Generalization of Binary Search: Searching in Trees and Forest-Like Partial Orders

Krzysztof Onak*
MIT, CSAIL
konak@mit.edu

Paweł Parys
Warsaw University
parys@mimuw.edu.pl

August 14, 2006

Abstract

We extend the binary search technique to searching in trees. We consider two models of queries: questions about vertices and questions about edges. We present a general approach to this sort of problem, and apply it to both cases, achieving algorithms constructing optimal decision trees. In the edge query model the problem is identical to the problem of searching in a special class of tree-like posets stated by Ben-Asher, Farchi and Newman [1]. Our upper bound on computation time, $O(n^3)$, improves the previous best known $O(n^4 \log^3 n)$. In the vertex query model we show how to compute an optimal strategy much faster, in $O(n)$ steps. We also present an almost optimal approximation algorithm for another class of tree-like (and forest-like) partial orders.

1 Introduction

Searching in ordered structures is a fundamental problem of computer science. In general, suppose that we are given a set with some structure imposed on it. It is often the case that this structure enables us to ask queries that help us locate a required element, and our goal is to minimize the number of inquiries in the worst case. This problem can be viewed as a game between us and a malicious oracle that answers our questions in a coherent way, but does not fix the location of the element in advance, and does its best to postpone the moment when we locate the required element.

One such problem is the problem of locating an element in a sorted array. An optimal solution to this problem is binary search (see Knuth's book [6] for an exhaustive description). The sorted array can be viewed as a line of vertices, where vertices correspond to entries of the array, and two consecutive vertices are connected with an edge. When we ask a question, we actually query whether the vertex that we are looking for is to the left or right of the object of our query. A natural generalization of this problem is the problem of locating a vertex in a tree when we are allowed to ask in which direction from a given place the required vertex is.

Note that searching in a sorted array corresponds to searching in a totally ordered set, and therefore another natural generalization of this problem is searching in partially ordered sets. In a given partial order we want to locate an element x in as few questions as possible, where each question has the form "$x \leq e$?" for some member e of the partial order. While a positive answer indicates that $x \leq e$, a negative answer

*This work was done when the author was a student at Warsaw University.

says only that $x \leq e$ does not hold, and does not indicate whether $e < x$ or both elements are completely unrelated. Our focus is on forest-like partial orders, where a partial order is generated by a forest-like diagram.

At the intersection of these two searching problems lies the following problem, which is a special case of both of them. Suppose that we are given a rooted tree, want to locate a vertex in it, and are allowed to ask whether the required vertex belongs to the subtree rooted at v for each vertex v.

This problem has been considered by Ben-Asher *et al.* [1], who pointed out possible practical applications. Consider a large file system that has a remote copy. At some point, for example after a temporary loss of connection, it turns out that they differ, and we wish to synchronize them, minimizing the amount of communication between servers on which they are stored. Provided the difference is not large, i.e. only a few files, it is plausible that we may want to find files and directories at which the file systems differ one by one, instead of checking the whole file system tree, descending from the root. The problem of locating a single directory or file at which they differ can be reduced to the tree searching problem being considered. To test whether two directories or files differ, it suffices that one server sends the name of this directory or file to the other server, and either receives its checksum, or learns that it does not exist on the remote server. If it does not exist, or the remote and local checksums are identical, we look for a difference that is not in this directory. Otherwise, we focus on this directory and its subdirectories.

Consider also an application built in such a way that dependencies between modules constitute a tree. We know that the application is buggy; specifically, we know that some module is buggy. For each module we have a set of exhaustive tests, that checks for instance whether all branches and statements in a given module work properly. The worst-case number of modules that we test to locate the buggy module would be minimum, if we were able to find an optimal solution to the tree searching problem.

1.1 Related work

Numerous modifications and extensions of binary search have been considered, for instance Fibonaccian search [4], interpolation search [10], searching when costs of questions are non-uniform [5, 9, 3], and searching in partially ordered sets [1, 2, 7, 8].

Linial and Saks have investigated related problems of searching in ordered structures, and among them searching an order ideal in a poset by querying whether the required order ideal contains a given element [7, 8]. They showed that in this setting the best strategy matches up to a multiplicative constant the information theoretic lower bound, i.e. in the optimum strategy the number of queries is of order $O(\log |\mathcal{I}|)$, where $\mathcal{I}$ is the set of order ideals.

Ben-Asher *et al.* [1] have shown how to find a worst-case optimal searching strategy in $O(n^4 \log^3 n)$ steps for forest-like posets of type A (see Section 1.2). Carmoe *et al.* [2] have proven that the problem of determining an optimal strategy for searching in posets is NP-hard, and presented an approximation algorithm for random posets.

1.2 Preliminaries

For convenience, we will call a vertex or element of a poset that we wish to locate a *target*. By a *strategy* for a given instance of a problem, we mean a correct decision tree enabling discovery of the target, and by an *optimal strategy* (in which the worst-case number of questions is minimum) we mean a correct decision tree of the lowest possible height.

Definition 1 Let $\langle S, \leq \rangle$ be a finite partial order. We say that a directed acyclic graph $G = (V, E)$ is a

diagram for $\langle S, \leq \rangle$ if $V = S$, and for every $a, b \in S$, $a \leq b$ if and only if there is a directed path from b to a.

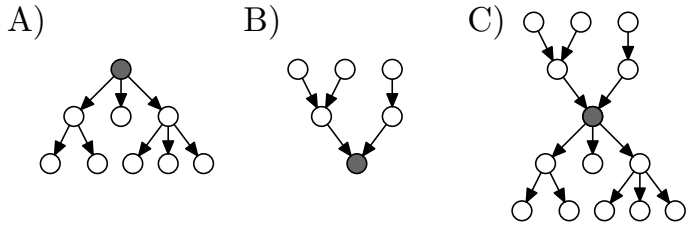

Figure 1. Sample posets of each type. The gray vertices are the vertices v from the definition of considered types of posets.

We consider partial orders that have special directed trees as diagrams, and we distinguish three types of them (see Figure 1 for examples):

Type A There exists a vertex v from which one can reach any vertex.

Type B There exists a vertex v reachable from every vertex. Type B is basically the reverse of type A, and vice versa.

Type C There exists a vertex v such that for every vertex w, either v is reachable from w, or w is reachable from v. Call the lowest such element v a *connecting element*. Type C combines types A and B.

Our algorithms work for forests as well, where a forest of type X is a diagram that consists of disjoint trees of type X.

1.3 Considered problems and results

1.3.1 Searching in trees

Our goal is to find a vertex in a tree. In order to determine it we are allowed to ask queries of one of the following two types:

- **questions about vertices**—we ask a question about a vertex v, and we learn either that v is the target, or which edge outgoing from v starts the simple path from v to the target;
- **questions about edges**—an answer to the question about an edge e indicates which of e's two endpoints is closer to the target.

