Electronic Colloquium on Computational Complexity - Reports Series 1996 - available via:
ECCC FTP: ftp.eccc.uni-trier.de:/pub/eccc/
TR96-044 WWW: http://www.eccc.uni-trier.de/eccc/
Email: ftpmail@ftp.eccc.uni-trier.de with subject 'help eccc'

Optimal Search in Trees
Extended Abstract + Appendix

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 the binary search. In the 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 bellow 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 $O(n^2 \log^2 n)$ 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.

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 $s_i < s_{i+1}$. 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 need 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 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, or 'no' if the element is not bellow x. 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 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

*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

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.

One motivation for study search in trees (and Posets in general) is that it forms a generalization of the binary search in linear orders to a 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 to 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 in general 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, say by checksum-like tests (or randomized communication complexity equality testing). Namely, such equality test easily detect that there is a fault but give no information which node of the tree is corrupted. Using search on the tree (by querying correctness of subtrees) allows us to 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 cost by 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 plausible that in certain situations it might be possible to embed such a set of tests (e.g., the union over all tests 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 binary search in the syntactic tree of the program.

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 that are "stored" as 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 in the stored set. The possible queries in this case is, 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 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. Linial and Saks prove lower and upper bounds for the number of queries need to search in Posets in terms of some of the Poset properties, however, they present 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 for any tree (forest). Let $T(v)$ be the sub tree 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 need 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 if its subtrees, avoiding the need to compute the cost of searching in various complement subtrees. Next, we generalize the results for Forest, and also draw some conclusions for cartesian product Posets.

2 Basic definitions and preliminary Facts

Let us start with some notations: The sub-tree 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 $v\in T-r$, Q_v 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 is define as follows:

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

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 cost of a single node is zero, since this node must be buggy and no query is needed.

The following is immediate from the definitions.

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

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, lead 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 any $j < i$, $\mu_j = \rho_j$.*

Note that $<_L$ is a total order on the set of all finite sequences of natural numbers.

Definition 2.3 *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 $\mu(L)$ of a path L with five nodes is $< 3, 0 >$ since the node u that achieves the smallest $\mu(L-u)$ is the son of the root. μ of a star with n leaves is $< n, n-1, \ldots, 2, 1, 0 >$.

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, and it is taken as the empty sequence.*

See Fig 1 for an example of λ_T and $\mu(T)$.

In several cases we will also need the nodes that are associated with the coordinates of $\mu(T)$.

Definition 2.5 *The sequence of nodes $\beta(T) = < \beta_1, \beta_2, \ldots, \beta_l >$ associated with the sequence of complements $\mu(T) = \mu_0, \mu_1, \mu_2, \ldots, \mu_l$ is defined recursively as follows:*
$\beta_1 = \lambda_T$ and $< \beta_2, \ldots, \beta_l >$ is $\beta(T - \lambda_T)$.

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, $w(T)$, using the formula of Fact 2.1 in a bottom-up manner. Our solution is to compute $\mu(T)$ for subtrees of T. The decomposition of T into subtrees is chosen so that we can characterize $\mu(T)$ as a function of the μ's of its components. 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, where $\mu(T'_1) \leq_L \ldots \leq_L \mu(T'_k)$. Note that we write it as an operation on the rooted trees $T'_1, \ldots, T'_k$ as we will relate $\mu(T)$ to $\mu(T'_1), \ldots, \mu(T'_k)$.

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

By Fact 2.1 it is obvious that the ability to compute an optimal first query for a tree T defines an optimal search algorithm for T. Thus computing $\mu(T), \beta(T)$ for T certainly provides this information as $\lambda_T = \beta(T)_0$. In the following two sections we will show how $\mu(T)$ is determined by the μ's of the subtrees used to decomposed T according to the operations defined above.

