that we test in order to locate the buggy module reduces to our searching problem.

Related research. Many other extensions of binary search are reported in the literature. These include querying vertices rather than edges [15], Fibonaccian search [8], interpolation search [16], searching when query costs are non-uniform [4, 9, 14], and searching an order ideal in a poset [12, 13]. Some other fundamental algorithmic problems for posets that have been studied include sorting, selection, computing a linear extension, and computing the heights of all elements [5, 7].

2 Preliminaries — Machinery for Solving Tree Searching Problems

In this section we review the techniques required for a bottom-up construction of the optimal strategy as introduced by Onak and Parys [15].

2.1 Strategy functions. Recall that given a tree $T = (V, E)$, our goal is to find an *optimal strategy* for searching in T. This strategy should minimize the worst-case number of queries required to locate a target vertex, or equivalently, correspond to a correct decision tree of the lowest possible height. Onak and Parys showed that finding an optimal strategy is equivalent to finding an *optimal strategy function*. A *strategy function* $f\colon E \to \mathbb{Z}_+$ is a function from the set of edges into the set of positive integers that satisfies the following condition. If f takes the same value on two different edges e_1 and e_2, then on the simple path from e_1 to e_2, there is an edge e_3 on which the function takes a greater value (i.e. $f(e_3) > f(e_1) = f(e_2)$). An *optimal* strategy function is one with the lowest possible maximum. We make use of the following lemma.

Lemma 2.1. ([15]) *For every tree T, the worst-case number of queries in an optimal searching strategy in T equals the lowest possible maximum of a strategy function on T.*

The intuition behind strategy functions is that if $f(e) = k$, then when we query the edge e we have at most k more queries until we find the target vertex. It turns out that a strategy function f with maximum k easily transforms into a searching strategy with at most k queries in the worst-case. The first query in the strategy being constructed is about the edge with the maximal value k. If we remove this edge, the tree breaks into two subtrees and the problem reduces to finding the target vertex in one of the subtrees, say T'. The second query is about the edge with maximal value in T' and we continue recursively. By definition of the strategy function f, in the ith query there is a unique edge with maximal value. An example of a strategy function and the corresponding search strategy is illustrated in Figure 1. In section 5 we present an $O(n)$-time algorithm that transforms a strategy function into a decision tree.

2.2 Bottom-up computation. Our objective is thus to find an optimal strategy function f. To do so, we arbitrarily root the tree T, and compute $f(e)$ for every $e \in E$ in a bottom-up fashion. More formally, suppose we have a node u with children $u_1, \ldots, u_k$ connected to u by the edges $e_1, \ldots, e_k$. Assuming that f has already been computed for all $T(u_1), \ldots, T(u_k)$ (where $T(u)$ is the subtree rooted at u), we extend the function to $T(u)$ by computing $f(e_1), \ldots, f(e_k)$ without changing $f(e)$ for any $e \notin \{e_1, \ldots, e_k\}$. This means that the restriction to $T(u_i)$ of our desired optimal strategy function for searching $T(u)$ is optimal for searching $T(u_i)$ for every $1 \le i \le k$.

To describe this extension we need the notion of *visibility*. We say that an edge e is *visible* from a vertex

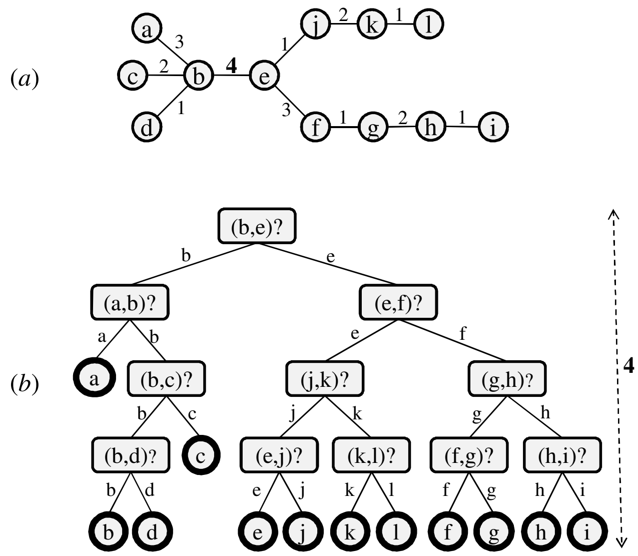

Figure 1: (a) a sample input tree with its optimal strategy function. (b) the corresponding optimal decision tree. The height of the decision tree is equal to the maximum value assigned by the strategy function

u, if on the simple path from u ending with the edge e there is no edge e' such that $f(e') > f(e)$. In other words, the visible values from u are those which are not "screened" by greater values of f. Note that by the definition of a strategy function, each value of f is visible from u at most once. The enumeration in descending order of all values visible from u in $T(u)$ is called the *visibility sequence* of u.

Note that in order to extend the strategy functions on $T(u_1), T(u_2), \ldots T(u_k)$ to a correct strategy function on the entire $T(u)$, it suffices to know just the visibility sequence of each u_i. Denote by s_i the visibility sequence at u_i in $T(u_i)$. We want to assign values to the edges e_1 to e_k so that we achieve a correct strategy function on $T(u)$. We need to make sure that:

- For every two edges $e_i \neq e_j$, $f(e_i) \neq f(e_j)$.
- For every edge e_i, $f(e_i)$ is not present in s_i.
- For every edge e_i, if $f(e_i)$ is present in s_j, then $f(e_j) > f(e_i)$.
- If the same value v appears in two visibility sequences s_i and s_j, where $i \neq j$, then the maximum of $f(e_i)$ and $f(e_j)$ is greater than v.

