Optimal Search in Trees *

Yosi Ben-Asher, † Eitan Farchi, ‡ Ilan Newman §

Abstract

It is well known that the optimal solution for searching in a finite total order set is binary search. In binary search we divide the set into two "halves" by querying the middle element and continue the search on the suitable half. What is the equivalent of binary search when the set P is partially ordered? A query in this case is to a point $x \in P$, with two possible answers: 'yes', indicates that the required element is "below" x, or 'no' if the element is not below x. We show that the problem of computing an optimal strategy for search in Posets that are tree-like (or forests) is polynomial in the size of the tree, and requires at most $O(n^4 \log^3 n)$ steps.

Optimal solutions of such search problems are often needed in program testing and debugging, where a given program is represented as a tree and a bug should be found using a minimal set of queries. This type of search is also applicable in searching classified large tree-like data bases (e.g. the Internet).

Keywords: Optimal search, Search in trees, Binary search, Search in graphs, Posets

AMS subject classifications. 68P10

1 Introduction

Binary search is a well known technique used for searching in total order sets. Let $S = \{s_1, \ldots, s_n\}$ be a total ordered set such that $s_i < s_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. Usually, a binary search is used to find out whether a given element s is a member of S. However, a different interpretation can be used, namely, that, one of the elements in S is "buggy" and needs to be located. In a binary search, we query the middle element $s_{\frac{n}{2}}$. If the answer is 'yes', then the bug is in $S' = \{s_1, \ldots s_{\frac{n}{2}}\}$ and we continue the search on S', otherwise we search on the complement of S', namely $\{s_{\frac{n}{2}+1}, \ldots, s_n\}$. It is well known that the binary search is optimal for this case.

In this work we consider the generalization of searching in partially ordered sets (Poset). A query is to a point x in the Poset, with two possible answers: 'yes', indicates that the required element is "below" x (less than or equal to x), or 'no' if the element is not below x. Equivalently, P can be represented as a directed acyclic graph, G, a query is to a node $x \in G$, a 'yes'/'no' answer indicates that the answer is at a node $y \in G$ reachable/not-reachable from x, correspondingly. We are interested in the problem of computing the optimal search algorithm for a given input Poset. The complexity measure here is the number of queries needed for the worst case input (buggy node). We concentrate on Posets that are "tree" (forest) -like (every element except one has one

*Preliminary version in SODA97.

†Dept. of Comp. Sci., Haifa U, Haifa, Israel, yosi@mathcs.haifa.ac.il

‡I.B.M research center ,Haifa, Israel.

§Dept. of Comp. Sci., Haifa U, Haifa, Israel, ilan@mathcs.haifa.ac.il

element that covers it). In this case, the partial order set is represented as a rooted tree T, whose nodes are the elements of S. A query can be made to any node $u \in T$. A 'yes'/'no' answer indicates whether the buggy node is in the subtree rooted at u, or in its complement. Our results extend naturally to 'forest' like Posets too. Some comments are made for Cartesian product Posets.

An example of a (optimal) search on a tree of 5 nodes is shown in figure 1. The arrow points to the next query node. The search takes 3 queries in the worst case.

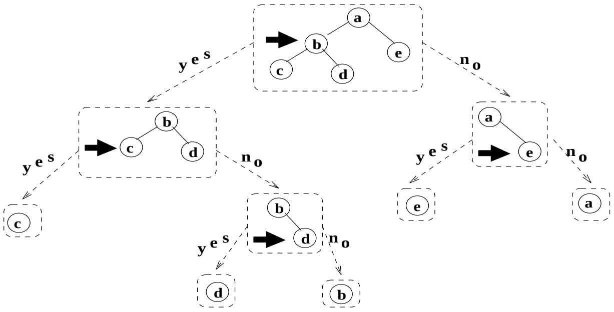

Figure 1: *Searching in a tree.*

One motivation for study search in trees (and Posets in general) is that it forms a generalization of the search in linear orders to more complex sets: The known binary-search is optimal for the path of length n with $\lceil \log n \rceil$ cost. Another extreme example is the search in a completely non-ordered set of size n. This is equivalent to search in a star with n leaves. Here, as we must query each leaf separately, the search takes n queries. Searching in Trees spans a spectrum between these two examples in terms of the cost function as well as the the Poset type.

There is also a practical motivation; consider the situation where a large tree-like data structure is being transfered between two agents. Such a situation occurs when a file system (data base) is sent across a network, or a back/restore operation is done. In such cases, it is easy to verify the total data in each subtree by checksum-like tests (or randomized communication complexity equality testing). Namely, such equality test easily detects that there is a fault but gives no information which node of the tree is corrupted. Using search on the tree (by querying correctness of subtrees) allows us to find the buggy node and avoid retransmitting the whole data structure.

Software testing is another motivation for studying search problems in Posets (and in particular in trees). In general, program testing can be viewed as a two person game, namely the tester and its 'adversary'. The adversary injects a fault into the program and the tester has to find the fault while minimizing the number of tests. A typical scenario in software testing is that the user tests his program by finding a "test bucket" (a set of inputs) that meets a certain coverage criteria, e.g., branch coverage or statement coverage [1, 2, 3]. It is plausible that in certain situations it might be possible to embed such a set of tests (e.g., the union over all test buckets that meet branch coverage) in a Poset or in a Tree such that the requirement for covering all tests can be replaced by a requirement for searching in this Poset or Tree. Finding an optimal search can save a lot of tests as the cost of a search might be considerably smaller than the size of the domain. For example the syntactic structure of a program forms a tree, thus if suitable tests are available, statement coverage might be replaced by a search in the syntactic tree of the program.

Finally, a possible motivation and direct application is in the area of Information retrieval: Consider a 'Yahoo' search like scenario. The Yahoo contains an immense tree that classifies home pages (right now - estimated as about $1-2\%$ of the total number of WWW homepages). In a typical search, a node is reached and it exposes the next level of the tree (or part of it). The user chooses the appropriate branch, according to the query it has in mind. As it turns out, this tree is quite deep which often results in a numerous amount of queries before reaching the target. Clearly, such a top down search might be inefficient compared to the possibility of the optimal search of the Yahoo tree (e.g., searching in a chain of n nodes (a tree of depth n) requires n queries if we search top-down and only $\log n$ queries if we allow to query arbitrary nodes).