3.1 Properties of the re-rooting operation

As explained in section 3, we need to compute $\mu(T')$, and determine $\lambda_{T'}$. The proposed method is based on the observation that a "jump" in the coordinates of the sequence $\mu(T)$ is a necessary and sufficient indication that the cost of T' remains the same. The term "jump" indicates that the difference between some two consecutive coordinates 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 consider a path L with three nodes, so that $\mu(L) =< 2, 0 >$ contains a jump. Indeed L' is a path with 4 nodes and its cost remains 2. The reason why $w(L')$ did not increased is that we can query λ_L. For a 'yes' answer we proceed as we would have done in L and complete the search with two more queries. For a 'no' answer we are left with $L' - \lambda_L = (L - \lambda_L)'$ but $w(L - \lambda_L) = 0$ as $(L - \lambda_L)$ is a single node. Thus $w(L' - \lambda_L) = 1$ which increase comparing to L but not enugh to increase $w(L')$. Another example is given in the appenix section A. the re-rooting T' of the tree T in figure 1, showing that the cost increases if $\mu(T)$ contains no jump. For T there we get that $\lambda_T = v$ and that $\mu(T) =< 3, 2, 1, 0 >$ contains no jump. Indeed, the cost has increased, and $w(T') = 4$.

Since we can start a search algorithm by querying $root(T)$ we get that the cost can increase by at most one.

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

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

The above properties can help us prove that a jump in $\mu(T)$ indicates that $w(T') = w(T)$.

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: We will use an induction on $|T|$ to show that there exist a search algorithm Q for T', with $w(Q) = w(T)$. The premise of the Lemma is possible only for a tree with cost greater than 1, hence

the induction base includes only a path L 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 L, the sequence of complements is $< 2, 0 >$, and indeed $w(L') = w(L) = 2$.

Let T be a tree with $w(T) \geq 3$, and consider the following search algorithm:

$$
\begin{array}{lll}
Q = & \lambda_T \stackrel{\text{no}}{\longrightarrow} & Q_1 \\
 & \downarrow & \\
 & Q_2 &
\end{array}
$$

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, $w(Q_2) \leq w(T) - 1$ and thus $w(Q) \leq \max(w, 1 + w(T' - \lambda_T)) = \max(w, 1 + w((T - \lambda_T)'))$.

Let us first consider the case where the jump is for $i = 1$ namely $\mu_1 = w(T - \lambda_T) < w(T) - 1$. By proposition 3.2 for $T - \lambda_T$ we have that $w((T - \lambda_T)') \leq w(T - \lambda_T) + 1 \leq \mu_1 + 1 \leq w(T) - 1$. Thus $1 + w(T' - \lambda_T) \leq w(T)$.

Consider the case where $i > 1$. Let $T_1 = T - \lambda_T$ then by the definition of μ we get that $\mu(T_1) = < \mu_1, \ldots, \mu_l >$, while $\mu_1 = w(T - \lambda_T) = w(T) - 1$. Thus, the premise of the Lemma holds for T_1 (with $i' = i - 1$), so by the induction hypothesis $w(T_1') = w(T_1) = w(T) - 1$, yielding that $1 + w(T' - \lambda_T) \leq w(T)$. ∎

The counter direction of Lemma 3.1 is also true:

Lemma 3.2 *Let $\mu(T) =< \mu_0, \ldots, \mu_l >$. If $w(T') = w(T)$, then there exists i ($i \in \{1, \ldots, l\}$) 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.

By definition $\mu(T)$ is a (strongly) monotone decreasing sequence of natural numbers, with $\mu_0 = w$, hence we get that if $\forall i,\ \mu_i \geq w - i$ then $\mu(T) = < w, w-1, w-2, \ldots, 1, 0 >$. Let Q be an optimal algorithm for T' starting with a vertex x. If $w(T(x)) \geq w$ then on 'yes' answer, we are left with $T(x)$ hence $w(Q') \geq w+1$ 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 $\mu(T-x) \geq_L \mu(T-\lambda_T)$, and in particular $w(T-x) \geq w(T-\lambda_T) \leq w-1$. Hence, we may assume that $w(T-x) = w(T-\lambda_T)$, otherwise $w(Q) = w+1$ and we are done. But this means that $\mu(T-x) = \mu(T-\lambda_T) =< w-1, w-2, \ldots, 1, 0 >$ as the successor of $\mu(T-\lambda_T)$ in the order L has 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 3.1 *for a given tree T, $w(T') = w(T)$, iff there exists $i \geq 0$ such that $\mu(T)_i < w(T) - i$.*