One can easily verify that these conditions suffice to obtain a valid extension of the strategy functions on $T(u_1), \ldots, T(u_k)$ to a strategy function on $T(u)$.

Consider a lexicographical order on visibility sequences. A valid extension that yields the smallest visibility sequence at u is called a *minimizing extension*. An extension is called *monotone* if increasing the visibility sequences at the children does not decrease the visibility sequence which the extension computes for their parent. Onak and Parys proved that extensions that are both minimizing and monotone accumulate to an optimal strategy function. They further showed that, for the tree searching problem being considered, every minimizing extension is also monotone.

Lemma 2.2. ([15]) *The bottom up approach yields an optimal strategy if at every node we compute a minimizing extension.*

3 Computing a Minimizing Extension

In this section we describe a novel algorithm for computing a minimizing extension. An efficient implementation of this algorithm, presented in Section 4, yields an $O(n)$-time algorithm for the computation of an optimal strategy function.

3.1 Algorithm Description. We first describe the intuition behind the algorithm, and introduce helpful notions. Along the explanation we refer to the relevant line numbers in the pseudocode of the algorithm, which is given in Figure 2.

Consider a vertex u with k children $u_1, u_2, \ldots, u_k$, connected to u along edges $e_1, e_2, \ldots, e_k$ respectively. Let $S = \{s_1, s_2, \ldots, s_k\}$ be the set of already computed visibility sequences at the children. Consider the largest value that appears in more than one of the visibility sequences. Denote this value v (Line 6). In a sense, v is the most problematic value, since any valid assignment of values to the edges must assign some value $w > v$, to one of the edges corresponding to the visibility sequences in which v appears.

What would be a good value for w? We say that a positive value is *free* if it is not visible from u. The set of free values changes as we modify the values assigned to e_1, e_2, $\ldots$, e_k during the execution of the algorithm. Obviously, choosing a value which is not free for w will not result in a valid visibility sequence. Our algorithm therefore chooses the smallest free value greater than v as w (Line 7). But to which edge should w be assigned?

If we assign w to an edge e_i, all values in s_i that are smaller than w become "hidden" from all the edges not in $T(u_i)$. In other words, these values, which were not free until w was assigned (they appeared in s_i), may now become free, and will not contribute to the resulting visibility sequence at u. This means that we should assign w to such an edge so that the greatest possible values will be freed. In other words, we should assign w to an edge whose visibility sequence contains

1: let all $g_i = 0$
2: let all entries of the array U contain the value **free**
3: add the value 0 to every s_i
4: $v \leftarrow$ largest value in all s_i
5: while not all edges assigned a positive value:
6:     if v is exposed at least twice or $v = 0$:
7:         $w \leftarrow$ smallest i s.t. $i > v$ and $U[i]$ is **free**
8:         $T \leftarrow \{i \in \{1, \ldots, k\} : s_i$ contains an exposed value smaller than $w\}$
9:         $j \leftarrow$ any $i \in T$ such that $s_i \geq_w s_{i'}$ for every $i' \in T$
10:         $U[w] \leftarrow$ **taken**
11:         for all values w' in s_j such that w' is exposed and $v < w' < w$:
12:             $U[w'] \leftarrow$ **free**
13:         $g_j \leftarrow w$
14:     else
15:         $U[v] \leftarrow$ **taken**
16:         $v \leftarrow$ largest exposed value in S smaller than v

Figure 2: Algorithm for computing a minimizing extension

the greatest elements smaller than w (Lines 8–9,13). We call such an edge *maximal with respect to* w, a notion that is formalized in Definition 3.2. It turns out that it is worthwhile to assign w to the maximal edge with respect to w regardless of whether this edge contains an occurrence of the multiple value v we originally wanted to take care of.

Once w is assigned to e_i, we only care about values in s_i that are greater than w. We refer to these values as *exposed* values. The values in s_i that are smaller than w no longer affect the visibility sequence in u (Lines 11-12). We then repeat the same process for the largest value currently exposed more than once in S (Lines 5,16). Note that this value may still be v, or some other value smaller than v. The process is repeated until no value is exposed multiple times. Note that during this process we may decide to assign a greater value to an edge e_i that was previously assigned a smaller value. However, as we will see, this procedure never decreases the values we assign to the edges e_i. It is important to emphasize that the only values assigned by the extension algorithm are to the edges e_i, connecting u to its children u_i. We never change values of edges in the subtrees $T(u_i)$. Once all values are exposed at most once, we have to assign values to any edges that were not yet assigned. This is done by assigning the smallest free value according to the same considerations described above. In Section 3.2 we prove that the values assigned to the edges at the end of this process constitute a valid minimizing extension.
We now formally state the necessary definitions.

