SIAM J. COMPUT.
Vol. 40, No. 3, pp. 597–622
© 2011 Society for Industrial and Applied Mathematics

SORTING AND SELECTION IN POSETS*

CONSTANTINOS DASKALAKIS†, RICHARD M. KARP‡, ELCHANAN MOSSEL§, SAMANTHA J. RIESENFELD¶, AND ELAD VERBIN‖

Abstract. Classical problems of sorting and searching assume an underlying linear ordering of the objects being compared. In this paper, we study these problems in the context of partially ordered sets, in which some pairs of objects are incomparable. This generalization is interesting from a combinatorial perspective, and it has immediate applications in ranking scenarios where there is no underlying linear ordering, e.g., conference submissions. It also has applications in reconstructing certain types of networks, including biological networks. Our results represent significant progress over previous results from two decades ago by Faigle and Turán. In particular, we present the first algorithm that sorts a width-w poset of size n with query complexity $O(n(w + \log n))$ and prove that this query complexity is asymptotically optimal. We also describe a variant of Mergesort with query complexity $O(wn \log \frac{n}{w})$ and total complexity $O(w^2 n \log \frac{n}{w})$; an algorithm with the same query complexity was given by Faigle and Turán, but no efficient implementation of that algorithm is known. Both our sorting algorithms can be applied with negligible overhead to the more general problem of reconstructing transitive relations. We also consider two related problems: finding the minimal elements, and its generalization to finding the bottom k "levels," called the *k-selection* problem. We give efficient deterministic and randomized algorithms for finding the minimal elements with query complexity and total complexity $O(wn)$. We provide matching lower bounds for the query complexity up to a factor of 2 and generalize the results to the k-selection problem. Finally, we present efficient algorithms for computing a linear extension of a poset and computing the heights of all elements.

Key words. chain decomposition, partially ordered sets, query complexity, selection, sorting, transitive relations

AMS subject classifications. 68, 05

DOI. 10.1137/070697720

1. Introduction.

Sorting is a fundamental and, by now, well-understood problem in combinatorics and computer science. Classically, the problem is to determine the structure of a totally ordered set, and *comparison algorithms*, in which direct

*Received by the editors July 19, 2007; accepted for publication (in revised form) January 3, 2011; published electronically May 4, 2011. A previous, limited version of this article appeared in *Proceedings of the Twentieth ACM-SIAM Symposium on Discrete Algorithms*, 2009, pp. 392–401.
http://www.siam.org/journals/sicomp/40-3/69772.html

†EECS and CSAIL, MIT, Cambridge, MA 02139 (costis@csail.mit.edu). This research was done while this author was a student at UC Berkeley, supported by a Microsoft Research Fellowship and NSF grant CCF-0635319.

‡Department of Computer Science, UC Berkeley, Berkeley, CA 94720, and International Computer Science Institute, Berkeley, CA 74704 (karp@icsi.berkeley.edu). This author's work was supported by NSF grant CCF-0515259.

§Department of Statistics and Computer Science, UC Berkeley, Berkeley, CA 94720, and Department of Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel (mossel@stat.berkeley.edu). This author's work was supported by DOD grant N0014-07-1-05-06, NSF grants DMS 0528488 and DMS 0548249, and a Sloan Fellowship in Mathematics.

¶The J. David Gladstone Institutes, University of California, San Francisco, CA 94158 (samantha.riesenfeld@gladstone.ucsf.edu). This author's work was supported in part by a grant from the Gordon & Betty Moore Foundation to Katherine S. Pollard. Part of this research was done while this author was a student at UC Berkeley, supported by NSF grant CCF-0515259.

‖Institute for Theoretical Computer Science, Tsinghua University, Beijing 10084, China (eladv@tsinghua.edu.cn). Part of this research was done while this author was a student at Tel Aviv University. This author's work was supported by National Natural Science Foundation of China grant 60553001 and National Basic Research Program of China grants 2007CB807900 and 2007CB807901.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

comparisons between pairs of elements are the only means of acquiring information about the linear order, form an important subclass. Usually, sorting algorithms are evaluated by two complexity measures: *query complexity*, the number of comparisons performed, and *total complexity*, the number of basic computational operations of all types performed.

The generalization of the sorting problem to partially ordered sets, or *posets*, has been considered in the literature (see, e.g., Faigle and Turán [9] and our discussion of related work in section 1.2). However, sorting problems appear to be more intricate for partial orders, which may explain why there has not been an account of a query-optimal (possibly inefficient) sorting algorithm.

In this paper we make significant progress on the problem of sorting posets and related problems. In particular, we provide the first asymptotically query-optimal algorithm for sorting and give an efficient algorithm which matches the query complexity of the algorithm by Faigle and Turán (no efficient implementation of which is known). Moreover, we provide upper and lower bounds for the problem of finding the minimal elements of a poset and its generalization to selecting the bottom k-layers, which asymptotically match for $k = 1$. Our algorithms gather information about the poset by querying an oracle; the oracle responds to a query on a pair of elements by giving their relation or a statement of their incomparability.

Apart from having an interesting combinatorial structure, the generalization of sorting to posets is useful for treating ranking scenarios where certain pairs of elements are incomparable. Examples include ranking conference submissions, strains of bacteria according to their evolutionary fitness, and points in R^d under the coordinate-wise dominance relation. Note that a query may involve extensive effort (for example, running an experiment to determine the relative evolutionary fitness of two strains of bacteria). Hence, query complexity may be just as important as total complexity.

A partial order on a set can be thought of as the reachability relation of a directed acyclic graph (DAG). More generally, a transitive relation (which is not necessarily irreflexive) can be thought of as the reachability relation of a general directed graph. In applications, the relation represents the direct and indirect influences among a set of variables, processes, or components of a system, such as the input-output relations among a set of metabolic reactions or the causal influences among a set of interacting proteins. We show that, with negligible overhead, the problem of sorting a transitive relation reduces to the problem of sorting a partial order. Our algorithms thus allow one to reconstruct general directed graphs, given an oracle for queries on reachability from one node to another. As directed graphs are the basic model for many real-life networks including social, information, biological, and technological networks (see, e.g., [17]), our algorithms provide a potential tool for the reconstruction of such networks.

Partial orders often arise in the classical sorting literature as a representation of the "information state" at a general step of a sorting algorithm. In such cases the incomparability of two elements simply means that their true relation has not been determined yet. The present work is quite different in that the underlying structure to be discovered is a partial order, and incomparability of elements is inherent, rather than representing temporary lack of information. Nevertheless, the body of work on comparison algorithms for total orders provides insights for the present context (see, e.g., [2, 5, 11, 13, 15, 16]).

1.1. Definitions. To precisely describe the problems considered in this paper and our results, we require some formal definitions. A partially ordered set, or poset,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

is a pair $\mathcal{P} = (P, \succ)$, where P is a set of elements and $\succ \subset P \times P$ is an irreflexive, transitive binary relation. For elements $a, b \in P$, if $(a, b) \in \succ$, we write $a \succ b$ and say that a *dominates* b, or that b is *smaller than* a. If $a \not\succ b$ and $b \not\succ a$, we say that a and b are *incomparable* and write $a \not\sim b$.

A *chain* $C \subseteq P$ is a subset of mutually comparable elements, that is, a subset such that for any elements $c_i, c_j \in C$, $i \neq j$, either $c_i \succ c_j$ or $c_j \succ c_i$. An *ideal* $I \subseteq P$ is a subset of elements such that if $x \in I$ and $x \succ y$, then $y \in I$. The *height* of an element a is the maximum cardinality of a chain whose elements are all dominated by a. We call the set $\{a : \forall b,\ b \succ a \text{ or } b \not\sim a\}$ of elements of height 0 the *minimal* elements. An *antichain* $A \subseteq P$ is a subset of mutually incomparable elements. The *width* $w(\mathcal{P})$ of poset $\mathcal{P}$ is defined to be the maximum cardinality of an antichain of $\mathcal{P}$.

A *decomposition* $\mathcal{C}$ of $\mathcal{P}$ into chains is a family $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ of disjoint chains such that their union is P. The *size* of a decomposition is the number of chains in it. The width $w(\mathcal{P})$ is clearly a lower bound on the size of any decomposition of $\mathcal{P}$. We make frequent use of *Dilworth's theorem*, which states that there is a decomposition of $\mathcal{P}$ of size $w(\mathcal{P})$. A decomposition of size $w(\mathcal{P})$ is called a *minimum chain decomposition*.