Note that the classical binary search technique works in both cases if the tree is a simple path. The cases of questions about vertices and questions about edges are equivalent to, respectively, the case of 3-way comparisons (the response to a comparison $a \,?\, b$ is one of $a < b$, $a = b$, $a > b$) and 2-way comparisons (when we simply learn whether $a < b$ is true) in binary search. In total orders we have only two "directions": greater or lower numbers, while in trees the situation may be more complicated, as there may be more than two directions in which we can leave a vertex. See Figure 2 for an example of a tree and optimal strategies for it.

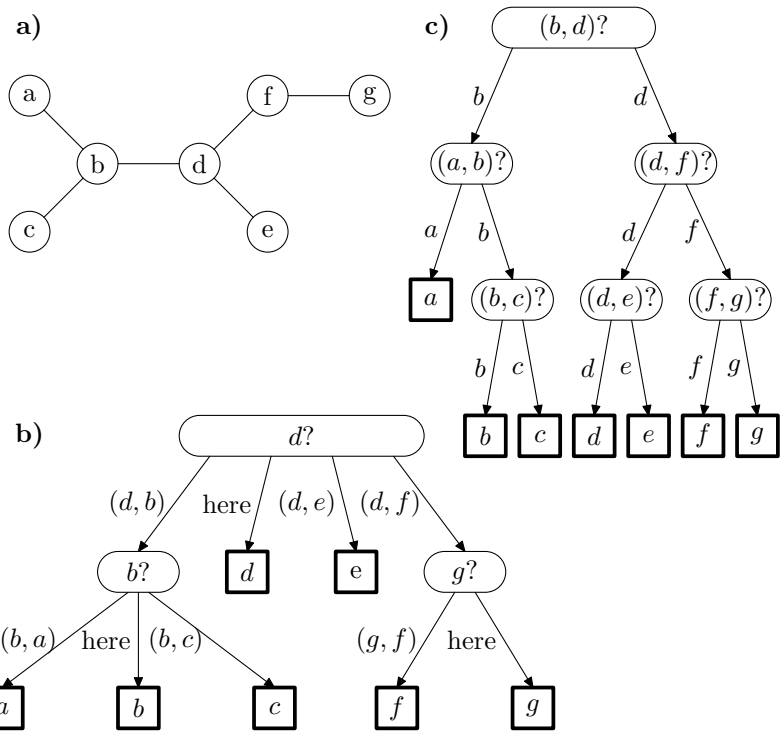

Figure 2. A sample tree (a) and optimal strategies in the vertex query model (b) and in the edge query model (c).

Both cases and their solutions are alike. Therefore, we first present a general approach that works for this kind of problem, and then we show how to efficiently apply our method to compute optimal strategies. Namely, it turns out that to determine an optimal worst-case strategy it suffices to find a specific function on objects that we can ask about. This function must meet the condition which says that two objects on which the function takes the same value should be separated by an object of greater value. Moreover, the function must have the lowest possible maximum. This maximum equals the optimum worst-case number of questions for a given tree, and the function describes an optimal strategy, which can be easily extracted from it. In the beginning, we arbitrarily root our input tree, and give a procedure that constructs the required function in a bottom-up manner.

We achieve running time of $O(n)$ in the vertex query model, and $O(n^3)$ in the edge query model.

1.3.2 Searching in forest-like posets

This time we consider searching in forest-like posets. If x denotes the target, we are allowed to ask questions of the form "$x \leq e$?", for each element e. We assume that the input partial order is given as its forest-like diagram. For forest-like posets of type A we can find optimal decision trees in $O(n^3)$ time (which improves the previous best $O(n^4 \log^3 n)$ in [1]). For posets of type B and C we show how to find decision trees of height at most 1 more than the optimal in $O(n)$ and $O(n^3)$ steps, respectively.

2 Searching in trees

First we show a general technique that enables us to determine an optimal strategy in both models of queries, and can be successfully applied to many similar problems.

We define a specific type of functions such that there is almost mutual correspondence between functions of this type and reasonable strategies, where a strategy is reasonable if we never ask in it a question to which an answer can be inferred from previous questions and answers. The domain of a function of this type is the set of objects about which we can ask questions. The function maps them into integers, and meets the following condition: if at two distinct objects the function takes on the same value, then on the simple path from one of them to the other the function achieves a greater value. It turns out that due to the previously mentioned correspondence the maximum of the function is not less than the minimum worst-case number of queries. Furthermore, a function of the lowest maximum corresponds to an optimal strategy, and its maximum equals the optimal worst-case number of queries. We show also how to extract a strategy from a function describing it.

To build a function that meets the required conditions, we arbitrarily root the tree, and in a bottom-up fashion, fix its consecutive values. We strengthen the property of having the lowest maximum, and recursively construct a function of the new stronger property in each subtree. To achieve this goal we find for each model of queries a special procedure that given functions having the new property, and defined on subtrees rooted at children of a vertex v, extends them to a function on the tree rooted at v which also has the new property.

2.1 Strategy description

The aforementioned functions, which we will call strategy functions, are a handy way to describe strategies. The general idea behind them is the following. A value of a strategy function at a vertex or edge x says how many questions suffice to finish a search when we ask queries according to the strategy that the function describes, and when the next query is about x. First, we give different, though similar, definitions of strategy functions in each model. Later lemmas and theorems will be common and independent of a model of queries.

Definition 2 A function $f\colon V \to \mathbb{N}$ is a *strategy function* in the vertex query model if for each pair of distinct vertices $v_1, v_2 \in V$ if $f(v_1) = f(v_2)$, then on the simple path from v_1 to v_2 there is a vertex v_3 such that $f(v_3) > f(v_1)$.

Definition 3 Function $f\colon E \to \mathbb{Z}_+$ is a *strategy function* in the edge query model if for each pair of distinct edges $e_1, e_2 \in E$ if $f(e_1) = f(e_2)$, then on the simple path from e_1 to e_2 there is an edge e_3 such that $f(e_3) > f(e_1)$.

It is worth noticing that in the edge query model $\mathbb{Z}_+ = \{1, 2, 3, \ldots\}$, not $\mathbb{N} = \{0, 1, 2, \ldots\}$, is the codomain. This is a consequence of the fact that for every edge we must ask a question about it to distinguish vertices it connects. Two examples of strategy functions are shown in Figure 3.

Definition 4 If G is a subgraph of a tree T, and f is a strategy function on T, we denote by $\mathcal{M}_f(G)$ the maximum of f on elements of G, or 0, if f is not defined on any element of G (possible only for questions about edges).

In strategy functions we separate the same values of a function with greater values. This allows us to state the following obvious lemma:

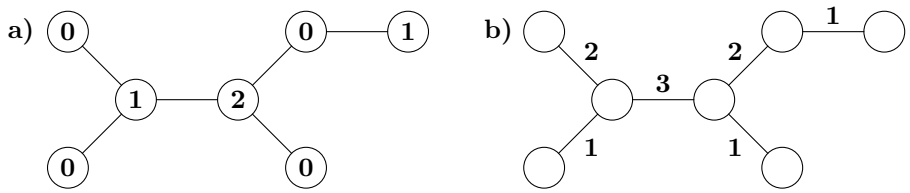

Figure 3. Sample strategy functions on vertices (a) and on edges (b) corresponding to strategies in Figure 2.