Definition 3.1. ($s_i \geq s_j$) *The $\geq$ relation denotes lexicographic order on visibility sequences. We define $s_i > s_j$ analogously.*

Definition 3.2. ($s_i \geq_v s_j$) *We say that s_i is larger than s_j with respect to v, denoted $s_i \geq_v s_j$, if after removing all values greater than v from both visibility sequences, s_i is not lexicographically smaller than s_j. We define the relations $>_v$ and $=_v$ analogously.*

For example, if $s_1 = \{5, 1\}$ and $s_2 = \{3\}$ then $s_1 >_6 s_2$, $s_1 >_2 s_2$, but $s_2 >_4 s_1$.

Throughout the execution of the algorithm, we maintain values g_i, which keep track of the values assigned to the edges. Eventually, at the end of the execution, these values describe our extension. Below we define two different kinds of values.

Definition 3.3. (Exposed values) *Let v be a value in a visibility sequence s_i. During execution, we say that v is* exposed *in s_i if v is greater than or equal to the current value of g_i. We define exposed(s_i) to be the set of exposed values in s_i.*

Definition 3.4. (Free values) *A positive value v is free at a given time during execution if at that time it is neither exposed in any visibility sequence s_i, nor equals any g_i.*

We keep track of the free values using the array U. Recall that the algorithm in the form of pseudocode is given in Figure 2.

3.2 Proof of Correctness. Let us start with a simple observation about the algorithm (the formal proof is given in the full version of this paper).

Observation 3.5. *The algorithm has the properties:*

1. *The values g_j never decrease.*

2. *If $g_j > 0$ then g_j is greater than v's current value.*

3. *If $g_j > 0$ then g_j is not exposed in any of the input sequences.*

4. *The current v in the algorithm is always greater than or equal to the current largest value exposed at least twice.*

5. *If $g_j > 0$ then there are no values greater than g_j which are exposed more than once.*

To be able to describe the state of the algorithm at intermediate stages, we introduce a slightly modified problem. This problem captures the fact that some edges have already been assigned a value, and that our algorithm will never decrease this value. We will discuss a natural one-to-one correspondence between instances of the modified problem and of the original one.

DEFINITION 3.6. (MODIFIED PROBLEM)

Given a set of k sequences $\bar{S} = \{\bar{s}_1, \ldots, \bar{s}_k\}$, where each sequence $\bar{s}_i \subseteq \{0, 1, \ldots, n\}$, find a sequence of non-negative values $F = f_1, \ldots, f_k$ such that:

1. (*no duplicates within sequence*) $\forall i : f_i \notin \bar{s}_i$

2. (*no duplicates between sequences*) $\forall 1 \leq v \leq n : |\{i : v = f_i\} \cup \{i : v \in \bar{s}_i \text{ and } v > f_i\}| \leq 1$

3. (*no decreasing assignments*) $\forall i : f_i > 0 \Rightarrow f_i > \min(\bar{s}_i)$

The modified problem can be used to describe intermediate situations, where some of the edges e_i were already assigned a value.

DEFINITION 3.7. ($\bar{S}(S, G)$) *Let S be the original set of visibility sequences at the children and $G = g_1, \ldots, g_k$ be the values assigned to the edges so far ($g_i = 0$ if e_i was not assigned any value). We define the modified problem $\bar{S}(S, G)$ associated with S and G as the set of modified input sequences $\bar{s}_i(s_i, g_i) \stackrel{\text{def}}{=} \{g_i\} \cup \text{exposed}(s_i)$.*

Obviously, g_i is the smallest element in $\bar{s}_i(s_i, g_i)$. If $g_i = 0$, then e_i must be assigned a positive value by f_i (the first condition in definition 3.6). If $g_i > 0$, then we do not have to assign a value to e_i, so f_i may be zero. We may, however, increase the assigned value by choosing $f_i > g_i$ if we wish to (the last condition in definition 3.6).

DEFINITION 3.8. ($Q(S, F)$) *Let $Q(S, F)$ be the set of visible values at the root for a valid solution F of the original problem S.*

DEFINITION 3.9. ($\bar{Q}(\bar{S}, F)$) *Let $\bar{Q}(\bar{S}, F) \stackrel{\text{def}}{=} \cup_i \bar{q}_i(\bar{s}_i, f_i)$ be the set of visible values at the root u for a valid solution F of the modified problem $\bar{S}$. $\bar{q}_i(\bar{s}_i, f_i) \stackrel{\text{def}}{=} \{f_i \text{ if } f_i > 0\} \cup \{v : v \in \bar{s}_i \text{ and } v > f_i\}$.*

In other words, $\bar{q}_i(\bar{s}_i, f_i)$ is the set consisting of $\max(g_i, f_i)$ and all values in $\bar{s}_i$ greater than $\max(g_i, f_i)$. Note that the validity of the solution F assures that the $\bar{q}_i$'s are disjoint sets.

The correspondence between intermediate situations of the original problem and inputs to the modified problem leads to a correspondence between valid assignments of the two. The formal proof of this observation is omitted for brevity.

OBSERVATION 3.10. *Let $G = g_1, \ldots, g_k$ be an intermediate assignment in the execution of the algorithm for input visibility sequences $S = s_1, \ldots, s_k$. Let $\bar{S} = \bar{S}(S, G)$. The minimal $Q(S, F)$, where F ranges over valid solutions to S and such that for each i, $f_i \geq g_i$, equals the minimal $\bar{Q}(\bar{S}, F')$, where F' ranges over valid solutions of $\bar{S}$.*

We now proceed with the proof of correctness. Consider the running of our algorithm and let w be as in Line 7 of our algorithm. That is, if v is the maximal element which is exposed at least twice in S, then w is the smallest free element greater than v. Also, let G denote the current values assigned to the edges (G is the set of values assigned when Line 13 was executed in the previous loops).