Theorem 3.1 together with Proposition 3.2 show how to compute $w(T')$ knowing $\mu(T)$. We now show how to compute the rest of the components of $\mu(T')$ based on the values of $\mu(T)$.

Lemma 3.3 *Let $\mu(T) =< \mu_0, \mu_1, \ldots, \mu_l >$. Assume that there is a "jump", and let i be the largest index for which $\mu_{i-1} > l - i + 1$, namely*

$$\mu(T) =< \mu_0, \mu_1, \ldots, \mu_{i-1}, \overbrace{l-i}^{\mu_i}, l-i-1, \ldots, 2, 1, 0 > \quad and \quad \mu_{i-1} \geq l-i+2.$$

$$Then\ \lambda_{T'} = \lambda_T \ and \quad \mu(T') =< \mu_0, \mu_1, \ldots, \mu_{i-1}, l-i+1, 0 > \quad .$$

Proof: Omitted.

3.2 Properties of the grouping operation

We turn now to analyze the cost of grouping k re-rooted trees $T'_1 + T'_2 + \ldots + T'_k$, assuming that we have computed their sequences of complements. We first start with a simpler operation 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 T' (one of the trees) is a $re-rooted$ tree. The amalgamation operation exhibits some monotonicity:

Lemma 3.4 *If $\mu(T'_1) \leq_L \mu(T'_2)$ then $w(T'_1 + T) \leq w(T'_2 + T)$.*

Proof: Omitted.

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 $w(T'_k) \leq w(T) \leq w(T'_k) + k - 1$.*

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

Lemma 3.5 *Let $T = T'_1 + \ldots + T'_k$ 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 $w > w(T'_k)$, we have that $w(T) \leq w$ iff $w(T'_1 + \ldots + T'_{k-1}) \leq w - 1$.

Proof: The 'if' direction for (a) is trivial since, if $w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k})) \leq w(T'_k)$ then by querying $\lambda_{T'_k}$ we are done. Similarly querying $root(T_k)$ if $w(T'_1 + \ldots + T'_{k-1}) \leq w - 1$ proves the direction '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 $w(T'_1 + \ldots + T'_k) > w$. 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 $w(Q) \geq 1 + w(T(x)) \geq w + 1$). Thus, 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 3.4 we get that $w((T_k - x)' + \tilde{T}) \geq w((T_k - \lambda_{T'_k})' + \tilde{T})$,

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

For (b), let $w > w(T'_k)$ and assume that $w(T'_1 + \ldots + T'_{k-1}) \geq w$. Let Q be a search algorithm for T, that first queries x. If $x \in T'_k$ then $w(Q) \geq 1 + w(T'_1 + \ldots + T'_{k-1} + T'_k - \lambda_{T'_k}) \geq 1 + w(T'_1 + \ldots + T'_{k-1}) \geq w + 1$. 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 3.4 $w(T'_1 + \ldots + T'_{k-1}) = w(\tilde{T} + T'_i) \leq w(\tilde{T} + T'_k)$. However, $w(Q) > 1 + w(\tilde{T} + T'_k) \geq 1 + w(\tilde{T} + T'_i) \geq 1 + w$. ∎

Lemma 3.6 *Let* $T = T'_1 + \ldots + T'_k$ *then*
(a) if $w(T) = w(T'_k)$ *then* $\lambda_T = \lambda_{T'_k}$ *and* $\mu(T) = <\mu_0, \ldots, \mu_l>$ *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* $\lambda_T = root(T'_k)$ *and* $\mu(T) = <\mu_0, \ldots, \mu_l>$ *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 3.5 ∎