Lemma 5 *Let f be a strategy function on a tree T, and $T' = (V', E')$ a subtree of T. If $\mathcal{M}_f(T') > 0$, then there exists exactly one vertex $v \in V'$ such that $f(v) = \mathcal{M}_f(T')$ (exactly one edge $e \in E'$ such that $f(e) = \mathcal{M}_f(T')$). On the other hand, if $\mathcal{M}_f(T') = 0$, then there is at most one vertex in T'.*

In the next two lemmas we establish the aforementioned almost mutual correspondence between strategies and strategy functions.

Lemma 6 *For each strategy of asking questions in a given tree T, there exists a strategy function f such that $\mathcal{M}_f(T)$ is not greater than the maximum number of questions asked by the strategy in the worst case.*

Proof We can limit ourselves to strategies in which we do not ask questions that only have one answer consistent with previous answers. Such questions can be omitted.

Each answer points out a subtree induced by vertices that still can be the target. Subtrees for answers to a given query are pairwise disjoint. All following questions will be about objects (vertices or edges) only in the indicated subtree, since to all queries about objects outside this subtree there is always exactly one answer consistent with previous answers. This implies that we ask about a given object (a vertex or edge) at most once in the whole decision tree.

Let m be the worst-case number of questions in the strategy, and for each object x about which we can ask, let $f(x)$ be equal to $m - i + 1$, if we ask about x in the i-th turn in the strategy, or 0, if we never ask about x. Clearly $\mathcal{M}_f(T) \leq m$, and we only need to make sure that function f meets the definition of a strategy function.

Suppose that there exist x_1 and x_2 such that $x_1 \neq x_2$, and $f(x_1) = f(x_2)$. There must exist an object x_3 between x_1 and x_2 such that $f(x_3) > f(x_2)$. Otherwise, assuming that the target lies on the simple path from x_1 to x_2, we would ask about x_1 and x_2 for the same sequence of previous questions and answers, which is not possible. Furthermore, values of the function f belong to the set $\{0, 1, \ldots, m\}$, and in the edge query model, $f(e) > 0$ holds for each edge e, since to discriminate ends of e we need to ask about e. Hence f is a strategy function. ■

The constructive proof of the following lemma shows how to transform a strategy function into a strategy.

Lemma 7 *For each strategy function f on a given tree T there is a strategy that asks no more than $\mathcal{M}_f(T)$ questions in the worst case.*

Proof We will show a strategy described by f. At any given time a set of possible targets induces a subtree. Let $T' = (V', E')$ be a subtree to which the target belongs. In the beginning $T' = T$. As long as $|V'| > 1$, we find a unique object x such that $f(x) = \mathcal{M}_f(T')$, and ask about it. If the answer is "it is here" (which is

possible only in the vertex query model), then x is the target; otherwise, we replace T' by one of its subtrees (which we get by removing x), and the maximum of f in the new T' is by definition lower. Hence, until $\mathcal{M}_f(T') = 0$, with every question either $\mathcal{M}_f(T')$ decreases by at least one or the solution is found, and if $\mathcal{M}_f(T') = 0$, then $|V'| = 1$, and the solution is found. Therefore, we do not ask more than $\mathcal{M}_f(T)$ questions in the worst case. ■

We have shown that to determine an optimal strategy it suffices to compute a strategy function with the lowest maximum.

2.2 Visibility

Definition 8 Let f be a strategy function defined on a tree T, and v be a vertex in T. Say an object x is *visible* from v, if on the simple path from v finishing with x there is no object y such that $f(y) > f(x)$. If x is visible from v, say that value $f(x)$ is also *visible* from v.

Less formally, the visible values of f are those which are not screened by greater values of f. Note also that by the definition of a strategy function no two different objects of the same value can be visible from a vertex v. Therefore, from a given vertex v, each value of f is visible at most once.

Definition 9 A sequence of values is a *visibility sequence* from a vertex v, if it enumerates all values visible from v in descending order.

We will make use of the standard lexicographic order on visibility sequences. However, we will also use another representation of visibility sequences. Namely, we will encode them as natural numbers. In the vertex query model consecutive bits, starting from the least significant, will correspond to values 0, 1, 2, etc. For instance, visibility sequence $(5, 4, 2, 0)$ can be represented as $(110101)_2$. Analogously, in the edge query model consecutive bits will correspond to values 1, 2, 3, etc. Note that the standard order on natural numbers provides ordering of encoded visibility sequences identical to the previous lexicographic ordering.

2.3 An extension operator and bottom-up processing

Our input tree is unrooted, but in the beginning we arbitrarily root it. Denote for convenience by $T(v)$ a subtree of T rooted at v, and by $S(v)$ the visibility sequence from v in $T(v)$.

We will compute a strategy function in a bottom-up manner. For each vertex v we first recursively fix strategy function values in subtrees rooted at children of v, and then extend the strategy function to the whole $T(v)$, fixing values of f wherever it has not been done yet. Say that *an extension operator* is a procedure responsible for extension of the strategy function. An extension operator $\mathcal{O}[\ldots]$ for vertex v of children v_1, v_2, …, v_k takes $S(v_1)$, $S(v_2)$, …, $S(v_k)$, fixes new values of f, and returns newly created $S(v) = \mathcal{O}[S(v_1), S(v_2), \ldots, S(v_k)]$. Note that it suffices to know visibility sequences $S(v_i)$ to correctly construct the strategy function, as all other values are screened, and do not interfere with newly fixed values.

In both considered models we will show suitable extension operators, which will be monotone and minimizing. We say that an extension operator is *monotone* if by increasing sequences in children, we do not decrease the visibility sequence which the operator computes for their parent. We say that an operator is *minimizing* if for given sequences in children the operator produces the minimum visibility sequence $S(v)$.

Lemma 10 *If an extension operator $\mathcal{O}[\ldots]$ is monotone and minimizing, then the bottom-up algorithm calculates a strategy function f for which $\mathcal{M}_f(T)$ is minimum.*

Proof First, we will show that the sequence computed with this operator for the root is minimum. This can be easily proven by induction on the height of the tree. Suppose that sequences computed for children of v are minimum. It follows that they are not greater than corresponding sequences for any other strategy function, and since the operator is monotone, we get the minimum sequence $S(v)$ which it can compute. But as this operator is minimizing, there is no smaller visibility sequence $S(v)$ for any strategy function.

Note that when the visibility sequence at the root is minimum, so is $\mathcal{M}_f(T)$, because the visibility sequence from the root always contains $\mathcal{M}_f(T)$ and no greater values. ■

2.4 Questions about vertices

2.4.1 An extension operator

We will define an extension operator $\mathcal{V}$. Let v be a vertex of children v_i, where $i = 1, \ldots, k$. Extending strategy functions on $T(v_i)$ to a strategy function on $T(v)$, we only need to fix $f(v)$. Let m be the maximum of values that belong to at least two distinct $S(v_i)$, or -1 if there are no such values. Fix $f(v)$ to be the lowest integer greater than m that does not belong to any sequence $S(v_i)$.