A different notion of searching in Posets was considered by Linial and Saks [5, 4]. They consider the case where a set of real numbers are "stored" in a Poset, so that their natural order is consistent with the Poset order. A search in this case is to determine if a real x is stored in the set. The possible queries in this case are, as in our case, the Poset elements. The two possible answers for a query z is either 'yes' (means $x \leq e(z)$ where $e(z)$ is the element stored in z), or 'no' ($x > e(z)$). The first answer results in excluding all elements greater than z from the Poset and the later excludes all elements below z. Note that the difference between the two models is the resulting Poset after a 'yes' answer. It turns out that, in spite of the similarity in definition, the two models are quite different (e.g. the product of two path see sec. 5). Linial and Saks proved lower and upper bounds for the number of queries needed to search in Posets in terms of some of the Poset properties, however, they presented no algorithm to find the exact cost.

Our main result is a polynomial time algorithm (in the size of the tree) that finds the **optimal** search strategy for any tree (forest). Let $T(v)$ be the subtree of T rooted at v. The answer to a query $v \in T$ results in a search on either the subtree rooted at v, or its complement $T - T(v)$. Thus, the optimal complexity, $w(T)$ of the search for T is defined by minimizing over all $v \in T$ the expression $1 + \max\{w(T(v)), w(T - T(v))\}$. Direct use of this formula to compute $w(T)$ would give an exponential time algorithm. Another possible approach is trying to compute $w(T)$ in a bottom-up manner, however, it seems that this too needs exponential time. The reason is that knowing the cost of the subtrees is not enough to compute the cost of the complete tree: A complement subtree produced by querying a node results in a new subtree whose cost has to be computed all over. Our approach is to "get rid" of this difficulty by using further relevant information on subtrees (rather than just the cost of the optimal search for them). We use a somewhat non standard decomposition of a tree into subtrees, so that the cost of a tree can be determined using the relevant information of its subtrees. Next, we generalize the results for Forest, and draw some conclusions for cartesian product Posets.

We note that for bounded degree trees, an approximation of the optimal strategy may be obtained by finding a "splitting" vertex that splits the tree into two parts that are not too big. However, such an approach totally fails for unbounded degree trees.

2 Basic definitions and preliminary Facts

Let us start with some notations: The subtree of T that is rooted at u is denoted by $T(u)$. When it is clear from the context, we identify $T(u)$ with u. Deleting all nodes of a subtree T_1 from a tree T is denoted by $T - T_1$, in particular $T - u = T - T(u)$.

Definition 2.1 *A search algorithm Q_T for a tree T with root r is defined recursively as follows: If $|T| = 1$ the search is trivial and gives as output the only node in T. For $|T| \geq 2$ a search algorithm is a triplet (v, Q_v, Q_{T-v}), where $v \in T - r$ (a 'first query'), Q_v is a search algorithm for $T(v)$ (corresponds to a 'yes' answer), and Q_{T-v} is a search algorithm for $T - v$ ('no' answer).*

Graphically, we denote Q_T as follows:

$$\begin{array}{lccc} Q_T & = & v \xrightarrow{\text{no}} & Q_{T-v} \\ & & \downarrow & \\ & & Q_v & \end{array}$$

The cost of a search algorithm Q, denoted by $w(Q)$, is the number of queries needed to find any buggy node in the worst case. The cost of an optimal search algorithm for T, $w(T)$ is

$$w(T) \ = \ min_{Q_T} \ w(Q_T).$$

An optimal algorithm is any search algorithm Q_T such that $w(Q_T) \ = \ w(T)$. Note that the above definition conforms with the convention that there is always a "buggy node", thus in turn, the cost of a single node is zero since this node must be buggy and no query is needed. The case of search in which there is a possible "un-found" answer is easily obtained from the above definition (with the expense of one additional query). See section 5 for more details.

The following is immediate from the definitions.

Fact 2.1 *For a tree T, with $|T| > 1$, $w(T) \ = \ 1 + \ \min_{v \in T} \max\{w(v), w(T-v)\}$*

A useful property of the cost is that it is monotone:

Observation 2.1 *Let T_1 be a subtree of T_2 (namely T_1 is obtained from T_2 by deleting some nodes), then $w(T_1) \leq w(T_2)$.*

Fact 2.1 suggests that we might as well start the search by querying a node $u \in T$ for which $w(u) < w(T)$ and $w(T-u)$ is minimal. Applying this idea further on, leads us to the definition of the 'sequence of complements'. We start by defining a lexicographic order on finite sequences:

Definition 2.2 *Let $\mu = \ [\mu_1, \mu_2, \ldots \mu_k]$ and $\rho = \ [\rho_1, \rho_2, \ldots \rho_n]$ be two sequences of natural numbers, then μ is lexicographically smaller than ρ ($\mu <_L \rho$) if there exists i such that $\mu_i < \rho_i$ and for every $j < i$, $\mu_j = \rho_j$. In addition we use the term "order L" to denote the set of strictly decreasing finite sequences of non-negative integers with last element being 0, in lexicographic order.*

Definition 2.3 *For a given tree T, the sequence of complements $\mu(T) \ = \ [\mu_0, \mu_1, \mu_2, \ldots, \mu_l]$ is defined recursively as follows:*
$\mu_0 = w(T)$.
If $|T| = 1$ then $l = 0$ and $\mu(T) = [0]$
for $|T| > 1$,

$$[\mu_1, \mu_2, \ldots, \mu_l] = \min\{\mu(T-u)| \ w(T(u)) < w(T)\}$$

where min *is taken according to the lexicographic order.*

For example $\mu(P)$ of a path P with five nodes is $[3, 0]$ as its cost is 3 and the node u that achieves the smallest $\mu(P-u)$ is the son of the root for which $w(T-u) = 0$. The reader can verify that μ of a star with n leaves is $[n, n-1, \ldots, 2, 1, 0]$. The following properties are immediate from the definition:
Let $\mu(T) = [\mu_0, \mu_1, \ldots, \mu_l]$ then

1. $\mu_0 \ > \ \mu_1 \ > \ \ldots \ > \ \mu_l \ = 0.$
2. $l \leq \ w(T).$

As the sequence of complements is always strictly decreasing, we restrict the order L to all *strictly decreasing* finite sequences of non negative integers with last element being 0. The order L is a *discrete total* order on this set of sequences. In particular, every sequence has a rank in this total order, and every sequences has a successor. Thus, it also follows that $\mu(T)$ is a unique, well defined sequence for every tree T.