1.2. Sorting and k-selection. The central computational problems of this paper are *sorting* and *k-selection*. The sorting problem is to completely determine the partial order on a set of n elements, and the k-selection problem is to determine the set of elements of height at most $k-1$, i.e., the set of elements in the k bottom levels of the partial order. In both problems we are given an upper bound w on the width of the partial order. (But see also section 6 for a relaxation of this assumption.) In the absence of a bound on the width, the query complexity of the sorting problem is exactly $\binom{n}{2}$, in view of the worst-case example in which all pairs of elements are incomparable. In the classical sorting and selection problems, $w = 1$.

Faigle and Turán [9] described two algorithms for sorting posets, which they term "identification" of a poset, both of which have query complexity $O\left(wn \log \frac{n}{w}\right)$. (In fact the second algorithm has query complexity $O(n \log N_{\mathcal{P}})$, where $N_{\mathcal{P}}$ is the number of ideals of the input poset $\mathcal{P}$; it is easy to see that $N_{\mathcal{P}} = O(n^w)$ if $\mathcal{P}$ has width w, and that $N_{\mathcal{P}} = (n/w)^w$ if $\mathcal{P}$ consists of w incomparable chains, each of size n/w.) The total complexity of sorting has not been considered. It turns out that the total complexity of the first algorithm of Faigle and Turán depends on the subroutine for computing a chain decomposition, the complexity of which was not analyzed in their work [9]. Furthermore, it is not clear whether there exists a polynomial-time implementation of the second algorithm.

Boldi, Chierichetti, and Vigna [1] have independently considered the query complexity of a problem related to k-selection: Given a poset and an integer t, find t elements such that the maximum height of the elements in the set is minimized. Their results are not directly comparable to ours, since translating from one setting to the other would require knowledge of the number of elements in each layer of the poset. A recent paper [18] considers an extension of the searching and sorting problem to posets that are either trees or forests.

1.3. Techniques. It is natural to approach the problems of sorting and k-selection in posets by considering generalizations of well-known algorithms for the case of total orders, whose running times are closely matched by proven lower bounds. However, natural generalizations of classic algorithms *do not* provide optimal poset algorithms in terms of the total complexity and query complexity.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

In the case of sorting, the generalization of Mergesort considered here loses a factor of w in its query complexity compared to the lower bound established in Theorem 3.2. Surprisingly, we can match the query complexity lower bound (up to a constant factor) by carefully exploiting the structure of the poset. We do not know whether it is possible to achieve optimal query complexity efficiently, although it is conceivable that approximate-counting techniques similar to those used in Dyer, Frieze, and Kannan [8] could be used to make our query-optimal EntropySort algorithm (described in section 3.1) efficient.

The seemingly easier problem of k-selection still poses interesting challenges, particularly in the proofs of our lower bounds (see, for example, the proofs of Theorems 4.6 and 4.7).

1.4. Main results and paper outline. In section 2, we briefly discuss an efficient representation of a poset. The representation has size $O(wn)$ and permits retrieval of the relation between any two elements in time $O(1)$. In sections 3.1 and 3.2, we give an algorithm for sorting a poset with query complexity $O(n(logn+w))$ and show that this query complexity is asymptotically optimal. We also provide a generalization of Mergesort with query complexity $O(wn \log n)$ and total complexity $O(w^2 n \log n)$. In section 4, we give upper and lower bounds on the query complexity and total complexity of k-selection within deterministic and randomized models of computation. For the special case of finding the minimal elements ($k = 1$), we show that the query complexity and total complexity are $\Theta(wn)$; the query upper bounds match the query lower bounds up to a factor of 2. In section 5, we give a randomized algorithm, based on a generalization of Quicksort, of expected total complexity $O(n(\log n + w))$ for computing a linear extension of a poset. We also give a randomized algorithm of expected total complexity $O(wn \log n)$ for computing the heights of all elements in a poset. Finally, in section 6, we show that the results on sorting posets generalize to the case where an upper bound on the width is not known and to the case of transitive relations.

2. Representing a poset: The ChainMerge data structure. Once the relation between every pair of elements in a poset has been determined, some representation of this information is required, both for output and for use in our algorithms. The simple ChainMerge data structure that we describe here supports constant-time look-ups of the relation between any pair of elements. It is built from a chain decomposition.

Let $\mathcal{C} = \{C_1, \ldots, C_q\}$ be a chain decomposition of a poset $\mathcal{P} = (P, \succ)$. Then ChainMerge$(\mathcal{P}, \mathcal{C})$ stores, for each element $x \in P$, q indices as follows: Let C_i be the chain of $\mathcal{C}$ containing x. The data structure stores the index of x in C_i and, for all j, $1 \le j \le q$, $j \ne i$, the index of the largest element of chain C_j that is dominated by x. The performance of the data structure is characterized by the following lemma.

Claim 2.1. *Given a query oracle for a poset $\mathcal{P} = (P, \succ)$ and a decomposition $\mathcal{C}$ of $\mathcal{P}$ into q chains, building the* ChainMerge *data structure has query complexity at most $2qn$ and total complexity $O(qn)$, where $n = |P|$. Given* ChainMerge$(\mathcal{P}, \mathcal{C})$, *the relation in $\mathcal{P}$ of any pair of elements can be found in constant time.*

Proof. The indices corresponding to chain C_j that must be stored for the elements in chain C_i can be found in $O(|C_i|+|C_j|)$ time, using $|C_i|+|C_j|$ queries, by simultaneously scanning C_i and C_j. The scan begins with the smallest elements of C_i and C_j and moves upward. At each step, it queries whether the current element of C_j dominates the current element of C_i. If so, it stores the index of the element in C_j and considers the next element in C_i; if not, it considers the next element in C_j. Since each chain

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

is scanned $2q-1$ times, it follows that the query complexity of ChainMerge$(\mathcal{P},\mathcal{C})$ is at most $2qn$, and the total complexity is $O(q \cdot \sum_{i=1}^{q} |C_i|) = O(qn)$.

Let $x, y \in P$, with $x \in C_i$ and $y \in C_j$. The look-up operation works as follows: If $i = j$, we simply do a comparison on the indices of x and y in C_i, as in the case of a total order. If $i \neq j$, then we look up the index of the largest element of C_j that is dominated by x; this index is greater than (or equal to) the index of y in C_j if and only if $x \succ y$. If $x \not\succ y$, then we look up the index of the largest element of C_i that is dominated y; this index is greater than (or equal to) the index of x in C_i if and only if $y \succ x$. If neither $x \succ y$ nor $y \succ x$, then $x \not\sim y$. ∎

3. The sorting problem. We address the problem of *sorting a poset*, which is the computational task of producing a representation of a poset $\mathcal{P} = (P, \succ)$, given the set P of n elements, an upper bound w on the width of $\mathcal{P}$, and access to an oracle for $\mathcal{P}$. (See section 6.1 for the case where an upper bound on the width is not known.) The following theorem of Brightwell and Goodall [3] provides a lower bound on the number $N_w(n)$ of posets of width at most w on n elements.

Theorem 3.1 (Brightwell and Goodall [3]). *The number $N_w(n)$ of partially ordered sets of n elements and width at most w satisfies*

$$\frac{n!}{w!}\, 4^{n(w-1)}\, n^{-24w(w-1)} \leq N_w(n) \leq n!\, 4^{n(w-1)}\, n^{-(w-2)(w-1)/2} w^{w(w-1)/2}.$$

Using Theorem 3.1 and our lower bound for finding the minimal elements of a poset, provided in Theorem 4.6, we establish a lower bound on the number of queries required to sort a poset.

Theorem 3.2. *Any algorithm which sorts a poset of width at most w on n elements requires $\Omega(n(\log n + w))$ queries.*

Proof. From Theorem 3.1, if $w = o(\frac{n}{\log n})$, then $\log N_w(n) = \Theta(n \log n + wn)$; hence $\Omega(n(\log n + w))$ queries are required information theoretically for sorting. Theorem 4.6 gives a lower bound of $\Omega(wn)$ queries for finding the minimal elements of a poset. Since sorting is at least as hard as finding the minimal elements, it follows that $\Omega(wn)$ queries are necessary for sorting. For the case $w = \Omega(\frac{n}{\log n})$, $wn = \Omega(n \log n + wn)$. ∎

3.1. A query-optimal sorting algorithm. We describe a sorting algorithm that has optimal query complexity; i.e., it sorts a poset of width at most w on n elements using $\Theta(n \log n + wn)$ oracle queries. Our algorithm is not necessarily efficient, so in section 3.2 we consider efficient solutions to the problem.