Note that defining $f(v)$ this way, we get a strategy function on $T(v)$. Because $f(v)$ does not belong to any $S(v_i)$, it will not interfere with any other value, and the same values visible from different subtrees will not interfere with each other as they are separated by $f(v)$.

Lemma 11 *Operator $\mathcal{V}$ is minimizing and monotone.*

Proof It is clear that f_1, fixed by $\mathcal{V}$ to be the value of f at v, is the lowest value that $f(v)$ may take and still yield a correct strategy function on $T(v)$. Take another value f_2 which correctly extends the strategy function and is greater than f_1. Let s_1 or s_2 be the visibility sequence $S(v)$, when $f(v)$ equals f_1 or f_2, respectively. To finish the proof that the operator is minimizing it suffices to show that $s_1 \leq s_2$. All values greater than f_2 are the same in both s_1 and s_2, since $f(v)$ does not screen them. Furthermore, f_2 belongs to s_2, but does not belong to s_1, since f_2 differs from all values in all sequences $S(v_i)$, and $f_1 < f_2$. Hence $s_1 < s_2$.

Now we only need to show monotonicity. It suffices to consider increasing just one sequence $S(v_i)$. We get the general case, increasing them one by one. Moreover, without loss of generality we assume that we modify the first sequence, and that it has two variants t_1 and t_2, where $t_1 < t_2$. Let a be the greatest value that belongs to t_2, and does not belong to t_1. Such a value exists, and all values in sequences t_1 and t_2 greater than a are the same. Furthermore, let f_1 and f_2 be the values of $f(v)$ which the operator fixes, and s_1 and s_2 be the resulting visibility sequences $S(v)$ for, respectively, $S(v_1) = t_1$ and $S(v_2) = t_2$. Our goal is to show that $s_1 \leq s_2$. Consider four cases:

Case 1 $a > f_1$ and $a \in s_1$.

Obviously, $a \in S(v_i)$ for some $i > 1$, and therefore $f_2 > a$ to separate these two values of a. Sequences s_1 and s_2 contain the same values greater than f_2. Sequence s_2 contains f_2, but s_1 does not. Hence $s_1 < s_2$.

Case 2 $a > f_1$ and $a \notin s_1$.

Sequence s_2 contains a value greater than or equal to a, and the values in s_1 and s_2 greater than this value are the same. This implies that $s_1 < s_2$.

Case 3 $a = f_1$.

If $f_2 > a$, then $f_2 \notin s_1$, $f_2 \in s_2$, s_1 and s_2 share values greater than f_2, and therefore, $s_1 < s_2$. If $f_2 < a$, then $s_1 < s_2$ as well, because s and t contain the same values not less than a, but s_1 does not contain any values less than a, while s_2 does.

Case 4 $a < f_1$.

Replacing t_1 by t_2, we do not decrease the greatest value occurring in at least two $S(v_i)$, which in turn is not less than a. Hence $f_1 = f_2$, and $s_1 = s_2$.

Whichever case we consider, the required $s_1 \leq s_2$ holds. ■

2.4.2 A fast implementation of the extension operator

There is always a query which reduces the size of the set of potential targets by at least half. To show this, start at an arbitrary vertex, and as long there is an edge leading to more than half of potential targets, follow it. This procedure stops at some point, otherwise we would traverse some edge in both directions, which would mean that more than half of the potential targets lie on both sides of this edge, an impossibility. The vertex at which we stop is a suitable query point, and any correct answer to the question about it reduces the set of potential targets by at least half.

This implies in turn that the optimum $\mathcal{M}_f(T)$ is never greater than $\lfloor \log_2 n \rfloor$. Assuming that the number of bits in a word is of order $\Omega(\log n)$, which is true if we can represent n, we use the aforementioned representation of visibility sequences as integers, and a single visibility sequence fits into a constant number of words (without loss of generality into a single word).

We will show how to compute the extension operator for a given vertex in time linear in the number of children. Let $\oplus$ and $\otimes$ be respectively bitwise addition and multiplication operators. Assume also that we can find in constant time the greatest and lowest set bits in a word. If we do not have such operations, we can precompute the required results in $O(n)$ time.

First, we calculate the following values:

$$a_i = \begin{cases} 0 & \text{for } i = 0, \\ a_{i-1} \oplus S(v_i) & \text{for } 1 \leq i \leq k; \end{cases}$$

$$b_i = \begin{cases} 0 & \text{for } i = 0, \\ b_{i-1} \oplus (a_{i-1} \otimes S(v_i)) & \text{for } 1 \leq i \leq k. \end{cases}$$

Clearly, a_i is the sequence of all values occurring in sequences $S(v_1)$ to $S(v_i)$, and b_i is the sequence of all values occurring at least twice in $S(v_1)$ to $S(v_i)$. In constant time we determine the greatest set bit m of b_k, and using logical shifts, also the lowest unset bit of a_k greater than m. This is the value of f at v. To compute $S(v)$ we take a_k, set the bit corresponding to $f(v)$, and clear all less significant bits. In total, we can apply the operator in $O(k)$ time, which gives $O(n)$ running time to compute the strategy function for the whole tree.

2.4.3 Decision tree construction in $O(n)$ time

Having computed the strategy function, we want to construct the corresponding decision tree. This can be easily done in $O(n \log n)$ time, but we will show how to do it in linear time.

Each edge connects two vertices of different values of the strategy function, and it can be an answer in exactly one situation in the decision tree, namely when we ask about the endpoint of larger value (which happens only once in the whole decision tree). Therefore, it suffices that for each edge e, we consider the situation in which it is an answer, and determine a vertex w which is either the next query point, or the target if we already know it. From this information one can easily construct the decision tree in linear time. Denote this vertex w by $q[e]$ for an edge e.

We traverse the tree in a DFS manner, starting from the root. In nonempty entries of a global array $W[0 \dots \lfloor \log_2 n \rfloor]$ we store pairs of a vertex and an edge. If at some point $W[i] = (v, e)$, it means that a value $i = f(v)$ is visible, v belongs to the already visited part of the tree, and e is the edge outgoing from v toward our current position. When we write $W[i].v$ and $W[i].e$, we refer to the first and second element of pair $W[i]$, respectively. To traverse the tree we use a procedure $visit(v, \hat{f})$, where v is a current vertex, and $\hat{f}$ is the value of f at the parent of v. For the root we assume that $\hat{f} = \infty$.