Definition 2.4 *For T with $|T| \geq 2$, $\lambda_T \in T$ is any node in T for which $\mu(T - \lambda_T)$ is the smallest and $w(T(\lambda_T)) < w(T)$. Namely λ_T is such that $\mu(T) = [w(T), \mu(T - \lambda_T)]$. Note that for $|T| = 1$, λ_T is not defined.*

It is immediate from the definition that λ_T is an optimal node, i.e., there exists an optimal search algorithm whose first query is λ_T. Fig 2 illustrates λ_T, $\mu(T)$ and an optimal search algorithm that starts in λ_T for a particular example. In the above tree (left side) the cost of each subtree is marked by its root. Here $\lambda_T = v$ since $\mu(T - v) = [2, 1, 0]$ compared (for example) to $\mu(T - u) = [3, 0]$.

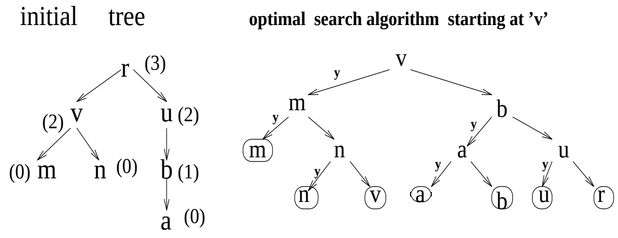

Figure 2:

3 The decomposition of T and the role of $\mu(T)$

As mentioned before it is not obvious how to efficiently compute the cost of T using the formula of Fact 2.1 in a bottom-up manner. Our solution is to decompose T into subtrees, so that we can characterize $\mu(T)$ as a function of the μ's of its subtrees. We define the following operations:

T' **:** "Re-rooting" operation, T' is the tree obtained from T by adding a new node x whose only child is the root of T.

$T'_1 + \ldots + T'_k$ **:** "grouping" operation, $T = T'_1 + T'_2 + \ldots + T'_k$ (also denoted by $\sum_{i=1}^{k} T'_i$) is obtained from disjoint trees $T_1, T_2, \ldots, T_k$ by attaching $T_1, T_2, \ldots, T_k$ as the children of a new root x. Note that we write it as an operation on the *re-rooted* trees $T'_1, \ldots, T'_k$ rather than on $T_1, \ldots, T_k$ as we will relate $\mu(T)$ to $\mu(T'_1), \ldots, \mu(T'_k)$. In the grouping operation we will always order $T_1, T_2, \ldots, T_k$ so that $\mu(T'_1) \leq_L \ldots \leq_L \mu(T'_k)$.

Fact 3.1 *Every rooted tree can be constructed from single nodes using the above operations.*

In the following two sections we will show how $\mu(T)$ is determined by the μ's of the subtrees used to decompose T according to the operations defined above.

3.1 Properties of the re-rooting operation

We relate here the sequence of complements of any tree T to the sequence of complements of the tree T' that is obtained from T by re-rooting. As it will turn out this can simply be characterized by the following Theorem.

Theorem 3.1 *Let T be a rooted tree and let T' be the tree obtained from T by re-rooting, then $\mu(T') = succ(\mu(T))$ where succ is the successor according to the order L on strictly decreasing non negative finite sequences, ending with* 0.

The key ingredient in the proof of the theorem is based on the observation that a "jump" in the coordinates of the sequence $\mu(T)$ is a necessary and sufficient condition for the cost of T' to remain the same. The term "jump" indicates that the difference between two consecutive coordinates of $\mu(T)$ is greater than 1. The intuition is that such a jump can "compensate" for the expected increase in $w(T')$ caused by the extra node added above the root of T. To demonstrate this on an example, consider a path P with three nodes, for which $\mu(P) = [2, 0]$, namely, it contains a jump. Indeed P' is a path with 4 nodes and its cost remains 2. The reason why $w(P')$ is not more than $w(P)$ is that we can start a search in P' by querying λ_P. For a 'yes' answer we proceed as we would have done in P and the cost will certainly not change. For a 'no' answer we are left with $P' - \lambda_P = (P - \lambda_P)'$ but $(P - \lambda_P)$ is a single node with zero cost. Thus $w(P' - \lambda_P) = 1$ which is more than $w(P - \lambda_P)$ but not more than $w(P) - 1$ (thus not enough to increase $w(P')$). Another example is the re-rooting T' of the tree T in Figure 2, showing that the cost does increase if $\mu(T)$ contains no jump. For T of Figure 2 $\lambda_T = v$ and $\mu(T) = [3, 2, 1, 0]$. Indeed, the reader can verify the cost increases, and $w(T') = 4$.

We proceed now with the proof of Theorem 3.1.

Proposition 3.1 *For a given tree T, $w(T) \leq w(T') \leq w(T) + 1$*

Proof: The left inequality is immediate from Observation 2.1. The right inequality follows from the search that starts by querying the (old) root of T. ∎

Proposition 3.2 *If $w(T') = w(T) + 1$ then $\lambda_{T'} = root(T)$ and $\mu(T') = [w(T'), 0]$.*

Proof: Querying $v = root(T)$ leads to $w(T' - v) = 0$ and thus $\mu(T') \leq [w(T'), 0]$, but $[w(T'), 0]$ is the smallest sequence $[\mu_0, \ldots, \mu_l]$ (according to $<_L$) for which $\mu_0 = w(T')$. ∎

Lemma 3.1 *Let $\mu(T) = [\mu_0, \ldots, \mu_l]$, then if there exists $i \geq 0$ such that $\mu_i < w(T) - i$, then $w(T') = w(T)$.*

Proof: Note that the condition of the Lemma just states that there is a "jump" between μ_i and μ_{i-1}. Let $\alpha = w(T)$, we will use induction on $|T|$ to show that there exist a search algorithm Q for T', with $w(Q) = \alpha$. The premise of the Lemma is possible only for a tree with cost greater than 1, hence the induction base includes only a path P with three nodes or a star with two leaves. For the latter, the sequence of complements $\mu()$ is $[2, 1, 0]$ and the premises of the Lemma is not met. For the path P, the sequence of complements is $[2, 0]$, and indeed $w(P') = w(P) = 2$.