Before presenting our algorithm, it is worth discussing an intuitive approach that is different from the one we take. For any set of oracle queries and responses, there is a corresponding set of posets, which we call *candidates*, that are the posets consistent with the responses to these queries. A natural sorting algorithm is to find a sequence of oracle queries such that, for each query (or for a positive fraction of the queries), the possible responses to the query partition the space of posets that are candidates (after the previous responses) into three parts, at least two of which are relatively large. Such an algorithm would achieve the information-theoretic lower bound (up to a constant factor).

For example, the effectiveness of Quicksort for sorting total orders relies on the fact that most of the queries made by the algorithm partition the space of candidate total orders into two parts, each of relative size of at least 1/4. Indeed, in the case of total orders, much more is known: for any subset of possible queries to the oracle,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Algorithm POSET–BININSERTIONSORT($\mathcal{P}$)
input: a set P, an oracle for a poset $\mathcal{P} = (P, \succ)$, an upper bound w on width of $\mathcal{P}$
output: a CHAINMERGE data structure for $\mathcal{P}$

1. $\mathcal{P}' := (\{e\}, \{\})$, where $e \in P$ is an arbitrary element; /* $\mathcal{P}'$ is the current poset*/
2. $P' := \{e\}$; $\mathcal{R}' := \{\}$;
3. $U := P \setminus \{e\}$; /* U is the set of elements that have not been inserted */
4. **while** $U \neq \emptyset$
  a. pick an arbitrary element $e \in U$; /* e is the element to be inserted into $\mathcal{P}'$*/
  b. $U := U \setminus \{e\}$;
  c. find a chain decomposition $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ of $\mathcal{P}'$, with $q \leq w$ chains;
  d. **for** $i = 1, \ldots, q$
    i. let $C_i = \{e_{i1}, \ldots, e_{i\ell_i}\}$, where $e_{i\ell_i} \succ \cdots \succ e_{i2} \succ e_{i1}$;
    ii. do binary search on C_i to find the smallest element (if any) dominating e;
    iii. do binary search on C_i to find the largest element (if any) dominated by e;
  e. based on the results of the binary searches, infer all relations of e with the elements of P';
  f. add into $\mathcal{R}'$ all the relations of e with the elements of P'; $P' := P' \cup \{e\}$;
  g. $\mathcal{P}' = (P', \mathcal{R}')$;
5. find a chain decomposition $\mathcal{C}$ of $\mathcal{P}'$; build CHAINMERGE($\mathcal{P}', \mathcal{C}$) (no additional queries);
6. **return** CHAINMERGE($\mathcal{P}', \mathcal{C}$);

FIG. 3.1. *Pseudocode for* POSET–BININSERTIONSORT.

there always exists a query that partitions the space of candidate total orders into two parts, each of relative size of at least 3/11 [14]; see also [5].

In the case of width-w posets, however, it could be the case that most queries partition the space into three parts, one of which is much larger than the other two. For example, if the set consists of w incomparable chains, each of size n/w, then a random query has a response of incomparability with probability about $1 - 1/w$. (On an intuitive level, this explains the extra factor of w in the query complexity of our version of Mergesort, given in section 3.2.) Hence, we resort to more elaborate sorting strategies.

Our optimal algorithm builds upon a straightforward algorithm, called POSET–BININSERTIONSORT, which is identical to "Algorithm A" of Faigle and Turán [9]. The algorithm is inspired by the binary insertion-sort algorithm for total orders. Pseudocode for POSET–BININSERTIONSORT is presented in Figure 3.1.

The natural idea behind POSET–BININSERTIONSORT is to sequentially insert elements into a subset of the poset, while maintaining a chain decomposition of the latter into a number of chains that is at most the upper bound w on the width of the poset to be constructed. A straightforward implementation of this idea is to perform a binary search on every chain of the decomposition in order to figure out the relationship of the element being inserted with every element of that chain and, ultimately, with all the elements of the current poset. It turns out that this simple algorithm is not optimal; it is off by a factor of w from the optimum. In the rest of this section, we show how to adapt POSET–BININSERTIONSORT to achieve the lower bound given in Theorem 3.2. First, we show the following lemma.

LEMMA 3.3 (Faigle and Turán [9]). POSET–BININSERTIONSORT *sorts any partial order $\mathcal{P}$ of width at most w on n elements with $O(wn \log n)$ oracle queries.*

Proof. The correctness of POSET–BININSERTIONSORT should be clear from its description. (The simple argument showing that step 4e can be executed based on

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

the information obtained in step 4d is similar to the proof for the ChainMerge data structure in section 2.) It is not hard to see that the number of oracle queries incurred by Poset–BinInsertionSort for inserting each element is $O(w \log n)$ and, therefore, the total number of queries is $O(wn \log n)$. ∎

It follows that, as n scales, the number of queries incurred by the algorithm is larger than the lower bound by a factor of w. The Achilles' heel of the Poset-BinInsertionSort algorithm is in the method of insertion of an element—specifically, in the way the binary searches are performed (see step 4d of Figure 3.1). In these sequences of queries, no structural properties of $\mathcal{P}'$ are used for deciding which queries to the oracle are more useful than others; in some sense, the binary searches give the same "attention" to queries whose answer is guaranteed to greatly decrease the number of remaining possibilities and those whose answer could potentially not be very informative. On the other hand, as we discussed earlier, a sorting algorithm that always makes the most informative query is not guaranteed to be optimal either.

Our algorithm tries to resolve this dilemma. We suggest a scheme that has the same structure as the Poset–BinInsertionSort algorithm but exploits the structure of the already constructed poset $\mathcal{P}'$ in order to amortize the cost of the queries over the insertions. The amortized query cost matches the lower bound of Theorem 3.2.

The new algorithm, named EntropySort, modifies the binary searches of step 4d into weighted binary searches. The weights assigned to the elements satisfy the following property: the number of queries it takes to insert an element into a chain is proportional to the (logarithm of the) number of candidate posets that will be eliminated after the insertion of the element. In other words, we use fewer queries for insertions that are not informative and more queries for insertions that are informative. In some sense, this corresponds to an *entropy-weighted binary search*. To define this notion precisely, we use the following definition.

Definition 3.4. *Suppose that $\mathcal{P}' = (P', \mathcal{R}')$ is a poset of width at most w, U is a set of elements such that $U \cap P' = \emptyset$, $u \in U$, and $\mathcal{ER}, \mathcal{PR} \subseteq (\{u\} \times P') \cup (P' \times \{u\})$. We say that $\mathcal{P} = (P' \cup U, \mathcal{R})$ is* a width-w extension of $\mathcal{P}'$ on U conditioned on $(\mathcal{ER}, \mathcal{PR})$ *if $\mathcal{P}$ is a poset of width at most w, $\mathcal{R} \cap (P' \times P') = \mathcal{R}'$, and, moreover, $\mathcal{ER} \subseteq \mathcal{R}$, $\mathcal{R} \cap \mathcal{PR} = \emptyset$. In other words, $\mathcal{P}$ is an extension of $\mathcal{P}'$ on the elements of U, which is consistent with $\mathcal{P}'$, and it contains the relations of u to P' given by $\mathcal{ER}$ and does not contain the relations of u to P' given by $\mathcal{PR}$. The set $\mathcal{ER}$ is called the* set of enforced relations *and $\mathcal{PR}$ the* set of prohibited relations.

We give in Figure 3.2 the pseudocode of step 4d′ of EntropySort, which replaces step 4d of Poset–BinInsertionSort. The correctness of EntropySort follows trivially from the correctness of Poset–BinInsertionSort. We prove next that its query complexity is optimal. Recall that $N_w(n)$ denotes the number of partial orders of width at most w on n elements.

Theorem 3.5. EntropySort *sorts any partial order $\mathcal{P}$ of width at most w on n elements using at most $2 \log N_w(n) + 4wn = \Theta(n \log n + wn)$ oracle queries. In particular, the query complexity of the algorithm is at most $2n \log n + 8wn + 2w \log w$.*

Proof. We characterize the number of oracle calls required by the weighted binary searches.