$visit(v, \hat{f})$:
1: • Memorize $W[0 \dots f(v)]$.
2: • Empty $W[0 \dots f(v)]$.
3: • For each child w of v:
4: ◇ $W[f(v)] := (v, (v, w))$
5: ◇ $visit(w, f(v))$
6: ◇ If $\exists x \in S(w)$ such that $x < f(v)$, then:
7: ○ $y :=$ the greatest such x
8: ○ $q[(v, w)] := W[y].v$
9: ◇ For each $x \in S(w)$ such that $x < f(v)$ empty $W[x]$.
10: • Restore the memorized contents $W[0 \dots f(v)]$.
11: • If $\hat{f} \neq \infty$, $W[f(v)] := (v, (v,$ parent of $v))$.
12: • For each pair $x, y \in S(v)$ s.t. $x < y < \hat{f}$, and $x + 1, \dots, y - 1 \notin S(v)$:
13: ◇ $q[W[y].e] := W[x].v$

Now we will show that the procedure *visit* calculates previously defined q, and that each $q[e]$ is set exactly once. Note first that W is correctly modified at each step of the computation. Let v and w be the endpoints of an edge e, where v is the parent of w in the tree. As we will see, depending on the relation between $f(v)$ and $f(w)$, $q[e]$ is set exactly once, either in Step 8 or in Step 13.

If $f(v) > f(w)$, then $q[e]$ is fixed right after the execution of *visit* for w in Step 8, and it becomes the vertex of largest value of f less than $f(v)$ which is visible from w in $T(w)$, that is it becomes exactly what we want. Later this edge is removed from W, and does not appear anymore, so its $q[e]$ is not set for the second time.

Suppose now that $f(v) < f(w)$. Let u be the first vertex on the path from v to the root such that the value of f for the parent of u is greater than $f(w)$ (we assume here that the root has a virtual parent of value ∞). The value of $q[e]$ is set at the end of the execution of *visit* for u in Step 13 to be the vertex visible from u in $T(u)$ of the greatest value less than $f(w)$. This vertex also has the greatest visible value less than $f(w)$, when looking from w toward v. Therefore, we correctly fix $q[e]$. The edge is removed from W in Step 9 in the *visit* execution for the parent of u, and therefore $q[e]$ is not set anymore.

We only need to prove that the whole procedure works in linear time. Determination of greatest and lowest values can be done in $O(1)$ time, using tricks similar to those during computation of the strategy function. In Step 9 each vertex is removed at most once, and in Step 12 and Step 13 each vertex is taken as a value of some $q[e]$ at most once. To finish the proof it suffices to show that the amount of computation in

Steps 1, 2, and 10 is of order $O(n)$, which we do in the lemma below. The total work in these steps, omitting constant terms, which sum to something of order $O(n)$, is proportional to the sum of $f(v)$ over all vertices.

Lemma 12 *Let f be the strategy function computed using $\mathcal{V}$. It holds that*

$$\sum_{v \in V} f(v) \leq 3n.$$

Proof We will trace the function as it is created by the extension operator, and we will show that we can pay $f(v)$ tokens for each vertex v, even if every vertex contributes only 3 tokens. Furthermore, each vertex v, after we pay the value of f at it, keeps a potential of $f(v)$ as long as it is visible on the path to the root. Let v be a vertex to which we apply the extension operator. If $f(v) \leq 1$, then 3 tokens suffice to both pay $f(v)$, and keep a potential of $f(v)$. Suppose then that $f(v) \geq 2$. From the method of computing $f(v)$ we deduce that there is a vertex u_1 visible from a child of v such that $f(u_1) = f(v) - 1$. Moreover, there must also be another visible vertex u_2 such that $f(u_2) = f(v) - 2$, or $f(u_2) = f(v) - 1$. These vertices will no longer be visible, as $f(v)$ will screen them, and therefore, we may safely take over their potentials. Adding 3 tokens contributed by v, we get at least $2f(v)$, which suffices both for payment of $f(v)$ and for the potential of v. ■

2.5 Questions about edges

2.5.1 Calculating the number of necessary questions

Let v be a vertex with children v_1, v_2, …, v_k, and e_1, e_2, …, e_k be edges connecting v with consecutive children. As before we assume that we have already computed strategy functions for each $T(v_i)$, and know all $S(v_i)$. We want to fix values $f(e_i)$ for $1 \leq i \leq k$. We will first show how to calculate the minimum number of questions that can be asked, or to be more precise, we will show how to recursively test whether it is possible to set values of the strategy function at all e_i in such a way that $\mathcal{M}_f(T(v)) \leq q$ for a given q. That is, we test whether we can bound the values of f by q. Denote this test by $\mathit{bound}_q[s_1, s_2, \ldots, s_k]$, for $s_i = S(v_i)$. Let t be equal to the greatest element in all sequences s_i, or to 0 if all sequences s_i are empty.

First, a few simple border cases, to which all others will be eventually reduced:

- Test $\mathit{bound}_q[]$ (i.e. $k = 0$) succeeds.
- Test $\mathit{bound}_0[s_1, s_2, \ldots, s_k]$ fails for $k > 0$.
- Test $\mathit{bound}_q[s_1, s_2, \ldots, s_k]$ fails for $q < t$, as there is already an edge e such that $f(e) = t > q$.
- Test $\mathit{bound}_q[s_1, s_2, \ldots, s_k]$ succeeds for $q \geq k + t$, since we can assign $f(e_i) := q - i + 1$. This gives us a correct strategy function, since values $f(e_i)$ are distinct, and large enough to keep smaller values in different subtrees $T(v_i)$ from interfering with each other.

Next we consider two recursive cases in which both k and q are positive. We may assume without loss of generality that $s_1 = \max_i s_i$.

- If $t = q$, and $k, q > 0$, then $\mathit{bound}_q[s_1, s_2, \ldots, s_k]$ succeeds if and only if $\mathit{bound}_{q-1}[s_1^\star, s_2, \ldots, s_k]$ succeeds, where $s_1^\star$ is the sequence s_1 with the greatest element removed (s_1 is nonempty as $t > 0$).

  In this case we do not really have a choice. We would like $\mathcal{M}_f(T(v))$ to be less than or equal to q, but there is already an edge e such that $f(e) = q$, so the first question has to be about this edge. Hence

the test succeeds if and only if we can determine the target, asking at most $q-1$ questions in the tree that is left when we remove the subtree connected by e to the rest of $T(v)$.

- If $t < q < k+t$, and $k, q > 0$, then *bound*$_q[s_1, s_2, \ldots, s_k]$ succeeds if and only if test *bound*$_{q-1}[s_2, \ldots, s_k]$ succeeds.