4 Constructing the optimal search algorithm

In this section we show that computing the optimal search algorithm for a given tree T, can be done in polynomial time (in the size of T). The output of the algorithm is $\mu(T)$ and λ_T. The algorithm constructs $\mu(T), \lambda_T$ bottom up using the re-rooting operation or the grouping operation. Depending on the current operation in the decomposition the algorithm performs the following computations:

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

1. Compute $\mu(T)$ by determining whether there is a 'jump' in $\mu(T_1)$ (Theorem 3.1, and Lemma 3.3).
2. If $w(T) = w(T_1) + 1$ then $\lambda_T = root(T_1)$, other wise (if $w(T) = w(T_1)$) then $\lambda_T = \lambda_{T_1}$.

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

1. The complement sequences of $T'_1, \ldots, T'_k$ are sorted so that $\mu(T'_1) \leq_L \ldots \leq_L \mu(T'_k)$. This can be done in $O(|T|)$ steps.

2. For any $w \leq w(T'_k) + k$ it can be determined whether $w(T'_1 + \ldots + T'_k) \leq w$. This is done using the following recursive test as suggested by Lemma 3.5.

$$test(T'_1 + \ldots + T'_k, w) = \begin{cases} test(T'_1 + \ldots + T'_{k-1} + T'_k - \lambda_{T'_k}, w-1) & w = \mu(T'_k)_0 \\ test(T'_1 + \ldots + T'_{k-1}, w-1) & w > \mu(T'_k)_0 \\ false & \exists i, w(T'_i) > w \\ true & k = 1 \ \wedge \ w(T'_k) \leq w \end{cases}$$

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 $\log k$ comparisons. The maximal time needed to compute $test(T'_1 + \ldots + T'_k, w)$ is therefore $O(|T| \log |T|)$.

We compute the cost $w(T'_1 + \ldots + T'_k)$ by binary searching for the correct w 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 is $O(|T| \log^2 |T|)$.

3. After determining $w(T) = \mu_0(T)$, λ_T is determined by Lemma 3.6 and we proceed finding the next component of $\mu(T)$ according to the two cases of Lemma 3.6:
(a) if $w(T) = w(T'_k)$ then $\lambda_T = \lambda_{T'_k}$ and $\mu(T)_1 = w(T'_1 + \ldots + T'_{k-1} + (T'_k - \lambda_{T'_k}))$
(b) if $w(T) > w(T'_k)$ then $\lambda_T = root(T'_k)$ and $\mu(T)_1 = w(T'_1 + \ldots + T'_{k-1})$ Note that in both cases the μ of each subtree in the new grouping is known! Thus determining the value of every component of $\mu(T)$ is reduced to computing μ_0 on a restricted set of subsequences of the known sequences: $\mu(T'_1), \mu(T'_2), \ldots, \mu(T'_k)$. The whole process takes at most $O((|T| \log |T|)^2)$ steps.

Theorem 4.1 *Let T be a n nodes tree then $\mu(T)$ and λ_T can be computed in $O(n^2 \log^2 n)$ steps.*

Proof: We compute $\mu(T)$ and λ_T by constructing T from subtrees working bottom up, starting from the leaves to the root. At each intermediate step we compute $\mu(T_1), \lambda_{T_1}$ for a subtree T_1 in $O(|T_1|^2 \log^2 |T_1|)$ steps using the above algorithm. Thus the whole process for a tree T with $|T| = n$ takes $t(n)$ steps, where $t(n)$ satisfies that $t(n) \leq n^2 \log^2 n + \sum t(n_i)$, and $\sum n_i = n$. This yields $t(n) = O(n^2 \log^2 n)$. ▮

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 $w'(T_1, T_2, \ldots, T_k)$ 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 $w'(T_1, T_2, \ldots, T_k) = w(T'_1 + T'_2 + \ldots + T'_k)$, 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 a search in the forest $T_1, \ldots T_k$ is search for $T'_1 + T'_2 + \ldots + T'_k$ with the same cost, where if the search gives "no buggy element" for $T_1, \ldots, T_k$ then the root of $T'_1 + T'_2 + \ldots + T'_k$ is the answer.