Lemma 3.6 (weighted binary search). *For every $j \in \{1, 2, \ldots, \ell_i + 1\}$, if e_{ij} is the smallest element of chain C_i which dominates element e ($j = \ell_i + 1$ corresponds to the case where no element of chain C_i dominates e), then j is found after at most $2 \cdot (1 + \log \frac{\mathcal{D}_i}{\mathcal{D}_{ij}})$ oracle queries in step* v *of the algorithm* EntropySort.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Step 4d′ for Algorithm EntropySort($\mathcal{P}$)
4d′. $\mathcal{ER} = \emptyset$; $\mathcal{PR} = \emptyset$;
for $i = 1, \ldots, q$
i. let $C_i = \{e_{i1}, \ldots, e_{i\ell_i}\}$, where $e_{i\ell_i} \succ \cdots \succ e_{i2} \succ e_{i1}$;
ii. for $j = 1, \ldots, \ell_i + 1$
• set $\mathcal{ER}_j = \{(e_{ik}, e) | j \le k \le \ell_i\}$; set $\mathcal{PR}_j = \{(e_{ik}, e) | 1 \le k < j\}$;
• compute $\mathcal{D}_{ij}$, the number of width-w extensions of $\mathcal{P}'$ on U,
conditioned on $(\mathcal{ER} \cup \mathcal{ER}_j, \mathcal{PR} \cup \mathcal{PR}_j)$;
/* $\mathcal{D}_{ij}$ represents the number of posets on P consistent with $\mathcal{P}'$, ($\mathcal{ER}$, $\mathcal{PR}$),
in which e_{ij} is the smallest element of chain C_i that dominates e;
$j = \ell_i + 1$ corresponds to the case that no element of C_i dominates e;*/
endfor
iii. set $\mathcal{D}_i = \sum_{j=1}^{\ell_i+1} \mathcal{D}_{ij}$;
/* $\mathcal{D}_i$ is equal to the total number of width-w extensions of $\mathcal{P}'$ on U conditioned on $(\mathcal{ER}, \mathcal{PR})$*/
iv. partition the unit interval $[0, 1)$ into $\ell_i + 1$ intervals $([b_j, t_j))_{j=1}^{\ell_i+1}$,
where $b_1 = 0$, $b_j = t_{j-1}$ for all $j \ge 2$,
and $t_j = (\sum_{j' \le j} \mathcal{D}_{ij'})/\mathcal{D}_i$ for all $j \ge 1$.
/* each interval corresponds to an element of C_i or the "dummy" element $e_{i\ell_i+1}$ */
v. do binary search on $[0, 1)$ to find smallest element (if any) of C_i dominating e:
/* weighted version of binary search in Poset–BinInsertionSort, step 4dii */
set $x = 1/2$; $t = 1/4$; $j^* = 0$;
repeat: find j such that $x \in [b_j, t_j)$;
if ($j = \ell_i + 1$ **and** $e_{i,j-1} \not\succ e$) **or** ($e_{ij} \succ e$ **and** $j = 1$)
or ($e_{ij} \succ e$ **and** $e_{i,j-1} \not\succ e$)
set $j^* = j$; break; /* found smallest element in C_i that dominates e */
else if ($j = \ell_i + 1$) **or** ($e_{ij} \succ e$)
set $x = x - t$; $t = t * 1/2$; /* look below */
else
set $x = x + t$; $t = t * 1/2$; /* look above */
vi. e_{ij^*} is the smallest element of chain C_i that dominates e;
set $\mathcal{ER} := \mathcal{ER} \cup \mathcal{ER}_{j^*}$ and $\mathcal{PR} := \mathcal{PR} \cup \mathcal{PR}_{j^*}$;
vii. find the largest element (if any) of chain C_i that is dominated by e:
for $j = 0, 1, \ldots, \ell_i$,
compute $\mathcal{D}'_{ij}$, the number of posets on P consistent with $\mathcal{P}'$, ($\mathcal{ER}$, $\mathcal{PR}$),
in which e_{ij} is the largest element of chain C_i dominated by e;
/* $j = 0$ corresponds to case that no element of C_i is dominated by e; */
let $\mathcal{D}'_i = \sum_{j=0}^{\ell_i} \mathcal{D}'_{ij}$;
do the weighted binary search analogous to that of step v;
viii. update accordingly the sets $\mathcal{ER}$ and $\mathcal{PR}$;
endfor

Fig. 3.2. *Algorithm* EntropySort *is obtained by substituting step* 4d′ *given above for step* 4d *of the pseudocode in Figure* 3.1 *for* Poset–BinInsertionSort.

Proof of Lemma 3.6. Let $\lambda = \frac{\mathcal{D}_{ij}}{\mathcal{D}_i}$ be the length of the interval that corresponds to e_{ij}. We wish to prove that the number of queries needed to find e_{ij} (and determine that it is the smallest element of C_i that dominates e) is at most $2(1 + \lfloor \log \frac{1}{\lambda} \rfloor)$. From the definition of the weighted binary search, we see that if the interval corresponding to e_{ij} contains a point of the form $2^{-r} \cdot m$ in its interior, where r, m are integers, then the search completes after at most r iterations, each of which involves at most two oracle queries. Now, an interval of length λ must include a point of the form $2^{-r} \cdot m$, where $r = 1 + \lfloor \log \frac{1}{\lambda} \rfloor$, which concludes the proof. ∎

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

It is important to note that the number of queries used by the weighted binary search is small for uninformative insertions, which correspond to large $\mathcal{D}_{ij}$'s, and large for informative ones, which correspond to small $\mathcal{D}_{ij}$'s. Hence, this explains our use of the term entropy-weighted binary search. A parallel of Lemma 3.6 holds, of course, for finding the largest element of chain C_i dominated by element e.

Suppose now that $P = \{e_1, \ldots, e_n\}$, where $e_1, e_2, \ldots, e_n$ is the order in which the elements of P are inserted into poset $\mathcal{P}'$. Also, denote by $\mathcal{P}_k$ the restriction of poset $\mathcal{P}$ onto the set of elements $\{e_1, e_2, \ldots, e_k\}$ and by Z_k the number of width-w extensions of poset $\mathcal{P}_k$ onto $P \setminus \{e_1, \ldots, e_k\}$ conditioned on $(\emptyset, \emptyset)$. Clearly, $Z_0 \equiv N_w(n)$ and $Z_n = 1$. The following lemma is sufficient to establish the optimality of EntropySort.

Lemma 3.7. EntropySort *needs at most* $4w + 2\log\frac{Z_k}{Z_{k+1}}$ *oracle queries to insert element* e_{k+1} *into poset* $\mathcal{P}_k$ *in order to obtain* $\mathcal{P}_{k+1}$.

Proof of Lemma 3.7. Let $\mathcal{C} = \{C_1, \ldots, C_q\}$ be the chain decomposition of the poset $\mathcal{P}_k$ constructed at step 4c of EntropySort in the iteration of the algorithm in which element e_{k+1} needs to be inserted into poset $\mathcal{P}_k$. Suppose also that, for all $i \in \{1, \ldots, q\}$, $\pi_i \in \{1, \ldots, \ell_i + 1\}$ and $\kappa_i \in \{0, 1, \ldots, \ell_i\}$ are the indices computed by the binary searches of steps v and vii of the algorithm. Also, let $\mathcal{D}_i$, $\mathcal{D}_{ij}$, $j \in \{1, \ldots, \ell_i + 1\}$, and $\mathcal{D}'_i$, $\mathcal{D}'_{ij}$, $j \in \{0, \ldots, \ell_i\}$, be the quantities computed at steps ii, iii, and vii. It is not hard to see that the following are satisfied:

$$Z_k = \mathcal{D}_1; \qquad \mathcal{D}'_{q\kappa_q} = Z_{k+1};$$
$$\mathcal{D}_{i\pi_i} = \mathcal{D}'_i \quad \forall i = 1, \ldots, q; \quad \mathcal{D}'_{i\kappa_i} = \mathcal{D}_{i+1} \quad \forall i = 1, \ldots, q-1.$$

Now, using Lemma 3.6, it follows that the total number of queries required to construct $\mathcal{P}_{k+1}$ from $\mathcal{P}_k$ is at most

$$\sum_{i=1}^{q} \left(2 + 2\log\frac{\mathcal{D}_i}{\mathcal{D}_{i\pi_i}} + 2 + 2\log\frac{\mathcal{D}'_i}{\mathcal{D}'_{i\kappa_i}} \right) \le 4w + 2\log\frac{Z_k}{Z_{k+1}}. \qquad \square$$