Let T be a tree with $w(T) = \alpha \geq 3$, and consider the following search algorithm for T':

$$\begin{array}{rcll} Q & = & \lambda_T \xrightarrow{\text{no}} & Q_1 \\ & & \downarrow & \\ & & Q_2 & \end{array}$$

where Q_1 is an optimal search algorithm for $T' - \lambda_T$ and Q_2 is an optimal search algorithm for $T(\lambda_T)$. Here, $w(Q) \leq 1 + \max(w(Q_2), w(T' - \lambda_T))$, but, by the definition of λ_T, $w(Q_2) \leq \alpha - 1$ and thus $w(Q) \leq \max(\alpha, 1 + w(T' - \lambda_T)) = \max(\alpha, 1 + w((T - \lambda_T)'))$. Hence, it is enough to show that $w((T - \lambda_T)') \leq \alpha - 1$.

Let us first consider the case where the jump is for $i = 1$ namely $\mu_1 = w(T - \lambda_T) \leq \alpha - 2$. By proposition 3.1 for $T - \lambda_T$ we have that $w((T - \lambda_T)') \leq w(T - \lambda_T) + 1 \leq \mu_1 + 1 \leq \alpha - 1$.

Consider the case where $i > 1$. Let $T_1 = T - \lambda_T$ then by the definition of μ we get that $\mu(T_1) = [\mu_1, \dots, \mu_l]$, with $\mu_1 = w(T - \lambda_T) \leq \alpha - 1$. Thus, the premise of the Lemma holds for T_1 (with $i' = i - 1$), so by the induction hypothesis $w(T_1') = w(T_1) \leq \alpha - 1$. ▮

The counter direction of Lemma 3.1 is also true:

Lemma 3.2 *Let $\mu(T) = [\mu_0, \dots, \mu_l]$. If $w(T') = w(T)$, then there exists $i \in \{1, \dots, l\}$ such that $\mu_i < w(T) - i$.*

Proof: Let $w(T) = w$. It is sufficient to show that if the condition is not met, i.e., $\forall i,\ \mu_i \geq w - i$, then $w(T') = w + 1$. We prove this by induction on $|T|$. The base case is a single node whose $\mu = [0]$ and for which T' is a path of length 1 with cost $= 1$.

For $|T| > 1$ the assumption above implies that $\mu(T) = [w, w - 1, w - 2, \dots, 1, 0]$. Let Q be an optimal algorithm for T' starting with a vertex x. We show that Q must spend $w + 1$ queries. If $w(T(x)) \geq w$ then on 'yes' answer, we are left with $T(x)$ on which additional w queries are needed and we are done. Thus we may assume that $w(T(x)) \leq w - 1$. Therefore by the definition of λ_T we get that $\mu(T - x) \geq_L \mu(T - \lambda_T)$. But this means that $\mu(T - x) = \mu(T - \lambda_T) = [w - 1, w - 2, \dots, 1, 0]$ as the successor of $\mu(T - \lambda_T)$ has a first coordinate equal to w. Thus by the induction hypothesis on $T - x$ (whose μ has no 'jump') $w(T' - x) \geq w(T - x) + 1 = w - 1 + 1 = w$. Hence $w(Q) \geq 1 + w(T' - x) = 1 + w$. ▮

The above two lemmas yields that

Theorem 3.2 *Let $\mu(T) = [\mu_0, \dots, \mu_l]$ then $w(T') = w(T)$, iff there exists $i \geq 0$ such that $\mu_i < w(T) - i$.*

We now conclude with the proof of Theorem 3.1.

Proof: If for a tree T there is no 'jump' namely $\mu(T) = [w, w - 1, ..., 1, 0]$ where $w = w(T)$, then by Theorem 3.2 $w(T') = w + 1$. Moreover, by Proposition 3.2, $\mu(T') = [w + 1, 0]$ which is the successor of $[w, w - 1, ..., 1, 0]$ in the order L.

Assume then that there is a 'jump' in $\mu(T)$, and let i be the largest index for which $\mu_{i-1} \geq l - i + 2$, namely

$$\mu(T) = [\mu_0, \mu_1, \dots, \mu_{i-1}, \overbrace{l - i}^{\mu_i}, l - i - 1, \dots, 2, 1, 0] \quad .$$

Then we prove by induction on $w(T)$ that $\mu(T') = succ(\mu(T))$ and that $\lambda_{T'} = \lambda_T$. By Theorem 3.2 $w(T') = w(T) = \mu_0$. Assume $\lambda_{T'} = x$ then $\mu(T') = [\mu_0, \mu(T' - x)]$. But $T' - x = (T - x)'$ and by the induction hypothesis for $(T - x)'$, $\mu(T - x)' = succ(\mu(T - x))$. However by the definition of λ_T,

$$\mu(T - x) \geq \mu(T - \lambda_T) = [\mu_1, ...\mu_{i-1}, l - 1, l - i - 1, ..., 1, 0] \quad .$$

however

$$succ([\mu_1, ...\mu_{i-1}, l - i, l - i - 1, ..., 1, 0]) = [\mu_1, ..., l - i + 1, 0]$$

thus we get that $\mu(T' - x) = succ(\mu(T - x) \geq [\mu_1, ..., l - i + 1, 0]$. On the other hand, by the definition of the node x in T' we have that $\mu(T' - x) \leq \mu(T' - \lambda_T) = \mu((T - \lambda_T)') = succ(\mu(T - \lambda_T)) = [\mu_1, ..., l - i + 1, 0]$. Thus we conclude that

$$\mu(T') = [\mu_0, \mu_1, ..., l - i + 1, 0] = succ([\mu_0, \mu_1, ...\mu_{i-1}, l - i, l - i - 1, ..., 1, 0]) \quad .$$