LEMMA 3.11. *In any valid assignment F for $\bar{S}(S, G)$, there is a value $z \in F$ such that $z \geq w$.*

Proof. Let F be a valid assignment for $\bar{S}$. Assume, contrary to fact, that $w' < w$ is the largest value in F. If $w' \leq v$, then v appears twice in $\bar{Q}(\bar{S}, F)$ in contradiction to the validity of F (the second condition in Definition 3.6 is violated). Otherwise, $w' > v$. The fact that w is the smallest free element greater than v implies that $w' \in \bar{s}_i$ for some i. If $f_i = w'$, then the first condition in definition 3.6 is violated. Otherwise, $f_i < w'$, so the second condition in Definition 3.6 is violated. ■

We next prove the main technical lemma which essentially shows that the way our algorithm assigns values to edges does not eliminate all optimal solutions.

LEMMA 3.12. *Let $G = g_1, \ldots, g_k$ be an intermediate assignment in execution of the algorithm for input visibility sequences $S = s_1, \ldots, s_k$. Furthermore, let s_j be as in Line 9 of the algorithm. That is, s_j contains an exposed value smaller than w, and $s_j \geq_w s_i$, for any s_i that has an exposed value smaller than w.*

There is an optimal solution F to the problem $\bar{S} \stackrel{\text{def}}{=} \bar{S}(S, G)$ such that $f_j \geq w$.

Proof. Let F' be an optimal solution to $\bar{S}(S, G)$. If $f'_j \geq w$, we are done. Otherwise, $f'_j < w$.

Case 1: If $w \notin F'$, then consider the smallest w' in F' such that $w' > w$. Such w' must exist by Lemma 3.11. Let i be such that $f'_i = w'$. We know the following:

1. $g_i < f'_i = w'$ (by Definition 3.6, third condition).

2. F' does not contain any values in the range $\{w, w+1, \ldots, w'-1\}$ (by definition of w').

3. No value greater than or equal to w appears in $\bar{S}$ more than once (by definition of w).

4. The value w does not appear in $\bar{S}$ and F'.

Consider the assignment F with $f_i \stackrel{\text{def}}{=} \{w \text{ if } w > g_i, 0 \text{ otherwise}\}$ and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ. By the above properties, F is a valid assignment with $\bar{Q}(\bar{S}, F) < \bar{Q}(\bar{S}, F')$ in contradiction to the optimality of F'.

Case 2: If $w \in F'$, then let i be such that $f'_i = w$.

Case 2.1: If $\bar{s}_i$ does not contain any values smaller than w, then g_i must be greater than w. We have $g_i > w = f'_i > 0$, in contradiction to the validity of F' (the third condition in Definition 3.6 is violated).

Case 2.2: If $\bar{s}_i$ contains just a single value smaller than w, then this value must be g_i.

- If $g_i = 0$, then we may exchange the values assigned to e_i and e_j. The desired assignment F is therefore: $f_j \stackrel{\text{def}}{=} f'_i = w$, $f_i \stackrel{\text{def}}{=} \max\{f'_j, g_j\}$, and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ.

- If $g_i > 0$, then there was no need to increase the value assigned to e_i from g_i to w. In particular, g_i must equal f'_m, for some m. Otherwise, by setting f'_i to 0 in F', we would get a valid solution better than F'. To be able to assign g_i to e_i, we must assign a different value to e_m. The assignment F'' with $f''_m \stackrel{\text{def}}{=} f'_i = w$, $f''_i \stackrel{\text{def}}{=} 0$, and $f''_\ell \stackrel{\text{def}}{=} f'_\ell$, for all other values of ℓ, is a valid assignment with $\bar{Q}(\bar{S}, F'') = \bar{Q}(\bar{S}, F')$. We may repeat the entire proof with F'' in the place of F'. The fact that $g_m < f'_m = g_i$ assures us that we will not repeat entering this case indefinitely.

Case 2.3: If $\bar{s}_i$ contains more than one element smaller than w, then *exposed*(s_i) is not empty, so $i \in T$ in Line 8 of the algorithm.

- If *exposed*$(s_j) =_w$ *exposed*(s_i), then by property 5 in Observation 3.5, $g_i = g_j = 0$. We may therefore exchange f'_j and f'_i, and we are done.

- Otherwise, *exposed*$(s_j) >_w$ *exposed*(s_i). To see that, consider m_i and m_j, the largest exposed values smaller than w in s_i and s_j respectively. Since $s_j \geq_w s_i$, we get that $m_j \geq m_i$. If $m_j = m_i$, then by property 5 in Observation 3.5, $g_i = g_j = 0$, so $s_i =$ *exposed*$(s_i) \neq_w$ *exposed*$(s_j) = s_j$. Therefore, *exposed*$(s_j) >_w$ *exposed*(s_i). Let x be the largest value smaller than w that is exposed in s_j but not in s_i. Consider the assignment F with $f_j \stackrel{\text{def}}{=} f'_i = w$, $f_i \stackrel{\text{def}}{=} \max\{f'_j, x\}$ and $f_\ell \stackrel{\text{def}}{=} f'_\ell$ for all other values of ℓ. $\bar{Q}(\bar{S}, F)$ is not larger than $\bar{Q}(\bar{S}, F')$, so F is an optimal solution with $f_j = w$. F is valid because $\max\{f'_j, x\}$ is not exposed in s_i. x is not exposed in s_i by definition, and if $f'_j > x$ then f'_j cannot be exposed in s_j since *exposed*$(s_j) >_w$ *exposed*(s_i). ■