Using Lemma 3.7, the query complexity of EntropySort is

$$\begin{aligned}
&\sum_{k=0}^{n-1} (\#\text{ queries needed to insert element } e_{k+1}) \\
&\qquad = \sum_{k=0}^{n-1} \left(4w + 2\log\frac{Z_k}{Z_{k+1}} \right) \\
&\qquad = 4wn + 2\log\frac{Z_0}{Z_n} = 4wn + 2\log N_w(n).
\end{aligned}$$

Taking the logarithm of the upper bound in Theorem 3.1, it follows that the number of queries required by the algorithm is $2n\log n + 8wn + 2w\log w$. $\square$

3.2. An efficient sorting algorithm. We turn to the problem of efficient sorting. Our Poset-Mergesort algorithm has superficially a recursive structure similar to the classical Mergesort algorithm. The merge step is quite different, however; it makes use of the technical Peeling algorithm in order to efficiently maintain a small chain decomposition of the poset throughout the recursion. The Peeling algorithm, described formally in section 3.2.2, is a specialization of the classic flow-based bipartite-matching algorithm [10] that is efficient in the comparison model.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Algorithm POSET-MERGESORT($\mathcal{P}$)
input: a set P, an oracle for a poset $\mathcal{P} = (P, \succ)$, an upper bound w on width of $\mathcal{P}$
output: a CHAINMERGE data structure for $\mathcal{P}$
run POSET-MERGESORT-RECURSE(P) to obtain decomposition $\mathcal{C}$ of $\mathcal{P}$ into w chains;
build and **return** CHAINMERGE($\mathcal{P}, \mathcal{C}$);

Procedure POSET-MERGESORT-RECURSE(P')
input: a subset $P' \subseteq P$, an oracle for $\mathcal{P} = (P, \succ)$, an upper bound w on width of $\mathcal{P}$
output: a decomposition into at most w chains of the poset $\mathcal{P}'$ induced by $\succ$ on P'
if $|P'| \leq w$
then return the trivial decomposition of $\mathcal{P}'$ into chains of length 1
else
1. partition P' into two parts of equal size, P'_1 and P'_2;
2. run POSET-MERGESORT-RECURSE(P'_1) and POSET-MERGESORT-RECURSE(P'_2);
3. collect the outputs to get a decomposition $\mathcal{C}$ of $\mathcal{P}'$ into $q \leq 2w$ chains;
4. if $q > w$, run PEELING($\mathcal{P}, \mathcal{C}$) to get a decomposition $\mathcal{C}'$ of $\mathcal{P}'$ into w chains;
return $\mathcal{C}'$;

FIG. 3.3. *Pseudocode for* POSET-MERGESORT.

3.2.1. Algorithm POSET-MERGESORT. Given a set P, a query oracle for a poset $\mathcal{P} = (P, \succ)$, and an upper bound of w on the width of $\mathcal{P}$, the POSET-MERGESORT algorithm produces a decomposition of $\mathcal{P}$ into w chains and concludes by building a CHAINMERGE data structure. To get the chain decomposition, the algorithm partitions the elements of P arbitrarily into two subsets of (as close as possible to) equal size; it then finds a chain decomposition of each subset recursively. The recursive call returns a decomposition of each subset into at most w chains, which constitutes a decomposition of the whole set P into at most $2w$ chains. Then the PEELING algorithm of section 3.2.2 is applied to reduce the decomposition to a decomposition of w chains: given a decomposition of $P' \subseteq P$, where $m = |P'|$, into at most $2w$ chains, the PEELING algorithm returns a decomposition of P' into w chains using $4wm$ queries and $O(w^2 m)$ time. The pseudocode of POSET-MERGESORT is given in Figure 3.3, and its performance is characterized by the following theorem.

THEOREM 3.8. POSET-MERGESORT *sorts any poset $\mathcal{P}$ of width at most w on n elements using at most $4wn \log(n/w)$ queries, with total complexity $O(w^2 n \log(n/w))$.*

Proof. The correctness of POSET-MERGESORT is immediate. Let $T(m)$ and $Q(m)$ be the worst-case total complexity and query complexity, respectively, of the procedure POSET-MERGESORT-RECURSE on a poset of width w containing m elements. When $m \leq w$, $T(m) = O(w)$ and $Q(m) = 0$. When $m > w$, $T(m) = 2T(m/2) + O(w^2 m)$ and $Q(m) \leq 2Q(m/2) + 2wm$. Therefore, $T(n) = O(w^2 n \log(n/w))$ and $Q(n) \leq 2wn \log(n/w)$. The cost incurred by the last step of the algorithm, i.e., that of building the CHAINMERGE, is negligible. ∎

3.2.2. The PEELING algorithm. We describe an algorithm that efficiently reduces the size of a given decomposition of a poset. It can be seen as an adaptation of the classic flow-based bipartite-matching algorithm [10] that is designed to be efficient in the oracle model and has been optimized for reducing the size of a given decomposition rather than constructing a minimum chain decomposition from scratch [6]. The PEELING algorithm is given an oracle for poset $\mathcal{P} = (P, \succ)$, where $n = |P|$, and a decomposition of P into $q \leq 2w$ chains. It first builds a CHAINMERGE data structure using at most $2qn$ queries and time $O(qn)$. Every query the algorithm makes after that is actually a look-up in the data structure and therefore takes constant time

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Algorithm PEELING($\mathcal{P}, \mathcal{C}$)
input: an oracle for poset $\mathcal{P} = (P, \succ)$, an upper bound w on the width of $\mathcal{P}$,
and a decomposition $C = \{C_1, \ldots, C_q\}$ of $\mathcal{P}$, where $q \leq 2w$
output: a decomposition of $\mathcal{P}$ into w chains
build CHAINMERGE($\mathcal{P}, \mathcal{C}$); /* All further queries are look-ups. */
for $i = 1, \ldots, q$
build linked list for chain $C_i = e_{i\ell_i} \to \cdots \to e_{i2} \to e_{i1}$, where $e_{i\ell_i} \succ \cdots \succ e_{i2} \succ e_{i1}$;
while $q > w$, perform a peeling iteration:
1. **for** $i = 1, \ldots, q$, set $C'_i = C_i$;
2. **while** every C'_i is nonempty
/* the largest element of each C'_i is a *top* element */
a. find a pair (x, y), $x \in C'_i$, $y \in C'_j$, of top elements such that $y \succ x$;
b. delete y from C'_j; /* *x dislodges y* */
3. in sequence of dislodgements, find subsequence $(x_1, y_1), \ldots, (x_t, y_t)$ such that
- y_t is the element whose deletion (in step 2b) created an empty chain;
- for $i = 2, \ldots, t$, y_{i-1} is the parent of x_i in its original chain;
- x_1 is the top element of one of the original chains;

4. modify the original chains $C_1, \ldots, C_q$:
a. **for** $i = 2, \ldots, t$
i. delete the pointer going from y_{i-1} to x_i;
ii. replace it with a pointer going from y_i to x_i;
b. add a pointer going from y_1 to x_1;
5. set $q = q - 1$, and reindex the modified original chains from 1 to $q - 1$;
return the current chain decomposition, containing w chains

FIG. 3.4. *Pseudocode for* PEELING.

and no oracle call.

The PEELING algorithm proceeds in a number of *peeling iterations*. Each iteration produces a decomposition of $\mathcal{P}$ with one less chain, until after at most w peeling iterations, a decomposition of $\mathcal{P}$ into w chains is obtained. A detailed formal description of the algorithm is given in Figure 3.4.

THEOREM 3.9. *Given an oracle for $\mathcal{P} = (P, \succ)$, where $n = |P|$, and a decomposition of $\mathcal{P}$ into at most $2w$ chains, the* PEELING *algorithm returns a decomposition of $\mathcal{P}$ into w chains. It has query complexity at most $4wn$ and total complexity $O(w^2 n)$.*

Proof. To prove the correctness of one peeling iteration, we observe first that it is always possible to find a pair (x, y) of top elements such that $y \succ x$, as specified in step 2a, since the size of any antichain is at most the width of $\mathcal{P}$, which is less than the number of chains in the decomposition. We now argue that it is possible to find a subsequence of dislodgements as specified by step 3. Let y_t be the element defined in step 3 of the algorithm. Since y_t was dislodged by x_t, x_t was the top element of some list when that happened. In order for x_t to be a top element, it was either top from the beginning, or its parent y_{t-1} must have been dislodged by some element x_{t-1}, and so on.