Moreover, as we get that $\mu(T' - x) = \mu(T' - \lambda_T)$ this also proves that $\lambda_T = \lambda_{T'}$. ▮

3.2 Properties of the grouping operation

We now relate the sequence of complements of grouping k re-rooted trees $T_1' + T_2' + \ldots + T_k'$, to the individual sequences. First a simpler operation is defined as follows:

Definition 3.1 *The "Amalgamation" operation $T_1' + T_2$ on disjoint trees T_1', T_2 is the tree that is obtained by amalgamating the roots of T_1' and T_2 into one node.*

Note that in the amalgamation operation it is required that one of the trees is a re-rooted tree. The amalgamation operation exhibits a certain monotonicity:

Lemma 3.3 *Let T_1, T_2, T be trees, then $\mu(T_1') \leq_L \mu(T_2') \implies w(T_1' + T) \leq w(T_2' + T)$.*

Proof: By induction on the sum $n = |T| + k$ where $|T|$ is the number of nodes in T and k is the rank of $\mu(T_1')$ defined by the total order $<_L$.

The induction base is for $n = 2$, and T that has one node. Since the amalgamation of one node to any tree does not change the tree at all, then the base of the induction is trivially true.

First case $w(T) \leq w(T_1')$: Let $w = w(T_1')$ and assume Q is any search algorithm for $T + T_2'$ with cost $w(Q)$. We give an algorithm for $T + T_1'$ with cost of at most $w(Q)$. Indeed let $u \neq root(T_2')$ be the first query of Q. If $u \in T$ then obviously $w(Q) \geq w + 1$, as $w((T-u) + T_2') \geq w(T_2) \geq w$. However, the algorithm that first queries the root of T_1, then proceeds optimally for T_1 if the answer is 'yes' and optimally for T if the answer is 'no', also achieves $w+1$ and we are done.

If $u \in T_2$, then we may assume that $w(T(u)) \leq w - 1$ (otherwise $w(Q) \geq w + 1$ and we are done as before). We also may assume that $w(T_2') = w$, as if $w(T_2') \geq w + 1$, then $w(Q) \geq w + 1$, but as we have shown, we can clearly search $T + T'$ in $w + 1$ queries. Thus $\mu(T_2' - \lambda_{T_2}) \geq \mu(T_1' - \lambda_{T_1})$ (as $\mu(T_2') \geq \mu(T_1')$ and the first coordinate in both sequences is equal to w). In this case by querying λ_{T_1} we get that $w(T + T_1') \leq 1 + \max(w - 1, w(T + (T_1 - \lambda_{T_1})'\ ))$. However, by our assumption $\mu(T_2' - u) \geq_L \mu(T_2' - \lambda_{T_2'}) \geq_L \mu(T_1' - \lambda_{T_1'})$. Thus by induction $w(T + (T_2' - u)) \geq w(T + (T_1' - \lambda_{T_1'}))$, so that $w(Q) \geq 1 + w(T + T_2' - u) \geq 1 + w(T + T_1' - \lambda_{T_1})$. On the other hand as $w(Q) \geq w$, it follows that $w(Q) \geq 1 + \max(w - 1, w(T + (T_1' - \lambda_{T_1'})) = w(T + T_1')$.

Second case $w(T) > w(T_1')$:

Consider an optimal search algorithm Q for $T + T_2'$. We may assume that $w(Q) \leq w(T)$, otherwise, a search algorithm for $T + T_1'$ that starts by querying the root of T_1, will use at most $w(T) + 1$ queries (as $w(T) > w(T_1)$), so that $w(T + T_1') \leq w(T + T_2')$ and the Lemma follows.

Let y be the first query of Q, it follows then that $y \in T$. Consider a search algorithm Q' for $T' + T_1$ that also starts by querying y. We are left with two trees $(T - y) + T_1'$ and $(T - y) + T_2'$. Here, $|T - y| < |T|$ so that we can use the induction assumption and obtain that $w((T - y) + T_1') \leq w((T - y) + T_2')$ and the Lemma follows. ∎

We now consider grouping of several trees.

Proposition 3.3 *Let $T = \sum_{i=1}^{k} T_i'$ (i.e., $T = T_1' + \ldots + T_k'$) and assume that $\mu(T_1') \leq \ldots \leq \mu(T_k')$, then $w(T_k') \leq w(T) \leq w(T_k') + k - 1$.*

Proof: The left inequality is immediate from Observation 2.1. The other inequality is obvious by the following search algorithm for T:

Start by querying $root(T_1), root(T_2), ...root(T_{k-1})$. If the answer is 'yes' on, say, the $i-th$ query then proceed by an optimal search on T_i. Otherwise proceed by an optimal search in T_k. Clearly this search takes at most $w(T_k') + k - 1$ queries. ∎

The criterion for determining the exact cost of the grouping operation is given by the following Lemma:

Lemma 3.4 *Let* $T = T'_1 + \ldots + T'_k$ *such that* $\mu(T'_1) \leq \cdots \leq \mu(T'_k)$ *then*
(a) $w(T) = w(T'_k)$ *iff* $w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k})) \leq w(T'_k) - 1$,
(b) For any integer $\alpha > w(T'_k)$, *we have that* $w(T) \leq \alpha$ *iff* $w(T'_1 + \ldots + T'_{k-1}) \leq \alpha - 1$.

Proof: The 'if' direction for (a) is trivial since, if $w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k})) \leq w(T'_k) - 1$, then by querying $\lambda_{T'_k}$ we are done. Similarly querying $root(T_k)$ if $w(T'_1 + \ldots + T'_{k-1}) \leq \alpha - 1$ proves the 'if' direction for (b).

For the other direction of (a) assume by negation that $w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k})) \geq w = w(T'_k)$. We show that $w(T'_1 + \ldots + T'_k) \geq w + 1$. Let Q be a search algorithm for $T'_1 + \ldots + T'_k$ that starts by querying x. If $x \notin T'_k$ then clearly $w(Q) \geq 1 + w$. Assume that $x \in T'_k$ and $w(T(x)) \leq w - 1$ (otherwise, for a 'yes' answer $w(Q) \geq 1 + w(T(x)) \geq w + 1$). Then, by the definition of μ, $\mu(T'_k - x) \geq_L \mu(T'_k - \lambda_{T'_k})$.