Theorem 3.13. *Our algorithm finds an optimal assignment in finite time.*

Proof. We will show that there is an optimal assignment $F = f_1, \ldots, f_k$, such that throughout the execution of our algorithm, $\forall i : g_i \leq f_i$, where g_i are the values assigned to the edges in Line 13 of our algorithm.

We proceed by induction on t, the number of times Line 13 has been executed. For $t = 0$, $g_i = 0$ for all i, so the claim trivially holds. Assume that the claim is true for $t-1$ and let $G = \{g_1, \ldots, g_k\}$ be the values assigned to the edges just before the t-th time Line 13 was executed. On the t-th time we execute Line 13, g_j will be increased by setting it to w, where w, j are as in the conditions of Lemma 3.11 and Lemma 3.12. Applying Lemma 3.12 with G shows that there exists an optimal solution F which assigns $f_j \geq w$, as required to prove the inductive step.

Since the algorithm keeps increasing the assigned values g_i, and since the sum of g_i's in an optimal solution is bounded, the algorithm will eventually terminate. ■

4 Optimal Strategy Function in Linear Time

In this section we show that the algorithm for computing a minimizing extension can be efficiently implemented, so that the total time spent on the bottom-up computation of an optimal strategy function is linear in the number of nodes. The proof is composed of two parts. First we bound the amount of time required for the computation of a single extension. Next, we use this to bound the total time required to find the optimal strategy function.

One might be tempted to think that the sum of lengths of all visibility sequences at the children of a node v is a lower bound on the time required to compute a minimizing extension and the resulting visibility sequence at v. This, in fact, is not true. An important observation is that in many cases, the largest values of the largest visibility sequence at the children of v appear unchanged as the largest values in the visibility sequence at v itself. By using linked-lists we reuse these values when computing an extension without ever reading or modifying them.

To state this idea accurately, we define the quantities $k(v), q(v)$ and $t(v)$ at each node v in the rooted tree as follows.

- $k(v)$ is the number of children of v.
- Let S denote the set of visibility sequences at the children of v, and let s_1 be the largest sequence in S. We define $q(v)$ as the sum of the largest values in each sequence over all input sequences in S except s_1. If a sequence is empty, then we say that its largest value is 0. If S is empty, $q(v) = 0$.
- Let s be the visibility sequence at v. We define $t(v)$ to be the largest value that appears in s but does not appear in s_1. If S is empty, $t(v) = 0$.

Lemma 4.1 bounds the time required to compute a minimizing extension and the resulting visibility sequence. The proof constructively describes how to implement each line of the algorithm. The important points in the proof are that we never read or modify any values greater than $t(v)$ in s_1, and that by using the appropriate data structures, we are able to find the desired sequence in Lines 8–9 of the algorithm efficiently.

LEMMA 4.1. *A minimizing extension and its resulting visibility sequence can be computed in $O(k(v) + q(v) + t(v))$ time for each node v.*

Proof. We keep visibility sequences as doubly-linked lists starting from the smallest value to the largest. This allows us to reuse the largest values of the largest input sequence.

We assume for simplicity that there are at least two input sequences. If there is no input sequence, then v is a leaf, and the corresponding visibility sequence is empty, so we can compute it in constant time. If there is just one input sequence, then we should assign the smallest positive value which does not appear in the input sequence to the edge leading to the only child of v. We can find this value in $O(t(v))$ time by going over the list representing the input sequence. We then create a new visibility sequence that starts with this value. The rest of the list is the same as the portion of input visibility sequence above $t(v)$.

Assume, without loss of generality, that $s_1, s_2, \ldots$ are ordered in descending order of the largest value in each sequence. Let l_1, l_2 be the largest values in s_1, s_2 respectively. We say that the largest value of an empty input sequence is zero. Note that we can compute l_2 in $O(k(v)+q(v))$ time by simultaneously traversing all the lists representing the input sequences until all but one are exhausted.

We modify the algorithm so that instead of starting from $v = l_1$ in Line 4, we start from v equal l_2. Clearly, there are no values greater than l_2 that appear in more than one sequence, so the only difference this modification introduces is that all the values between l_2 and l_1 that belong to s_1 would have been marked by the original algorithm as taken in the vector U (Line 15), but are not marked so after this modification. We will take care of such values when we discuss the data structure that represents the vector U below.