  We ask the first question about edge e_1, so we assign $f(e_1) := q$. It is trivial that if test *bound*$_{q-1}[s_2, \ldots, s_k]$ succeeds, then *bound*$_q[s_1, s_2, \ldots, s_k]$ succeeds as well. The other direction is much less obvious. First of all, if we can remove a subtree, we should do it, as it is not harder to find a strategy function with a maximum $\leq q-1$ in the subtree. Suppose that by cutting out some subtree with a smaller visibility sequence, we eventually succeed. We may assume without loss of generality that there is an extension f' of functions on subtrees $T(v_i)$ to $T(v)$, such that $f'(e_2) = q$, $s_2 < s_1$, and $\mathcal{M}_{f'}(T(v)) = q$. Let $a = f'(e_1)$, and b be the greatest number that belongs to s_1, but does not belong to s_2. Number b always exists, since $s_2 < s_1$. If $a > b$, we can simply exchange values of $f'(e_1)$ and $f'(e_2)$ achieving a correct strategy function, and the same visible values. Otherwise, $a < b$, so we can modify $f'(e_1)$ to q, and $f'(e_2)$ to b, and achieve an even smaller visibility sequence at v. In general, if $t < q$, and there exists an extension of f to $T(v)$ such that $\mathcal{M}_f(T(v)) \leq q$, then there always exists an extension in which additionally $f(e_1) = q$.

It is easy to check that we have covered all possibilities. Furthermore, the test always stops, since with each equivalence that we use the allowed maximum q of the function values decreases.

Note moreover that the test is *monotone*, that is if q questions suffice for given visibility sequences of children, decreasing any of them will not make the test fail. Let us assume that there is a strategy in which $\mathcal{M}_f(T) \leq q$, and that we decrease s_j for some j. We can change $f(e_j)$ so that $\mathcal{M}_f(T) \leq q$, and f is still a strategy function. To do so, we take as a new $f(e_j)$ the maximum of $f(e_j)$ and the values that disappeared from s_j during its decrease. One can easily check that this modified f remains a correct strategy function.

2.5.2 An extension operator

Now we will describe an extension operator $\mathcal{E}$. We need to fix values of f at edges e_i, $1 \leq i \leq k$. Using test *bound*, we can calculate $q_{\min}$, the lowest $\mathcal{M}_f(T(v))$ for all correct extensions. Assume once again that $s_1 = \max_i s_i$. If $q_{\min} > t$, then we assign $f(e_1) := q$, forget about the whole subtree rooted at v_1, and repeat the same procedure for the smaller tree. Otherwise, if $q_{\min} = t$, there is an edge e such that $f(e) = t$, we forget about the subtree to which e leads—this corresponds to removing the largest element t from s_1—and proceed with the smaller tree. Eventually, we will assign all values $f(e_i)$, and the sequence of $q_{\min}$'s that we determined in consecutive steps will be the visibility sequence from v in $T(v)$. The correctness of the operator, i.e. that it creates a strategy function, is obvious. The following lemma shows that the operator has the required properties.

Lemma 13 *Extension operator $\mathcal{E}$ is minimizing and monotone.*

Proof We will prove these two properties by induction on k, the number of children. If $k = 0$, the situation is trivial, so assume that $k > 0$, and that the properties hold for $k-1$.

First of all, we know that $q_{\min}$, the minimum number of questions that we determine is really minimum. If $q_{\min} > t$, we assign $q_{\min}$ to one of the edges e_i. If we assign it to an edge connecting v to the child of the minimum visibility sequence, we eliminate the greatest visibility sequence, and it follows from the monotonicity of $\mathcal{E}$ for $k-1$ arguments, that we get the minimum possible solution. On the other hand,

as long as $q_{\min} = t$, we do not have a choice. We continue with a smaller tree, and eventually reduce the problem to the previous case when $q_{\min} > t$.

To prove that the operator is monotone, we consider two k-tuples $(s_{11}, s_{12}, \ldots, s_{1k})$, and $(s_{21}, s_{22}, \ldots, s_{2k})$ of visibility sequences such that $s_{1i} \leq s_{2i}$ for each i. If the minimum numbers of questions for these tuples differ, by the monotonicity of test *bound*, we are done. Suppose then that $q_{\min}$ is the same for both tuples. Define t_1 and t_2 for s_{1i} and s_{2i} respectively as we defined t before for s_i. By definition $t_1 \leq t_2$. If $t_1 < q_{\min}$, and $q_{\min} = t_2$, then the monotonicity follows from the monotonicity of the operator for $k-1$ children, and from the fact that if we remove a subtree, we do not increase the visibility sequence. If both t_i are less than $q_{\min}$, the monotonicity follows from the monotonicity for $k-1$ children. There is only one case left that we need to consider. As long as both t_i equal $q_{\min}$, we may assume that the greatest elements in both tuples have the same index. Thus removing the greatest element from them does not affect the $\leq$ relation between them, and eventually we fall back upon one of the previous cases. ■

2.5.3 A fast implementation

To obtain a fast implementation of the algorithm, we need to quickly test whether a given number of questions suffices for given visibility sequences in children. What we do is determined by the greatest element in the greatest sequence. We would like to be able to quickly determine this element. All sequences that can appear during the computation belong to the set of suffixes of visibility sequences in children.

Let v_i, $1 \leq i \leq k$, be the children of the vertex on which we perform the extension operator, and let d be the sum of lengths of visibility sequences in children in the binary representation. Before we conduct any test, we sort all possible suffixes. This can be done in $O(d+k)$ time, using a modification of radix sort, where k is the number of children. In the beginning let L be the list of empty suffixes of $S(v_i)$ (each sequence has the empty suffix), and S be the list of all nonempty $S(v_i)$. For consecutive bits of binary representations of $S(v_i)$, starting from the least significant, we do the following. We sort S according to values of the bit being considered, using stable counting sort. Then collect, in the order on list S, all proper suffixes starting from this bit (i.e. those in which the current bit is set), and add them to the end of L. Finally, we remove from list S sequences that start from this bit, as all suffixes of these sequences are already on list L, and proceed with the next bit, unless S is already empty. When we finish the procedure, list L contains all suffixes of the visibility sequences in ascending order. Additionally, we may assume that every suffix on the list has a link to its longest proper suffix, if it exists. These links (see Figure 4 for an example) can be easily created during the described procedure.

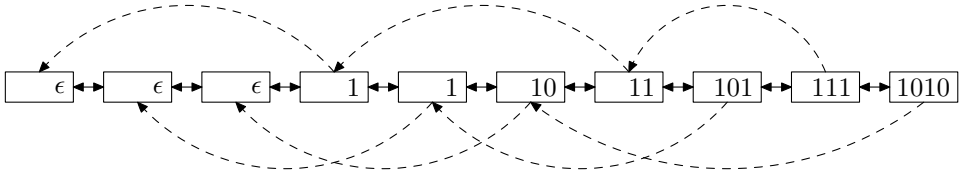

Figure 4. A sample sorted list of suffixes with additional links created for the following visibility sequences in the binary representation: 1010**,** 111**, and** 101**.**

Now we will show how to quickly perform the test, using the structure described above. When we evaluate test *bound*, sequences that are considered at the moment are marked on list L, and the greatest of them is last on the list. Knowing the greatest sequence, we can easily determine which case in evaluation

of test *bound* occurs. If this is one of the border cases, we are done. Otherwise, if we simply remove the greatest sequences, we just unmark it on the list. The only remaining case is when we remove the greatest element from the greatest sequence s. This can be done in $O(1)$ time, using our structure. We first unmark s, and then mark its longest proper suffix, to which we hold a link in s. Since the greatest sequence decreases during the procedure, we just go over list L starting from the greatest element. Eventually, we conduct the test in $O(d + k)$ time.