Now, let $\tilde{T} = T'_1 + \ldots + T'_{k-1}$, then by Lemma 3.3 we get that $w((T_k - x)' + \tilde{T}) \geq w((T_k - \lambda_{T'_k})' + \tilde{T}) \geq w$, hence

$$w(Q) = 1 + w((T_k - x)' + \tilde{T})) \geq 1 + w((T_k - \lambda_{T'_k})' + \tilde{T}) \geq 1 + w.$$

For (b) let $\alpha > w(T'_k)$ and assume that $w(T'_1 + \ldots + T'_{k-1}) \geq \alpha$. Let Q be a search algorithm for T, that first queries x. If $x \in T'_k$ then $w(Q) \geq 1 + w(T'_1 + \ldots + T'_{k-1} + T'_k - \lambda_{T'_k}) \geq 1 + w(T'_1 + \ldots + T'_{k-1}) \geq \alpha + 1$. If $x \in T'_i$ where $i \neq k$, then let $\tilde{T} = T'_1 + \ldots + T'_{i-1} + T'_{i+1} + \ldots + T'_{k-1}$. By Lemma 3.3 $w(T'_1 + \ldots + T'_{k-1}) = w(\tilde{T} + T'_i) \leq w(\tilde{T} + T'_k)$. This implies that (by considering the 'no' answer after query x) $w(Q) \geq 1 + w(\tilde{T} + T'_k) \geq 1 + w(\tilde{T} + T'_i) \geq 1 + \alpha$. ∎

Finally we relate now the sequences of complements:

Lemma 3.5 *Let* $T = T'_1 + \ldots + T'_k$ *then*
(a) if $w(T) = w(T'_k)$ *then* $\lambda_T = \lambda_{T'_k}$, $\mu(T) = [\mu_0, \ldots, \mu_l]$ *where* $\mu_0 = w(T'_k)$ *and* $[\mu_1, \ldots, \mu_l] = \mu(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k}))$.
(b) if $w(T) > w(T'_k)$ *then* $\lambda_T = root(T_k)$, $\mu(T) = [\mu_0, \ldots, \mu_l]$ *where* $\mu_0 = w(T)$ *and* $[\mu_1, \ldots, \mu_l] = \mu(T'_1 + \ldots + T'_{k-1})$.

Proof: Immediately follows from the proof of Lemma 3.4 ∎

4 Constructing the optimal search algorithm

In this section we show that computing $\mu(T)$ (and in fact λ_T) for a given tree T, can be done in polynomial time (in the size of T). The algorithm constructs $\mu(T), \lambda_T$ bottom up using the re-rooting and grouping operations.

We need the following notation: for a given tree T let $\mu(T) = [\mu_0, \mu_1, \ldots, \mu_l]$, we denote $\partial\mu = [\mu_1, \ldots, \mu_l] = \mu(T - \lambda_T)$. Note that $\partial\mu$ is computed from μ just by dropping out the first entry (although formally it depends on λ_T, one does not need to know it explicitly). Depending on the current operation in the decomposition the algorithm performs the following computations:

Re-rooting operation: Let $T = T'_1$.

1. Compute $\mu(T) = succ(\mu(T_1))$ (Theorem 3.1).
2. If $w(T) = w(T_1) + 1$ then $\lambda_T = root(T_1)$, otherwise (if $w(T) = w(T_1)$) then $\lambda_T = \lambda_{T_1}$.

   This can be done in $O(w(T))$ steps.

Grouping operation Let $T = T'_1 + \ldots + T'_k$

1. The μ sequences of $T'_1, \ldots, T'_k$ are sorted so that $\mu(T'_1) \leq_L \ldots \leq_L \mu(T'_k)$. Each sequence is of length at most $w(T)$ and each entry is bounded by $w(T)$. Thus sorting can be done (treating each sequence as a number) by $min(w^2(T)\log w(T), |T|\log|T|)$.

2. For any α it can be determined whether $w(T'_1 + \ldots + T'_k) \leq \alpha$. This is done using a recursive test as suggested by Lemma 3.4. The test is applied to the vectors of complement of subtrees $T'_1, \ldots, T'_k$, such that $test(\mu(T'_1), \ldots, \mu(T'_k), \alpha)$ returns true iff $w(T'_1 + \ldots + T'_k) \leq \alpha$. Let $test(\mu(T'_1), \ldots, mu(T'_k), \alpha) = TEST$, where $TEST$ is computed as follows:

   a) For $\alpha \geq \mu_0(T'_k) + k$ $\quad TEST = true.$
   b) For $\alpha \leq \mu_0(T'_k) - 1$ $\quad TEST = false.$
   c) For $\alpha = \mu_0(T'_k)$ $\quad TEST = test(\mu(T'_1), \ldots, \mu(T'_{k-1}), \partial\mu(T'_k), \alpha - 1)$,
   i.e, $test(T'_1 + \ldots + T'_k, \alpha) = test(T'_1 + \ldots + T'_{k-1} + T'_k - \lambda_{T'_k}, \alpha - 1)$.
   d) For $\mu_0(T'_k) < \alpha \leq \mu_0(T'_k) + k - 1$ $\quad TEST = test(\mu(T'_1), \ldots, \mu(T'_{k-1}), \alpha - 1)$,
   i.e., $test(T'_1 + \ldots + T'_k, \alpha) = test(T'_1 + \ldots + T'_{k-1}, \alpha - 1)$.

   The validity of *a*) and *b*) is due to Proposition 3.3, and that of *c*) and *d*) is due to Lemma 3.5.

   After every application of test it might be required to sort the remaining sequences of complements. The worst case is if the first coordinate of $\mu(T'_k)$ is deleted for which the correct place of $\mu(T'_k)$ amongst the k sequences can be found in $w(T)\log w(T)$ comparisons. The time needed to compute $test(T'_1 + \ldots + T'_k, \alpha)$ is therefore $O(w^2(T)\log w(T))$.