Representing the vector U. The vector U is used in the description of our algorithm to keep track of free and taken values. It is modified or accessed in Lines 2, 7, 12 and 15. We maintain the vector U using a stack. Values on the stack are free values. The stack always keeps the following invariant: values on the stack are ordered from the largest at the bottom to the smallest at the top. We do not mark all elements as free as in Line 2. Instead, the stack is initially empty, and if, when we decrease v in Line 16, we encounter a value that is not exposed at all, we insert this value into the stack. When implementing the loop in Lines 11–12, we insert to the stack all values greater than v that are currently exposed, and are about to become free. Since all of them are smaller than the current w, and as we will see below, all values on the stack are greater than the current w, we may insert them in decreasing order to the stack and maintain the stack invariant. We now describe how to find the value w in Line 7. If the stack is not empty, we pop the value from the top of the stack and use it as w since this is the smallest free value greater than v. If the stack is empty, then we must find a free value greater than l_2 (remember that such values were not inserted into the stack because we changed Line 4 to start from $v = l_2$). In this case, we traverse the list representing s_1 to find the smallest value that does not appear in it. In total, we will not spend more than $O(t(v))$ time on traversing s_1 in order to find such values.

The next two data structures we describe are used to efficiently find the correct s_j in Lines 8–9.

Lists of sorted sequences. For each i between 0 and l_2, we maintain a list L_i of the visibility sequences sorted in descending order according to $\leq_i$. Moreover, the following invariant always holds: a sequence s_j appears in L_i if the greatest value in s_j is at least i, and there is an exposed value in s_j that is not greater than i.

Initially, we create the lists as follows. We use a modification of radix sort. L_0 is simply the list of all sequences in an arbitrary order. To create L_i, we take L_{i-1} and create two lists, L_i^+ and L_i^-. L_i^+ contains all the sequences in L_{i-1} that contain i, and L_i^- contains all the sequences in L_{i-1} that do not contain i, but do contain a value greater than i. In both new lists, sequences occur in the same order as in L_{i-1}. L_i is a concatenation of L_i^+ and L_i^-. By induction, L_i contains sequences in descending order according to $\leq_i$. The total length and total setup time of all the lists L_i is $O(k(v)+q(v))$. Within the same time constraints we keep pointers from each sequence to its occurrences in the lists L_i, which will be used to remove them from the lists in constant time per removal. To maintain the invariant, we have to update the lists if at some point g_j is increased in Line 13. When this happens, we remove s_j from all L_i such that i is smaller than the smallest exposed value in s_j. If the new value of g_j is greater than the largest value in s_j, then we remove s_j from all the lists. Since initially, the total size of the lists L_i is $O(k(v)+q(v))$, we do not spend more than $O(k(v)+q(v))$ time on modifying the lists along the entire computation of the extension.

Counters of long sequences with exposed values. We introduce one more structure. For each i from 0 to l_2, let C_i keep track of the number of sequences whose maximal value is exposed and not smaller than i. We need to update this structure only if we set g_j in Line 13 with a value greater than the maximal exposed value, m, in s_j. If this happens, we subtract 1 from all C_0 to C_m (certainly, if $m > l_2$, we stop at C_{l_2}). Since this happens at most once for each visibility sequence, and by the definition of $k(v)$ and $q(v)$, we spend at most $O(k(v)+q(v))$ time on all such subtractions.

Finding the correct s_j in Line 9. We can finally describe how to find the correct sequence s_j in Lines 8-9. Suppose first that w is at most l_2.

- If L_w is empty, then all the sequences corresponding to indices in T have their largest exposed values smaller than w. To find the correct s_j, we start from C_0 and look for i such that $C_{i+1} = C_w$, but $C_i > C_w$. The first sequence in L_i is the required sequence. This is true since there are no sequences with exposed values between i and w, and all sequences with an exposed value of i are in L_i, and the first sequence in L_i is the largest sequence with an exposed value i with respect to i and therefore also with respect to w. Each sequence can be found this way at most once because once it is found and assigned the value w, it no longer has any exposed values. Therefore, over the entire computation of the extension we pay for this case at most $O(k(v)+q(v))$.
- Suppose now that L_w is not empty. We look at the first, and largest with respect to w, sequence s_* in L_w. It may be the case that there is a greater sequence with respect to w among the ones that have an exposed value smaller than w, but that the maximal value of this sequence is also smaller than w, and therefore, this sequence does not appear in L_w. Let m be the maximal value in s_* that is smaller than w. If $C_m = C_w$, then we did not miss any sequence, and s_* is indeed the required sequence. Note that we found it in $O(1)$ time. If $C_m > C_w$, but $C_{m+1} = C_w$, then m is the largest exposed value smaller than w, and the first sequence in L_m is the required sequence (again we have found the right sequence in constant time). If both C_m and C_{m+1} are greater than C_w, then there exists a sequence that has an exposed value greater than m and smaller than w that is not present in L_w. We find the largest exposed value i smaller than w as in the case of an empty L_w. Note that this value is at the same time the largest value of the first sequence in L_i. As was the case for an empty L_w, the desired sequence is the first one in L_i, and the total time spent on this case is at most $O(k(v)+q(v))$.

Now assume that w is greater than l_2. If s_1, the largest sequence, contains an exposed value smaller than w, then s_1 is the right sequence. We may keep track of the smallest exposed value in s_1 greater than l_2 in no more than $O(t(v))$ time. Then, we can check if this is the case in constant time. If s_1 has no exposed values between l_2 and w, we proceed as in the case of $w \leq l_2$, since it suffices to find the maximal sequence with respect to l_2.