We claim that, given a decomposition into q chains, one peeling iteration produces a decomposition of $\mathcal{P}$ into $q-1$ chains. Recall that $y_1 \succ x_1$ and, moreover, for every i, $2 \leq i \leq t$, $y_i \succ x_i$, and $y_{i-1} \succ x_i$. Observe that, after step 4 of the peeling iteration, the total number of pointers has increased by 1. Therefore, if the link structure remains a union of disconnected chains, the number of chains must have decreased by 1, since 1 extra pointer implies 1 less chain. It can be seen that the switches performed by step 4 of the algorithm maintain the invariant that the in-degree and out-degree of every vertex is bounded by 1. Moreover, no cycles are introduced, since every pointer

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

that is added corresponds to a valid relation. Therefore, the link structure is indeed a union of disconnected chains.

The query complexity of the PEELING algorithm is exactly the query complexity of CHAINMERGE, which is at most $4wn$. We show next that one peeling iteration can be implemented in time $O(qn)$, which implies the claim.

In order to implement one peeling iteration in time $O(qn)$, a little book-keeping is needed, in particular, for step 2a. We maintain during the peeling iteration a list L of potentially comparable pairs of elements. At any time, if a pair (x, y) is in L, then x and y are top elements. At the beginning of the iteration, L consists of all pairs (x, y), where x and y are top elements. Any time an element x that was not a top element becomes a top element, we add to L the set of all pairs (x, y) such that y is currently a top element. Whenever a top element x is dislodged, we remove from L all pairs that contain x. When step 2a requires us to find a pair of comparable top elements, we take an arbitrary pair (x, y) out of L and check whether x and y are comparable. If they are not comparable, we remove (x, y) from L and try the next pair. Thus, we never compare a pair of top elements more than once. Since each element of P is responsible for inserting at most q pairs into L (when it becomes a top element), it follows that a peeling iteration can be implemented in time $O(qn)$. ∎

4. The k-selection problem. The k-selection problem is the natural problem of finding the elements in the bottom k layers, i.e., the elements of height at most $k-1$, of a poset $\mathcal{P} = (P, \succ)$, given the set P of n elements, an upper bound w on the width, and a query oracle. We present upper and lower bounds on the query complexity and total complexity of k-selection, for deterministic and randomized computational models, for the special case of $k = 1$ as well as the general case. While our upper bounds arise from natural generalizations of analogous algorithms for total orders, the lower bounds are achieved quite differently. We conjecture that our deterministic lower bound for the case of $k = 1$ is tight, though the upper bound is off by a factor of 2.

4.1. Upper bounds. We provide deterministic and randomized upper bounds for k-selection, which are asymptotically tight for $k = 1$. The basic idea for the k-selection algorithms is to iteratively use the sorting algorithms presented in section 3 to update a set of candidates that the algorithm maintains. We begin with the 1-selection problem, i.e., the problem of finding the minimal elements.

THEOREM 4.1. *The minimal elements of a poset can be found deterministically with at most wn queries and total complexity $O(wn)$.*

Proof. The algorithm updates a set containing at most w elements that are candidates for being the smallest elements. This set is initialized to $T_0 = \emptyset$. Let $x_1, \ldots, x_n$ be the elements of the poset. For $1 \le t < n$, at step t, do the following:

- Compare x_t to all elements in T_{t-1}.
- If there exists some $a \in T_{t-1}$ such that $x_t \succ a$, then do nothing; i.e., let $T_t = T_{t-1}$.
- Otherwise, set T_t to contain x_t and those elements of T_{t-1} that do not dominate x_t; i.e., let $T_t = \{x_t\} \cup T_{t-1} \setminus \{a : a \succ x_t\}$.

At the termination of the algorithm, the set T_n contains all elements of height 0. By construction of T_t, for all t, the elements in T_t are mutually incomparable. Therefore, for all t, it holds that $|T_t| \le w$, and hence the query complexity of the algorithm is at most wn. ∎

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

THEOREM 4.2. *There exists a randomized algorithm that finds the minimal elements in an expected number of queries that is upper bounded by* $\frac{w+1}{2}n+\frac{w^2-w}{2}(\log n-\log w)$.

Proof. The algorithm is similar to the algorithm for the proof of Theorem 4.1, with modifications to avoid (in expectation) worst-case behavior. Let σ be a permutation of $[n]$ chosen uniformly at random. Let $T_1 = \{x_{\sigma(1)}\}$. For $1 \le t < n$, at step t, do the following:

- Let i be an index of the candidates in T_{t-1}, i.e., $T_{t-1} = \{x_{i(1)}, \ldots, x_{i(r)}\}$, where $r \le w$.
- Let $T_t = T_{t-1}$. Let τ be a permutation of $[r]$ chosen uniformly at random.
- For $j = 1, \ldots, r$, do the following:
  - If $x_{\sigma(t)} \succ x_{i(\tau(j))}$, exit the inner "for" loop indexed by j and move to step $t+1$.
  - If $x_{i(\tau(j))} \succ x_{\sigma(t)}$, remove $x_{i(\tau(j))}$ from T_t.
- Add $x_{\sigma(t)}$ to T_t.

As in the previous algorithm, it is easy to see that at each step t the set T_t contains all the minimal elements of $A_t = \{x_{\sigma(1)}, \ldots, x_{\sigma(t)}\}$ and that $|T_t| \le w$. Note, furthermore, that at step t,

$$\mathbf{P}[x_{\sigma(t)} \text{ is minimal for } A_t] \le \frac{w}{t}.$$

If $x_{\sigma(t)}$ is not minimal for A_t, then the expected number of queries needed until $x_{\sigma(t)}$ is compared to an element $a \in A_t$ that dominates $x_{\sigma(t)}$ is clearly at most $(w+1)/2$. We thus conclude that the expected running time of the algorithm is bounded by

$$\sum_{t=2}^{w}(t-1) + \sum_{t=w+1}^{n}\left(\frac{w}{t}w + \frac{(t-w)}{t}\frac{(w+1)}{2}\right) = \binom{w}{2} + \sum_{t=w+1}^{n}\frac{1}{2t}\left(w^2 - w + tw + t\right)$$
$$\le \frac{w+1}{2}n + \frac{w^2-w}{2}(\log n - \log w). \qquad \square$$

We now turn to the k-selection problem for $k > 1$.

THEOREM 4.3. *The query complexity of the k-selection problem is at most* $16wn + 4n\log(2k) + 6n\log w$. *Moreover, there exists an efficient k-selection algorithm with query complexity at most* $8wn\log(2k)$ *and total complexity* $O(w^2 n\log(2k))$.

Proof. The basic idea is to use the sorting algorithm presented in previous sections in order to update a set of candidates for the k-selection problem. Denote the elements by $x_1, \ldots, x_n$. Let $C_0 = \emptyset$. The algorithm proceeds as follows, beginning with t=1:

- While $(t-1)wk + 1 \le n$, let $D_t = C_{t-1} \cup \{x_{(t-1)wk+1}, \ldots, x_{\min(twk,n)}\}$.
- Sort D_t. Let C_t be the solution of the k-selection problem for D_t.

Clearly, at the end of the execution, the last C_t will contain the solution to the k-selection problem. As we have shown, the query complexity of sorting D_t is $4wk\log(2wk) + 16w^2k + 2w\log w$ and, therefore, the query complexity of the algorithm is $\frac{n}{wk}(4wk\log(2wk) + 16w^2k + 2w\log w) = 4n\log(2wk) + 16wn + \frac{2n}{k}\log w$. This proves the first result. Using the computationally efficient sorting algorithm, we have sorting query complexity $8w^2k\log(2k)$, which results in total query complexity $8nw\log(2k)$ and total complexity $O(nw^2\log(2k))$. $\square$

Next we outline a randomized algorithm with a better coefficient of the main term wn.