Knowing how to evaluate test *bound*, we are ready to evaluate the whole extension operator. We perform test *bound* with a lower and lower number of questions, starting with $t + k$ questions, where t is the greatest number in the greatest sequence. This number of questions definitely suffices. We decrease the number of questions as long as it is possible, and when we stop, we know the greatest number in the new visibility sequence. Then we continue with a smaller number of questions for the subtree left for further consideration. We do not conduct more than $2(t + k)$ tests, and hence we can evaluate the extension operator in $O((t + k)(d + k))$ time.

Denote by OPT the optimum number of questions for the whole tree. The length of every visibility sequence that we compute is bounded by OPT. Let k_v be the number of children of vertex v. Obviously, $k_v \leq \mathrm{OPT}$, and therefore, the computation time of $\mathcal{E}$ on vertex v can be bounded by $O(k_v \cdot \mathrm{OPT}^2)$, so the total time spent on computation of the strategy function is bounded by $O(n \cdot \mathrm{OPT}^2)$. Computation of an optimal decision tree in $O(n \cdot \mathrm{OPT})$ time, when we have an optimal strategy function, is trivial. Eventually, we achieve $O(n^3)$ time complexity. Furthermore, for bounded degree trees, we have $\mathrm{OPT} = O(\log n)$ with a running time of order $O(n \log^2 n)$.

3 Searching in forest-like posets

3.1 Forests of type A

For trees of type A the problem is identical to the problem of searching in trees in the edge query model. The question about an element e in a poset corresponds to a question about the edge connecting e with its parent in the tree diagram of this poset. Hence we can compute an optimal decision tree in $O(n^3)$ time, which improves the $O(n^4 \log^3 n)$ bound shown by Ben-Asher, Farchi, and Newman [1]. Moreover, for bounded degree trees we achieve a running time of $O(n \log^2 n)$, improving the previous upper bound of $O(n \log^4 n)$.

Our algorithm easily generalizes to forests. We add a virtual vertex $v^\star$, a parent of roots of all trees in the diagram, and fix values of the strategy function on all but one of the edges connecting $v^\star$ with its children, as we do not need to distinguish $v^\star$ from one of the roots of the trees. This slightly, but not significantly, changes the way in which the extension operator and test *bound* work for $v^\star$. We still get a running time of $O(n^3)$.

3.2 Forests of type B and C

3.2.1 An algorithm

Now we will show a simple greedy algorithm that for forest-like posets of type C constructs decision trees of height at most 1 more than the optimal. Let M be the set of all maximal elements in our input poset. First we determine a special ordering of elements in M in which we will query them. We deconstruct the whole forest as follows. We find a vertex $e \in M$ such that the set of elements reachable in the diagram from e is the largest. We remove these elements. Note that they constitute on their own a tree-like poset of type A. We continue the whole procedure until no element is left. Elements removed in each round form a single

part. For each part we compute an optimal decision tree, using the algorithm for posets of type A. Next, by counting sort we sort them in linear time, primarily in descending order according to the number of queries in the optimal strategy, and secondarily in descending order according to the number of elements. Then we create a decision tree that corresponds to the following procedure. We query elements of M in the order in which they occur on the list of sorted parts until we find $m \in M$ such that $x \leq m$, where x is the target. Such an m obviously exists, and having found it, we locate the target in the part which m belongs to, using the corresponding optimal decision tree.

Note that the target can be in less than or equal to more than one element in set M. If it turns out that the first element in M greater than or equal to the target is m, then how do we know that the target belongs to the part associated with m (denote this part by A)? Suppose to the contrary that it does not. It means that it belongs to some other part B that was removed earlier in the process of partitioning. Part A is smaller than B in the number of vertices. If $|A|$ were greater than or equal to $|B|$, then the vertex from M in A would have been removed first, since that vertex with all reachable elements, including the target, would have constituted a set of size greater than $|A|$. Part A on its own is a total order, or in other words, it has a diagram that is a directed path. Note that the optimal worst-case number of questions for A is $q_A = \lceil \log_2 |A| \rceil$. We cannot determine an element in B faster than in $\lceil \log_2 |B| \rceil$ queries in the worst case, which is not less than q_A, since $|A| < |B|$. Therefore, part B appears before part A on the sorted list of parts, and there was an element from M before m, which was greater than or equal to the target. This is a contradiction, proving the correctness of our algorithm.

3.2.2 Running time

The poset can be partitioned in linear time. In each tree we run from its connecting element a DFS algorithm that moves in the direction opposite to the direction of the edges. This way we only visit vertices representing elements greater than or equal to the connecting element. For each such vertex v, we recursively compute the length of the longest directed path incoming to it. Consider W, the set of all vertices w such that there is a directed edge from w to v. To compute the length of the longest path for v, we take the maximum of the lengths over W, and add one. Furthermore, at each visited vertex v, we choose a vertex $w^\star \in W$ of the longest incoming path, and remove all the edges incoming to v except the edge from $w^\star$. One can easily check that this gives us the required partition of the diagram.

In general, we can compute the decision tree for type C forest-like posets in $O(n^3)$ time, spending most of the time on computation of optimal decision trees for each part. It turns out that in the case of type B posets, all parts are directed paths, for which we can compute optimal decision trees corresponding to standard binary search in linear time. Therefore, for forest-like posets of type B, our algorithm achieves a running time of $O(n)$.

3.2.3 Almost optimality

Theorem 14 *The greedy algorithm presented in Section 3.2.1 constructs a decision tree of height at most* $\mathrm{OPT}+1$*, where* OPT *is the optimal worst-case number of questions for an input partial order of type C.*

Proof Our plan is to take an optimal strategy, and turn it into the strategy constructed by our algorithm in such a way that it is obvious that we have not exceeded $\mathrm{OPT}+1$ queries. We can assume that the optimal strategy has the following properties:

- After a negative response to the question whether $x \leq e$, all elements y such that $y \leq e$ are removed from the partial order, and the strategy is an optimal strategy for the new partial order.

- After a positive response to this question, the target is found, using an optimal strategy for the set of all elements from the current partial order such that $x \leq e$. Note that this set is always a tree-like poset of type A.

Let $e_1^\star, e_2^\star, \ldots, e_r^\star$ be the sequence of elements that we query in the optimal strategy as long as the responses that we get are negative, and S_i, $i = 1, \ldots, r$, be the partial order that we consider after positive answer to the i-th question. There can be at most one element in the poset that does not belong to any of them. Denote by $\mathrm{opt}(S)$ the optimum worst-case number of questions for poset S. It holds that

$$\mathrm{OPT} = \max_{i=1,\ldots,r} (i + \mathrm{opt}(S_i)),$$

and moreover, for each $k = 0, \ldots, r$, we have

$$\mathrm{opt}\left(S - \bigcup_{i=1}^{k} S_i\right) \leq \mathrm{OPT} - k,$$

where S is the input partial order. Furthermore, let L^- be the set of elements less than or equal to some connecting element, and L^+ be the set of elements greater than or equal to some connecting element.