Conclusion. We have already presented fast implementation of most of the steps of our algorithm. The remaining Lines 4, 6, and 16 can be efficiently implemented as follows. First, as we have already mentioned, we initialize v in Line 4 to l_2 instead of l_1. When we look for the next value of v, we simultaneously traverse the lists representing all sequences whose maximal element is at least v, and as v decreases we include into our working set new sequences that become relevant since v is their largest value. For each v we consider, we can check in constant time if there are such new relevant sequences, if we sort the sequences according to their maximum values less than l_2 in $O(k(v)+q(v))$ time at the beginning of the algorithm. The total time spent on decreasing v is at most $O(k(v)+q(v))$. When we find a new v, we count the number of times it is exposed, and we update this counter as some of them are removed. This way we implement the conditional statement in Line 6 efficiently. Finally, the total number of times we increment the value of g_i at all edges is at most $k(v)+q(v)+t(v)$, because each time at least one value in the corresponding sequence becomes unexposed. After we have computed the minimizing extension we create the new visibility sequence at v from the assigned values g_i and the exposed values we encounter while simultaneously traversing all visibility sequences until we reach the smallest value in s_1, the largest visibility sequence, that is greater than l_2 and than any g_i. This takes $O(k(v)+q(v)+t(v))$ time. To this newly created linked list we link the remaining unscanned portion of s_1 in constant time. We have thus shown the total amount of time required to compute the extension and the resulting visibility sequence is thus $O(k(v)+q(v)+t(v))$. ■

THEOREM 4.2. *An optimal strategy function can be computed in linear time.*

Proof. Recall that we arbitrarily root the input tree. For every node v, given the visibility functions and sequences on the subtrees rooted at the children of v, we compute a minimizing extension of them to a function and a visibility sequence at the subtree rooted at v. By Lemma 4.1, this takes $O(k(v)+q(v)+t(v))$ time at each node v. Eventually, according to Lemma 2.2, we get an optimal strategy function. The total computation thus takes

$$O\left(\sum_v k(v)+q(v)+t(v)\right).$$

Obviously, the total number of sequences that are used in the computation of different sequences is $n-1$, that is, $\sum k(v) = n-1$.

We next bound the sums of $q(v)$ and $t(v)$. We first show that $\sum t(v) \leq 2(n-1)+\sum q(v)$, which implies that it suffices to bound $q(v)$. Recall that $t(v)$ is the largest new value that appears in the largest visibility sequence. Let $l_2(v)$ be the maximal value in the second largest sequence, or 0 if there is no such sequence or if it is empty. Obviously, $l_2(v) \leq q(v)$ for every node v. What is the sum of all $t(v)-l_2(v)$? If $t(v)-l_2(v) > 0$, then at some point in our algorithm we assign $t(v)$ to one of the edges. This means that all the values $l_2(v)+1$ to $t(v)-1$ are taken at that moment. Each of them is taken either by a value assigned to one of the new edges or by an exposed value in the largest input sequence. If there are indeed exposed values between $l_2(v)+1$ and $t(v)$ in the largest input sequence, then our algorithm assigns $t(v)$ to the edge corresponding to the largest sequence, so all of these exposed values will not be visible in subsequent computations at the ancestors of v. It follows that each edge contributes to the sum of all $t(v)-l_2(v)$ at most once in each case. Hence,

$$\sum_v t(v) = \sum_v t(v)-l_2(v)+\sum_v l_2(v) \leq 2(n-1)+\sum_v q(v).$$

Now, it suffices to bound the sum of all $q(v)$. We show by induction that the sum of all $q(v)$ in the subtree $T(v)$ rooted at v is at most $n(v)-r(v)-1$, where $n(v)$ is the number of nodes in $T(v)$, and $r(v)$ is the maximum value in the visibility sequence computed for v. If v is a leaf, then $q(v)=0$, $n(v)=1$, and $r(v)=0$, that is, our claim holds. Suppose that v has $k=k(v)$ children, and that the claim holds for each of them. Let $u_1, u_2, \ldots, u_k$ be the children of v. Assume, without loss of generality, that u_1 has the largest visibility sequence among the children of v. This implies that $r(v) \leq r(u_1)+k$, since we can create a valid extension by assigning the values $r(u_1)+1$ to $r(u_1)+k$ to the edges between v and its children. Then,

$$\begin{aligned}\sum_{v' \in T(v)} q(v') &= \sum_{i=2}^{k} r(u_i) + \sum_{i=1}^{k} \sum_{u' \in T(u_i)} q(u') \\ &\leq \sum_{i=2}^{k} r(u_i) + \sum_{i=1}^{k} n(u_i) - r(u_i) - 1 \\ &= (n(v)-1) - r(u_1) - k \leq n(v) - r(v) - 1.\end{aligned}$$

This concludes the inductive proof and implies that $\sum_v q(v) \leq n-1$. Putting everything together we get

$$\sum_v k(v)+q(v)+t(v) \leq (n-1)+(n-1)+3(n-1) = O(n).$$

Thus proving that the time required to compute the optimal strategy function along with the intermediate visibility sequences at all nodes is linear in n. ■

5 From a Strategy Function to a Decision Tree in Linear Time

In this section we show how to construct an optimal decision tree in linear time using an optimal strategy function. We begin with some additional definitions and assumptions. To avoid confusion between the input tree and the decision tree we will refer to the decision tree by the acronym DT. Each node in DT represents an edge in the input tree. We refer to each node in DT by the edge in the input tree which it represents. Let r be the root of the tree. We introduce an additional node r', and connect it to r. We assign the value ∞ to the edge (r', r), and from now on we treat r' as a root of the tree. For an edge e, let *top*(e) denote the end point of e closer to the root of the tree and *bottom*(e) the end point of e farther from the root. The node e in DT will have exactly two children, one for the case when the query about e returns *top*(e) and the other for the case when it returns *bottom*(e).