Rooted Cartesian products:
A rooted Poset is a Poset with one 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. Thus after $w(P_2)$ 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 L_1, L_2 be disjoint chains and let $L = L_1 \times L_2$, then $w(P_1) + w(P_2) - 2 \leq w(L) \leq w(P_1) + w(P_2)$.*

Proof: The upper bound follows from Claim 5.1. The lower bound follows from the fact that $w(L) \geq \lceil \log |L| \rceil$. If one or both length of L_1, L_2 is a power of 2 then in fact $\lceil \log |L| \rceil$ is the exact answer.

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

6 Conclusions

We have presented a polynomial time algorithm for computing the optimal search strategy for 'forest' like Posets. Some problems are left open:

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. While our algorithm gives the exact cost for any tree, it would be interesting to prove 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, $d \log_{d+1} size(T)$ is an upper bound).

3. 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 Example for choosing λ_T and the re-rooting operation

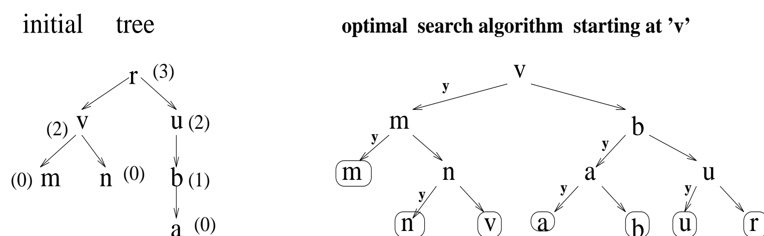


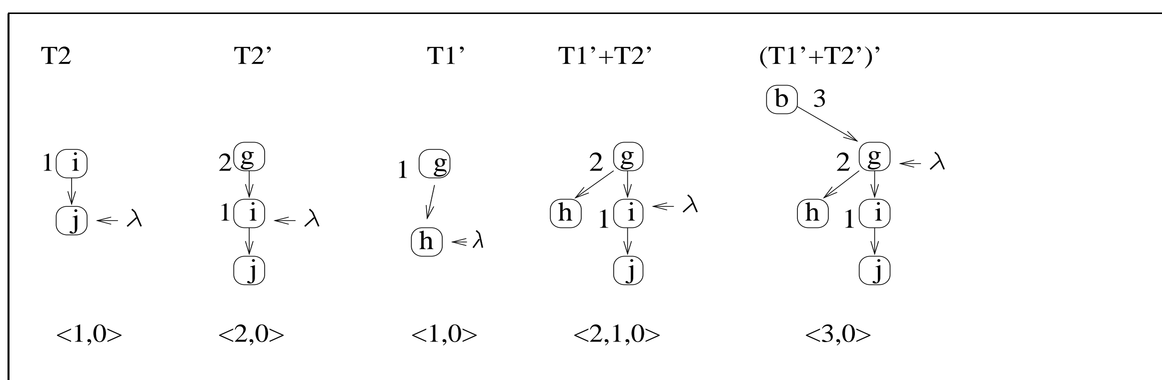
Figure 1:

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 $\mu(T - u) =< 3, 0 >$. The intuition that λ_T is an optimal node is reflected by the fact that there is an optimal search algorithm that starts by querying v (as is depicted in the right side of the figure). However, the reader can verify that the cost of any search algorithm starting at u will be at least 4.

Another use for this example is to show the effect of the re-rooting T' of the tree T in figure 1, where the cost increases if $\mu(T)$ contains no jump. For T there, we get that $\lambda_T = v$ and that $\mu(T) =< 3, 2, 1, 0 >$ contains no jump. Indeed, the cost has increased, and $w(T') = 4$.