THEOREM 4.4. *The k-selection problem has a randomized query complexity of at most* $wn + 16kw^2\log n\log(2k)$ *and total complexity* $O(wn + poly(k,w)\log n)$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Proof. We use the following algorithm:

- Choose an ordering $x_1, \ldots, x_n$ of the elements uniformly at random.
- Let $C_{wk} = \{x_1, \ldots, x_{wk}\}$ and $D_{wk} = \emptyset$.
- Sort C_{wk}. Remove any elements from C_{wk} that are of height greater than $k-1$.
- Let $t = wk + 1$. While $t \leq n$ do:
  - Let $C_t = C_{t-1}$ and $D_t = D_{t-1}$.
  - Compare x_t to the maximal elements in C_t in a random order.
    - For each maximal element $a \in C_t$: if height$(a) = k-1$ and $a \succ x_t$, or if height$(a) < k-1$ and $x_t \succ a$, then add x_t to D_t and exit this loop.
    - If for all elements $a \in C_t$, $x_t \not\sim a$, then add x_t to D_t and exit this loop.
  - If $|D_t| = wk$ or $t = n$:
    - Sort $C_t \cup D_t$.
    - Set C_t to be the elements of height at most $k-1$ in $C_t \cup D_t$.
    - Set $D_t = \emptyset$.
- Output the elements of C_n.

It is clear that C_n contains the solution to the k-selection problem. To analyze the query complexity of the algorithm, recall from Theorem 3.8 that $s(w,k) = 8w^2k\log(2k)$ is an upper bound on the number of queries used by the efficient sorting algorithm to sort $2wk$ elements in a width-w poset.

There are two types of contributions to the number of queries made by the algorithm: (1) comparing elements to the maximal elements of C_t, and (2) sorting the sets C_0 and $C_t \cup D_t$.

To bound the expected number of queries of the first type, we note that for $t \geq kw+1$, since $|\{C_t \cup D_t\}| \leq 2kw$ and the elements are in a random order, the probability that x_t ends up in D_t is at most $\min\left(1, \frac{2kw}{t}\right)$. If x_t is not going to be in D_t, then the number of queries needed to verify this is bounded by w. Overall, the expected number of queries needed for comparisons to maximal elements is bounded by wn.

To calculate the expected number of queries of the second type, we bound the expected number of elements that need to be sorted as follows:

$$\sum_{t=kw+1}^{n} \min\left(1, \frac{2kw}{t}\right) \leq 2kw(\log n - 1).$$

Thus the total query complexity is bounded above by $wn + 2s(w,k)\log n$. ∎

4.2. Lower bounds. We obtain lower bounds for the k-selection problem both for adaptive and nonadaptive adversaries. Some of our proofs use the following lower bound of Fussenegger and Gabow on finding the set containing the k smallest elements of a total order on n elements.

THEOREM 4.5 (Fussenegger and Gabow [12]). *The number of queries required to find the set of the k smallest elements of an n-element total order is at least $n - k + \log(\binom{n}{k-1}/k)$.*

The proof of Theorem 4.5 shows that every comparison tree that identifies the kth smallest element must have at least $2^{n-k}\binom{n}{k-1}$ leaves, which has the consequence that the theorem also holds for randomized algorithms.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

4.2.1. Adversarial lower bounds. We consider adversarial lower bounds for the k-selection problem. In this model, an adversary simulates the oracle and is allowed to choose her response to a query after receiving it. A response is legal if there exists a partial order of width at most w with which this response and all previous responses are consistent.

The adversarial algorithm for Theorem 4.6 outputs query responses that correspond to a poset $\mathcal{P}$ of w disjoint chains. Along with outputting a response to a query, the algorithm may also announce for a queried element to which chain it belongs. In any proof that an element a is *not* a smallest element, it must be shown to dominate at least one other element. The algorithm is designed so that in order for such a response to be given, a must first be queried against at least $w-1$ other elements with which it is incomparable.

The algorithm for Theorem 4.7 is based on a similar idea but uses a more specific rule for assigning queried elements to chains. The responses are designed to achieve a trade-off between the case when few chains are short, when Theorem 4.5 implies that the number of queries required must be large, and the case when many chains are short, when the algorithm must ensure that the number of pairs declared incomparable is large. Achieving this goal is technically challenging, as the rather involved details of the proof demonstrate.

THEOREM 4.6. *In the adversarial model, at least* $\frac{w+1}{2}n - w$ *queries are needed in order to find the minimal elements.*

Proof. Consider the following adversarial algorithm. The algorithm outputs query responses that correspond to a poset $\mathcal{P}$ of w disjoint chains. Given a query $q(a,b)$, the algorithm outputs a response to the query, and in some cases, it may also announce for one or both of a and b to which chain the element belongs. Note that receiving this extra information can make things only easier for the query algorithm. During the course of the algorithm, the adversary maintains a graph $G=(P,E)$. Whenever the adversary responds that $a \not\sim b$, it adds an edge (a,b) to E.

Let $q_t(a)$ be the number of queries that involve element a, out of the first t queries overall. Let $c(a)$ be the chain assignment that the adversary has announced for element a. (We set $c(a)$ to be undefined for all a, initially.) Let $\{x_i\}_{i=1}^n$ be an indexing, chosen by the adversary, of the elements of P. Let $q(a,b)$ be the tth query. The adversary follows the following protocol:

- If $q_t(a) \le w-1$ or $q_t(b) \le w-1$, return $a \not\sim b$. In addition, do the following:
  - If $q_t(a) = w-1$, choose a chain $c(a)$ for a that is different from all the chains to which a's neighbors in G belong and output it.
  - If $q_t(b) = w-1$, choose a chain $c(b)$ for b that is different from all the chains to which b's neighbors in G belong and output it.
- If $q_t(a) > w-1$, $q_t(b) > w-1$, and $c(a) \ne c(b)$, then output $a \not\sim b$.
- Otherwise, let i and j be the indices of a and b, respectively (i.e., $a = x_i$ and $b = x_j$). If $i > j$, then output $a \succ b$; otherwise, output $b \succ a$.

It is easy to see that the output of the algorithm is consistent with a width-w poset consisting of w chains that are pairwise incomparable. We will also require that each of the chains be chosen at least once (this is achieved easily).

We now prove a lower bound on the number of queries to this algorithm required to find a proof that the minimal elements are indeed the minimal elements.

In any proof that a is *not* a smallest element, it must be shown to dominate at least one other element, but to get such a response from the adversary, a must be queried against at least $w-1$ other elements with which it is incomparable. To

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

prove that a minimal element of one chain is indeed minimal, it must be queried at least against the minimal elements of the other chains to rule out the possibility it dominates one of them. Therefore, each element must be compared to at least $w-1$ elements that are incomparable to it. So the total number of queries of type $q(a,b)$, where $a \not\sim b$, is at least $\frac{w-1}{2}n$.

In addition, for each chain c_i of length n_i, the output must provide a proof of minimality for the minimal element of that chain. By Theorem 4.5, this contributes $n_i - 1$ queries for each chain c_i.

Summing over all the bounds proves the claim. ∎

THEOREM 4.7. *Let $r = \frac{n}{2w-1}$. If $k \le r$, then the number of queries required to solve the k-selection problem is at least*

$$\frac{(w+1)n}{2} - wk - \frac{w^3}{8} + \min\left((w-2)\log\left(\binom{r}{k-1}/k\right) + \log\binom{rw}{k-1},\right.$$
$$\left.\frac{n(w-1)(r-k)}{2r} - \log\left(\binom{r}{k-1}/k\right) + \log\binom{n-(w-1)k}{k-1}\right).$$

Proof. The adversarial algorithm outputs query responses exactly as in the proof of Theorem 4.6, except in the case where the tth query is (a,b) and $q_t(a) = w-1$ or $q_t(b) = w-1$. In that case it uses a more specific rule for the assignment of one or both of these elements to chains.

In addition to assigning the elements to chains, the process must also select the k smallest elements in each chain, and Theorem 4.5 gives a lower bound, in terms of the lengths of the chains, on the number of queries required to do so.

The specific color assignment rule is designed to ensure that if, at the end, the number of elements with color c is small, then there must have been many queries in which the element being colored could not receive color c because it had already been declared incomparable to an element with color c. ∎

It will then follow that if many of the chains are very short, then the number of pairs declared incomparable must be very large.

Assignment of colors. We think of the assignment of elements to chains as a coloring of the elements with w colors. The color assignment rule is based on a function $d_t(c)$, referred to as the *deviation* of color c after query t, and satisfies the initial condition $d_0(c) = 0$ for all c. The rule is "assign the eligible color with smallest deviation."