We will work on a sequence of elements e_i. All considered sequences will satisfy the condition that each element in the poset is less than or equal to at least one e_i. Sequence e_i describes a strategy in the following way. We query whether $x \leq e_i$ for consecutive e_i, removing all elements less than or equal to e_i if the answer is negative, and using an optimal strategy for elements less than or equal to e_i that have not been removed yet if the answer is positive. The sequence of e_i generates a partition of elements.

We start with $e_i = e_i^\star$, $1 \leq i \leq r$, and if there exists an element e such that $e \leq e_i^\star$ does not hold for any $e_i^\star$, we extend the sequence, letting e_{r+1} be this element. Obviously, the strategy described by this sequence does not exceed $\mathrm{OPT} + 1$ questions in the worst case.

Now we begin modification of the sequence, doing the following for each tree in the forest-like poset. Let c be the connecting element. Element c belongs to the i-th part generated by the sequence of e_i for some i. If there is no e_j such that $e_j \leq c$ for $j < i$, we are done, and proceed to the next tree. Otherwise, assume that j is the minimal such j. We replace current e_j with c, and possibly shorten the sequence, removing all elements of indices $k > j$ such that $e_k \leq c$. Note that this modification does not make the number of questions greater than $\mathrm{OPT} + 1$. Let L_c be the poset of elements less than or equal to c. In the original optimal strategy, e_j was at an index not less than j, say $j^\star$. It follows that

$$\begin{aligned} j + \mathrm{opt}(L_c) &\leq j + \mathrm{opt}\left(S - \bigcup_{i=1}^{j^\star - 1} S_i\right) \\ &\leq j + \mathrm{OPT} - (j^\star - 1) \\ &\leq \mathrm{OPT} + 1, \end{aligned}$$

since we can use any strategy for a subset of S containing L_c to find the target in L_c in at most the same number of questions. As the result of this modification, each pair of elements from L^- belongs to the same part generated by the sequence of e_i if and only if it belongs to the same part generated by our algorithm.

The second part of the sequence modification follows. We will consider all elements in L^+ in arbitrary topological ordering inferred by the forest-like diagram of the poset. The following invariants will hold during the procedure:

1. The worst-case number of questions in the strategy described by the sequence of e_i is at most $\mathrm{OPT}+1$.
2. Two elements in L^+ that have been already considered belong to the same part in the partition generated by sequence e_i if and only if they belong to the same part in the partition generated by our algorithm.
3. If an element $e \in L^+$ has not been considered yet, and it belongs to the i-th part generated by sequence e_i, and it belonged to the j-th part generated by the optimal strategy, then $i \leq j$.

In the beginning the invariants trivially hold. Suppose that they hold after we have already considered some elements, and that e is the next element to be considered. In the partition generated by the algorithm, it belongs to a part generated by some element s. There are two cases:

- If e and s belong to the same part generated by sequence e_i, we are done. Invariants 1 and 3 obviously still hold. If $e \neq s$, Invariant 2 also holds due to the transitive property of belonging to the same part. Otherwise, if $e = s$, e is maximal and is not less than or equal to any element considered so far. Hence as we expect it belongs to a different part than any of these elements, and therefore, Invariant 2 holds.
- If e and s do not belong to the same part generated by sequence e_i, then $e \neq s$. Element e belongs to the j-th part generated by the sequence of e_i for some j. It follows from Invariant 2 that in the strategy described by the current sequence e_i, s generates a chain of elements less than or equal to s, but greater than e. Once again we consider two cases:
  - If $e_j = e$, we put s at index j instead of e_j, and shorten the sequence, removing s from its previous position. Invariant 3 still holds, and Invariant 2 follows as before from the transitive property. By Invariant 2 the whole part generated by s in the partition generated by the new sequence e_i was a subset of $S - \bigcup_{i=1}^{j-1} S_i$, and therefore, if the target belongs to this part, we ask at most

$$\begin{aligned} j + \mathrm{opt}\left(S - \bigcup_{i=1}^{j-1} S_i\right) &\leq j + \mathrm{OPT} - (j-1) \\ &= \mathrm{OPT} + 1 \end{aligned}$$

    questions, and Invariant 1 still holds.
  - If $e_j \neq e$, we find a subsequence e_{a_i} of sequence e_i that consists of all elements t from e_j to s such that $e \leq t$. The subsequence has q elements for some q, and by definition $a_1 = j$, and $e_{a_q} = s$. We perform a cyclic shift, putting e_{a_1} in the place of e_{a_2}, e_{a_2} in the place of e_{a_3}, and so forth, until we put $e_{a_{q-1}}$ in the place of e_{a_q}, and finally, e_{a_q} in the place of e_{a_1}. Denote elements of the new sequence by e'_i to distinguish the modified sequence from the original. The cyclic shift takes the part generated by e_j, disconnects all elements greater than e from this part, creates a separate part from them, and connects the remaining part to the part generated by s. This way Invariant 2 holds after consideration of e. The index of the part to which an element belongs increases only for some elements greater than e, i.e. only for these that have already been considered, and therefore Invariant 3 still holds.

    We only need to show Invariant 1. It follows from the algorithm that parts generated by e'_{a_2} to e'_{a_q} in the new sequence are total orders consisting of at most the same number of elements as in the part generated by $e_{a_q} = s$ in the old sequence. This means that if the target belongs to the

parts generated by one of e'_{a_2} to e'_{a_q} in the new sequence, we do not exceed $\mathrm{OPT}+1$ questions in the worst case in the new strategy, as the old strategy would exceed $\mathrm{OPT}+1$ questions in the worst case as well, which by the induction hypothesis is not true. To finish the proof that Invariant 1 holds, we should show that if the target belongs to the part generated by e'_{a_1} we do not ask more than $\mathrm{OPT}+1$ questions in the new strategy. The proof of this fact is analogous to the proof for the case of $e_j = e$.

Eventually, each two elements in L^+ belong to the same part generated by the strategy that we get if and only if they belong to the same part in the partition generated by the algorithm. The same claim still holds for L^-, and since these sets are connected by the connecting elements, this holds for each pair of elements. The worst-case number of questions in the strategy that we have created is no greater than $\mathrm{OPT}+1$. However, the only difference from the strategy generated by our algorithm is that our algorithm sorts these parts in descending worst-case number of questions to locate an element in them. Therefore, the strategy generated by the algorithm has a worst-case number of questions of at most $\mathrm{OPT}+1$ as well. ■

4 Acknowledgments

The authors would like to thank Krzysztof Diks for inspiring discussions, Ronitt Rubinfeld for useful comments on the first write-up and Kevin Matulef for his invaluable help with the final version.

References

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

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

[3] 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.

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

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

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

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

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

[9] 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.

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