   We compute the cost $w(T'_1 + \ldots + T'_k)$ by binary searching for the correct α in the range $w(T'_k), ..., w(T'_k) + k - 1$. This may require $\log k$ applications of $test()$, Hence the maximal number of steps needed to compute the exact α is $O(w^2(T)\log^2 w(T))$ (not including the sorting of phase 1).

   Note that the computation of $w(T'_1 + \ldots + T'_k)$ also determines λ_T at the same time (as implied by Lemma 3.5).

3. After determining the exact cost $\alpha = w(T) = \mu_0(T)$, we proceed to find the next component of $\mu(T)$ according to the two cases of Lemma 3.5:
   (a) if $w(T) = w(T'_k)$ then $\lambda_T = \lambda_{T'_k}$ and $\mu_1(T) = w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k}))$
   (b) if $w(T) > w(T'_k)$ then $\lambda_T = root(T'_k)$ and $\mu_1(T) = w(T'_1 + \ldots + T'_{k-1})$ In this way we find each entry of $\mu(T)$. Note that re-sorting $\mu(T'_1), \ldots, \mu(T'_k - \lambda_{T'_k})$ is done in $w(T)\log w(T)$ steps, hence the whole process takes $O(w^3(T)\log^3 w(T))$ steps.

Theorem 4.1 *Let T be a tree with n nodes, then $\mu(T)$ and λ_T can be computed in $O(n \cdot w^3(T)\log w^3(T)) = O(n^4 \log^3 n)$ steps.*

Proof: We compute $\mu(T)$ by constructing T from subtrees working bottom up, starting from the leaves to the root. At each intermediate step we compute $\mu(T_1)$ for a subtree T_1 in $O(w^3(T)\log^3 w(T))$ steps using the above algorithm. Thus the whole process for a tree T with $|T| = n$ takes at most $O(n \cdot w^3(T)\log^3 w(T))$ steps. For bounded degree trees where $w(T) = O(\log n)$ this gives $O(n\log^4 n)$. For general trees this might be $O(n^4\log^3 n)$. Note, that each time we compute $w(T^i)$ of a subtree T^i, we also determine λ_{T^i}. Thus λ_T is computed in the same time bound, and an optimal search algorithm can be constructed. ∎

An example of the main stages of the algorithm is given in the appendix.

5 Search in Forests and Cartesian products Posets

Forests:
Our results hold for forests as well. Recall that in our model we assumed that one of the nodes in the given tree must be buggy. Let $\tilde{w}(T_1, T_2, \ldots, T_k)$ denote the cost of searching in a forest with k trees, where we do not assume that one of the nodes in $T_1, \ldots, T_k$ is buggy. It is easy to see that $\tilde{w}(T_1, T_2, \ldots, T_k) = w(T'_1 + T'_2 + \ldots + T'_k)$, where $T'_1 + T'_2 + \ldots + T'_k$ is just the grouping operation defined in section 3. The reason is that if none of the nodes in the forest is buggy, then an optimal search algorithm will identify the root of $(T'_1 + T'_2 + \ldots + T'_k)$ as the buggy node. On the other hand $T'_1 + T'_2 + \ldots + T'_k$ can be searched by applying a search algorithm for the forest $T_1, T_2, \ldots, T_k$; if this search gives as a result "no buggy element" then the answer for $T'_1 + T'_2 + \ldots + T'_k$ is its root.
Rooted Cartesian products:
A rooted Poset is a Poset with unique greatest element.

Claim 5.1 *Let P_1, P_2 be two rooted Posets and let $P = P_1 \times P_2$ be the (rooted) product Poset of P_1 and P_2, then $w(P) \leq w(P_1) + w(P_2)$.*

Proof: One may search P by first querying (a, x) where a is the greatest element of P_1 and x is optimal first query for P_2. This results in either searching in $P_1 \times P_2(x)$ or $P_1 \times (P_2 - P_2(x))$, where $P_2(x)$ is the Poset of all elements below x in P_2. Going on in this way by following the optimal strategy for P_2 for at most $w(P_2)$ queries we are left with a sub Poset that is isomorphic to P_1 for which additional $w(P_1)$ queries is enough.

As it is clear that $w(P) \geq max(w(P_1), w(P_2))$, the above observation gives $w(P)$ up to a factor of two.

An interesting example (the simplest non trivial) of rooted product is a product of two chains which is a rectangular lattice.

Claim 5.2 *Let P_1, P_2 be disjoint chains and let $P = P_1 \times P_2$, then $w(P_1) + w(P_2) - 2 \leq w(P) \leq w(P_1) + w(P_2)$.*

Proof: The upper bound follows from Claim 5.1. The lower bound follows from the fact that $w(P) \geq \lceil \log |P| \rceil$. If one or both length of P_1, P_2 is a power of 2 then in fact $\lceil \log |P| \rceil$ is the exact answer.

It is interesting to compare the above with the result of [5] for Cartesian product of chains for their model. Let P_n be the n element chain. In both models $w(P_n) = \lceil \log n \rceil$. However, in the Linial-Saks model $w(P_n \times P_n) = 2n - 1$ while in our model $w(P_n \times P_n) = 2\lceil \log n \rceil$.

6 Conclusions

We have presented a polynomial time algorithm for computing the optimal search strategy for 'forest' like Posets. The crux of the proof, and the algorithm, is the observation that the structure of a given tree T can be represented by a sequence of numbers $\mu(T)$. As an example, the cost of an optimal search of a complete binary tree (T) with n nodes is $\log n + \log^* n + \theta(1)$. We do not know any direct method for proving this result rather than computing $\mu(T)$ recursively (following our algorithm). [1]

Some problems are left open:

[1]Further details are left to the reader.

1. The main open problem is to give an algorithm that determines the cost (optimal strategy) for general Posets (namely, searching in directed acyclic graphs).