B A detailed example of the algorithm

This section contains a detailed example showing the operation of the algorithm on a complex tree with 11 nodes. The figures describe a sequence of re-rooting and grouping operations. Each item in the figure is titled by the corresponding operation (e.g., $T_1' + T_2'$ or T'), the cost of every node, λ_T, and $\mu(T)$.



First $\mu(T_2) =< 1, 0 >$, there is no jump, hence, $w(T_2') = w(T_2) + 1 = 2$, and $\lambda_{T_2'} = root(T_2)$. Next, the computation of $test(T_1' + T_2', 2)$:

$$test(T_1' + T_2', 2) = test(T_1' + (T_2' - \lambda_{T_2'}), 1) = true,$$

hence, the cost remains the same $w(T_1' + T_2') = 2$ and $\lambda_{T_1'+T_2'} = \lambda_{T_2'}$. As there is no jump in $mu(T_1' + T_2')$ the cost of $(T_1' + T_2')'$ has increased and $\lambda_{(T_1'+T_2')'} = g$, and $\mu((T_1' + T_2')') =< 3, 0 >$.

The grouping $T3' + ((T_1' + T_2')' + T4'$ obtains the cost 4 as follows:

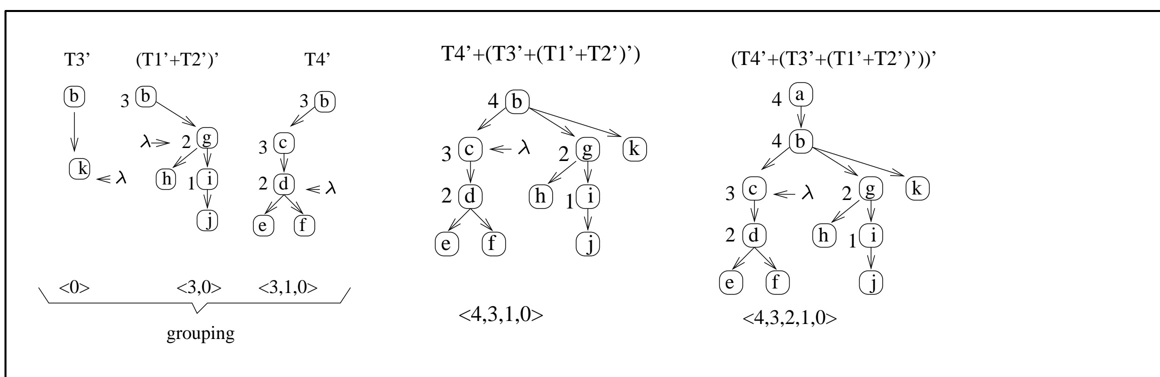
$$test(T3' + ((T_1' + T_2')' + T4', 4) = test(T3' + ((T_1' + T_2')', 3) =$$

$$test(T3' + ((T_1' + T_2')' - \lambda_{((T_1'+T_2')'}, 2) = test(T3', 2) = true$$

and $\lambda_{T3'+(T_1'+T_2')'+T4'} = \lambda_{T4'} = c$. Note that 4 is indeed the minimal cost as

$$test(T3' + ((T_1' + T_2')' + T4', 3) = test(T3' + ((T_1' + T_2')' + T4' - \lambda_{T4'}, 2) =$$

$$test(T3' + ((T_1' + T_2')' - \lambda_{((T_1'+T_2')'} + T4' - \lambda_{T4'}, 1) = \ldots false$$



Finally, $\mu(T4' + (T3' + (T_1' + T_2')'))$ contains a jump, hence the cost of $(T4' + (T3' + (T_1' + T_2')'))'$ does not increase and $\lambda_{(T4'+(T3'+(T_1'+T_2')'))'} = \lambda_{T4'+(T3'+(T_1'+T_2')')} = c$.