Initially, the nodes in DT are disconnected. We now describe an algorithm that uses the computed strategy function and visibility sequences to connect the nodes in DT. We assume that along with each value in each visibility sequence we keep a link to the edge to which that value is assigned. Clearly, this can be done within the same time complexity.

5.1 The Algorithm. We processes all edges of the input tree in any order. For each edge e, let s be the visibility sequence at *bottom*(e).
If s contains no values smaller than $f(e)$, then

(1) set *bottom*(e) as the solution in DT when the query about e returns *bottom*(e).

Otherwise, let $v_1 < v_2 \cdots < v_k$ be the values smaller than $f(e)$ in s, and let e_i be the edge v_i is assigned to.

(2a) set node e_k in DT as the solution when the query on e returns *bottom*(e).

(2b) for every $1 \leq i < k$ set the node e_i in DT as the solution when the query on e_{i+1} returns *top*(e_{i+1}).

(2c) set *top*(e_1) to be the solution in DT when the query on e_1 returns *top*(e_1).

Finally, after applying the above procedure to all edges in the input tree, the root of DT is the only child of the DT node (r', r).

5.2 Correctness and Time Complexity. Let e be an arbitrary edge. We prove that in DT, the children of e corresponding to the answers *top*(e) and *bottom*(e) are assigned correctly and exactly once. It is easy to see that the algorithm assigns *bottom*(e) exactly once – when we process edge e. Recall that if the query on e returns *bottom*(e), then the next query should be on the edge with largest value smaller than $f(e)$ which is visible from *bottom*(e) in the subtree rooted at *bottom*(e). This is done in (2a). If there is no such value, there is no next query either, and *bottom*(e) is the solution to the searching problem. This case is handled in (1). Therefore, *bottom*(e) is assigned correctly and exactly once for each e in the input tree.

We now show that *top*(e) is also assigned correctly and exactly once. Let e' be the first edge with value greater than $f(e)$ on the path from *top*(e) to r'. Such an edge always exists since we assigned the value ∞ to the edge (r', r). First notice that *top*(e) is assigned exactly once – when we process edge e'. It therefore only remains to show that *top*(e) is assigned correctly. Recall that when a query about e returns *top*(e), then the next query should be on the edge with greatest value smaller than $f(e)$ that is visible from *top*(e) in the entire tree (viewing the tree as unrooted). Note that this edge is also the edge with greatest value smaller than $f(e)$ that is visible from *bottom*(e') in the subtree rooted at *bottom*(e') (viewing the tree as rooted). If such an edge exists, we make it a child of e in (2b). If there is no such edge, then obviously, when the query about e results in *top*(e), then *top*(e) is the solution to the searching problem. We handle this case in (2c). Therefore, *top*(e) is assigned correctly and exactly once for each e in the input tree. We have thus established the correctness.

To analyze the time complexity, notice that for each edge e, the algorithm runs in time proportional to the number of values in the visibility sequence of *bottom*(e) that are screened by $f(e)$. Since no value can be screened twice, the total runtime is linear.

It should be noted that independently of this contribution, Erik Demaine [6] gave a linear-time algorithm for constructing a decision tree given just the strategy function (we also need the visibility sequences), using ideas from decremental connectivity in trees [1].

References

[1] S. Alstrup and M. Spork. Optimal on-line decremental connectivity in trees. *IPL*, 64(4):161–164, 1997.

[2] Y. Ben-Asher, E. Farchi, and I. Newman. Optimal search in trees. *SIAM Journal on Computing*, 28(6):2090–2102, 1999.

[3] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theoretical Computer Science*, 321(1):41–57, 2004.

[4] M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan, and A. Sahai. Query strategies for priced information. *Journal of Computer and System Sciences*, 64(4):785–819, 2002.

[5] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, and E. Verbin. Sorting and selection in posets. arXiv:0707.1532, 2007.

[6] E. Demaine. Private communication, 2007.

[7] U. Faigle and G. Turán. Sorting and recognition problems for ordered sets. *SIAM Journal on Computing*, 17(1):100–113, 1988.

[8] D. E. Ferguson. Fibonaccian searching. *Communications of the ACM*, 3(12):648, 1960.

[9] W. J. Knight. Search in an ordered array having variable probe cost. *SIAM Journal on Computing*, 17(6):1203–1214, 1988.

[10] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[11] E. Laber and L. T. Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:1–4, 2001.

[12] N. Linial and M. Saks. Every poset has a central element. *Journal of combinatorial theory*, A 40(2):195–210, 1985.

[13] N. Linial and M. Saks. Searching ordered structures. *Journal of algorithms*, 6(1):86–103, 1985.

[14] G. Navarro, R. Baeza-Yates, E. Barbosa, N. Ziviani, and W. Cunto. Binary searching with non-uniform costs and its application to text retrieval. *Algorithmica*, 27(2):145–169, 2000.

[15] K. Onak and P. Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *FOCS*, pages 379–388, 2006.

[16] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.