2. The randomized complexity is quite interesting too: For a given tree T and any buggy node can one do better on the average using a randomized search algorithm? (a randomized search algorithm can be viewed as a probability distribution on a set of deterministic search algorithms). For example, let T be a rooted star with n leaves, then it easy to see that any randomized search algorithm will make on the average n queries. The reason is that an adversary will put the "bug" in the root of the star, using the fact that any deterministic algorithm must query all the leaves first (this corresponds to the fact that the non-deterministic complexity of the star of n leaves is n). Randomization can help for some trees, e.g., for d regular tree of height h, $w(T) = h \cdot d$ while its randomized complexity is $\leq \frac{d \cdot (h+1)}{2}$. Our conjecture is that for every tree, T, its randomized complexity is at least $\frac{1}{2}w(T)$.

3. While our algorithm gives the exact cost for any tree, it would be interesting to prove tight upper and lower bounds as a function of some natural property of the tree. For example, $\log size(T)$ and d are lower bounds, where d is the maximal degree of T. However, none of these is tight in general. Another example is that for trees with maximum degree d, $\log_{\frac{d+1}{d}} size(T)$ is an upper bound, simply by querying a node that "splits" the tree into two parts of size $\leq \frac{d}{d+1} size(T)$ each.

4. For product of rooted Posets we have seen that the cost is at most the sum of the costs. Is this tight (up to an additive $O(1)$ term) for every rooted product?

References

[1] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[2] Phyllis G. Frankl and Elaine J. Weyker. Provable improvments on branch testing. *IEEE Transactions on Computer*, 9(10), September 1994.

[3] Joseph R. Horgan, Saul London, and Michael R. Lyu. Achieving software quality with testing coverage measures. *IEEE Transaction on Software Engineering*, October 1993.

[4] N. Linial and M. Saks. Every poset has a central element. *Journal of combinatorial theory*, 40:86–103, 1985.

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

APPENDIX

A A detailed example of the algorithm

This section contains a detailed example showing the main phases of the algorithm combining three subtrees $T = T'_3 + T'_2 + T'_1$ as described in Figure 3. In early stages, the algorithm computed $\mu(T'_i), \lambda_{T'_i}$ $i = 1, 2, 3$, such that:

$$\mu(T'_1) = [3,1,0] \;\; \lambda_{T'_1} =' c' \;\; \mu(T'_2) = [3,1,0] \;\; \lambda_{T'_2} =' g' \;\; \mu(T'_3) = [1,0] \;\; \lambda_{T'_3} =' z' \;\; .$$

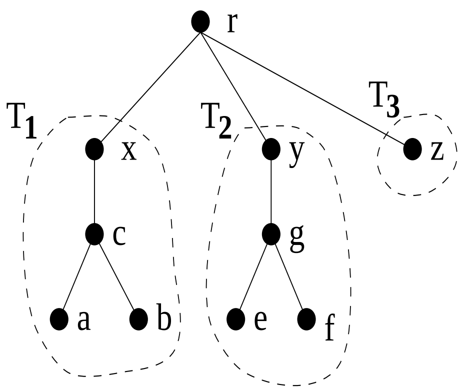

Figure 3:

We proceed according to the three stages of the grouping operation in the algorithm to compute $\mu(T)$. After sorting the vectors of complements we get that in our case (the number of subtrees is $k=3$) the possible cost of $w(T)$ is $3 \le w(T) \le 3+2=5$. Next, $test(T'_3+T'_2+T'_1,\alpha)$ is applied for $\alpha=4$ and is computed as follows (the label on the arrow $\xrightarrow{x)}$ denotes one of the four cases of $test()$ as described in section 4):

$$\begin{bmatrix} test(T'_3+T'_2+T'_1,4) \\ \mu(T'_1)=[3,1,0] \\ \mu(T'_2)=[3,1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{d)} \begin{bmatrix} test(T'_3+T'_2,3) \\ \mu(T'_2)=[3,1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{c)} \begin{bmatrix} test(T'_3+T'_2-\lambda_{T'_2},2) \\ \partial\mu(T'_2)=[1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{a)} TRUE$$

We continue to check if $w=3$ using:

$$\begin{bmatrix} test(T'_3+T'_2+T'_1,3) \\ \mu(T'_1)=[3,1,0] \\ \mu(T'_2)=[3,1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{c)} \begin{bmatrix} test(T'_3+T'_2+T'_1-\lambda_{T'_1},2) \\ \partial\mu(T'_1)=[1,0] \\ \mu(T'_2)=[3,1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{b)} FALSE$$

Hence, we have computed the first component in the vector of complements namely, $\mu_0(T)=4$.

The second coordinate $\mu_1(T)$ is obtained by computing $w(T'_3+T'_2)$ since $\lambda_T = root(T_1)$ and the complement is $T'_3+T'_2$. The possible range for $\mu_1(T)$ is $3 \le \mu_1(T) \le 3+2-1=4$, however 'true' is obtained for the minimal value $\mu_1(T)=3$:

$$\begin{bmatrix} test(T'_3+T'_2,3) \\ \mu(T'_2)=[3,1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{c)} \begin{bmatrix} test(T'_3+T'_2-\lambda_{T'_2},2) \\ \partial\mu(T'_2)=[1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{a)} TRUE$$

we get that $\mu_1(T)=3$. The third coordinate is computed in a similar way, namely, $1 \le \mu_2(T) \le 1+2-1=2$, yet

$$\begin{bmatrix} test(T'_3+T'_2-\lambda_{T'_2},1) \\ \partial\mu(T'_2)=[1,0] \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{c)} \begin{bmatrix} test(T'_3,0) \\ \mu(T'_3)=[1,0] \end{bmatrix} \xrightarrow{b)} FALSE \quad ,$$

hence $\mu_2(T)=2$.

At this stage we are left with a two nodes tree, hence $\mu_3(T) = 1$ and $\mu_4(T) = 0$. The set of nodes that were used in the different stages of $test(T_3' + T_2' + T_1', 4)$ forms the back-bone of the optimal search algorithm, i.e.,

$$
\begin{array}{cccccccccc}
Q_T & = & x & \xrightarrow{\text{no}} & g & \xrightarrow{\text{no}} & z & \xrightarrow{\text{no}} & y & \xrightarrow{\text{no}} & r \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
 & & Q_{T_1} & & Q_{T(g)} & & z & & y & &
\end{array}
$$