More specifically, let the tth query be (a_t, b_t). The adversary processes a_t and then b_t. Recall that $q_t(a)$ is the number of queries involving element a out of the first t queries overall. Element $e \in \{a_t, b_t\}$ is processed exactly as in the proof of Theorem 4.6, except when $q_t(e) = w-1$. In that case, let $S_t(e)$ be the set of colors that are *not* currently assigned to neighbors of e, i.e., the set of colors eligible to be assigned to element e. Let $c* = \operatorname{argmin}_{c \in S_t(e)} d_{t-1}(c)$. The adversary assigns color $c*$ to e. Then the deviations of all colors are updated as follows:

1. if $c \notin S_t(e)$, then $d_t(c) = d_{t-1}(c)$;
2. $d_t(c^*) \leftarrow d_{t-1}(c^*) + 1 - \frac{1}{|S_t(e)|}$;
3. for $c \in S_t(e) \setminus \{c^*\}$, $d_t(c) \leftarrow d_{t-1}(c) - \frac{1}{|S_t(e)|}$.

The deviation function $d_t(c)$ has the following interpretation: over the history of the color assignment process, certain steps occur where the adversary has the choice of whether to assign color c to some element; $d_t(c)$ represents the number of times that color c was chosen up to step t, minus the expected number of times it would

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

have been chosen if the same choices had been available at all steps and the color had been chosen uniformly at random from the set of eligible colors.

Because the smallest of the deviations of eligible colors is augmented at each step, it is not possible for any deviation to drift far from zero. It is easily shown by induction on t that at every step t the sum of the deviations is zero. In addition, we use the following lemma.

LEMMA 4.8. *For $m = 1, 2, \ldots, w$, the sum of the m smallest deviations is at least $\frac{m(m-w)}{2}$.*

In other words, the average deviation of every set of m colors is at least $(m-w)/2$.

Proof of Lemma 4.8. The proof is by induction on t. Initially, all deviations are 0. We assume the claim in Lemma 4.8 is true after each of the first $t-1$ queries. For any set C of colors, let $d_t(C) = \sum_{c \in C} d_t(c)$ be the total deviation of C after the tth update.

There are two possibilities: c' was ineligible or eligible for the tth query. If c' was ineligible, then its deviation did not change; hence $d_t(c') = d_{t-1}(c') \geq \frac{(1-w)}{2}$. If c' was eligible and chosen, then $c' = c^*$ and $d_t(c') \geq d_{t-1}(c^*) + 1 - \frac{1}{|S_t(e)|} \geq \frac{(1-w)}{2}$. If c' was eligible but not selected, then $d_{t-1}(c') \geq d_{t-1}(c^*)$, and we use the inductive hypothesis for $m = 2$, which guarantees that $d_{t-1}(c') + d_{t-1}(c^*) \geq 2 - w$, as follows.

For any $m \in [1, w]$, let C_t^m be the set of colors with the m smallest deviations after the tth query. Let $X = C_t^m \cap S_t(e)$ be the set of colors in C_t^m that were eligible for the tth query. For all $c \in C_t^m \setminus X$, $d_t(c) = d_{t-1}(c)$. Hence, if $X = \emptyset$, then the bound holds trivially. If $X \neq \emptyset$, we consider two possibilities: the color c^* that is assigned after the tth query is either in X or not.

If $c^* \in X$, then the deviation gain of c^* easily makes up for any loss among the other colors in C_t^m:

$$\begin{aligned}
d_t(C_t^m) &= d_t(X) + d_t(C_t^m \setminus X) \\
&= d_{t-1}(c^*) + 1 - \frac{1}{|S_t(e)|} + \sum_{c \in X \setminus \{c^*\}} \left(d_{t-1}(c) - \frac{1}{|S_t(e)|} \right) + d_{t-1}(C_t^m \setminus X) \\
&\geq d_{t-1}(c^*) + d_{t-1}(X \setminus \{c^*\}) + d_{t-1}(C_t^m \setminus X) \\
&= d_{t-1}(C_t^m) \\
&\geq \frac{m(m-w)}{2},
\end{aligned}$$

where the last inequality holds by the inductive hypothesis.

We now assume $c^* \notin X$. Let $k = |X| \geq 1$, so $S_t(e)$ must contain at least $k+1$ colors. By the update rules, we have the following lower bound:

$$d_t(C_t^m) \geq d_{t-1}(C_t^m) - \frac{k}{k+1}.$$

It remains to show that the set C_t^m has enough total deviation at step $t-1$ to make up for the loss. By definition of c^*, for all $c \in S_t(e)$ (and thus for all $c \in X$), $d_{t-1}(c) \geq d_{t-1}(c^*)$. Let $Q = C_t^m \cup \{c^*\}$, and let $R = C_t^m \setminus X$. Then $|Q| = m+1$, and $|R| = m-k$. We observe that, since c^* has minimum deviation among the colors of

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

$X \cup \{c^*\}$,

$$\begin{aligned} d_{t-1}(X) &\geq \frac{k}{(k+1)} d_{t-1}(X \cup \{c^*\}) \\ &= \frac{k}{(k+1)} \left(d_{t-1}(Q) - d_{t-1}(R)\right). \end{aligned}$$

We use this inequality along with two applications of the inductive hypothesis to get a lower bound on $d_{t-1}(C_t^m)$ as follows:

$$\begin{aligned} d_{t-1}(C_t^m) &= d_{t-1}(X) + d_{t-1}(R) \\ &\geq \frac{k}{(k+1)}(d_{t-1}(Q) - d_{t-1}(R)) + d_{t-1}(R) \\ &= \frac{k}{(k+1)} d_{t-1}(Q) + \frac{1}{(k+1)} d_{t-1}(R) \\ &\geq \frac{k}{(k+1)} \frac{(m+1)(m+1-w)}{2} + \frac{1}{(k+1)} \frac{(m-k)(m-k-w)}{2} \\ &= \frac{m(m-w)}{2} + \frac{k}{2}. \end{aligned}$$

Putting the bounds together yields the following result:

$$d_t(C_t^m) \geq \frac{m(m-w)}{2} + \frac{k}{2} - \frac{k}{k+1} \geq \frac{m(m-w)}{2}. \qquad \square$$

Let $\deg_G(a)$ be the degree of a in G at the end of the color assignment process. The total number of pairs of elements that have been declared incomparable is $\frac{1}{2}\sum_a \deg_G(a)$. At the end of the process, every element of degree in G at least $w-1$ has been assigned to a chain. Each element of degree less than $w-1$ has not been assigned to a chain and is therefore called *unassigned*. An unassigned element is called *eligible* for chain c if it has not been compared (and found incomparable) with any element of chain c.

Let $s(c)$ be the length of chain c. We define $\mathrm{def}(c)$, the *deficiency* of chain c, as $\mathrm{def}(c) = \max(0, k - s(c))$ and the *total deficiency* DEF as the sum of the deficiencies of all chains: $\mathrm{DEF} = \sum_c \mathrm{def}(c)$.

The following two lemmas lower bound the degrees in G of unassigned elements and chains.

LEMMA 4.9. *Let U be the set of unassigned elements at the termination of the color assignment process. Let $u = |U|$ be the number of unassigned elements. Then*

$$\sum_{a \in U} \deg_G(a) \geq (w-1)u - \mathrm{DEF}.$$

Proof of Lemma 4.9. Upon the termination of the process it must be possible to infer from the results of the queries that every unassigned element is of height at most $k-1$. This implies that the number of unassigned elements eligible for chain c must be at most $\mathrm{def}(c)$. Thus the number of pairs (a, c) such that unassigned element a is eligible for chain c is at most DEF.

Let the *deficiency* of unassigned element a be defined as $\mathrm{def}(a) = w - 1 - \deg_G(a)$. Then $\sum_{a \in U} \mathrm{def}(a) \leq \mathrm{DEF}$, and therefore $\sum_{a \in U} \deg_G(a) \geq (w-1)u - \mathrm{DEF}$. $\square$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

LEMMA 4.10. *Let $d(c)$ be the deviation of color c at the end of the color assignment process. Then*

$$\sum_{a|c(a)=c} \deg_G(a) \geq \max((w-1)s(c), n - w(s(c) - d(c))).$$

Proof of Lemma 4.10. Let $r(c)$ be the number of steps in the course of the process at which the element being colored was eligible to receive color c. If, at each such step, the color had been chosen uniformly from the set of eligible colors, then the chance of choosing color c would have been at least $\frac{1}{w}$. Thus, by our interpretation of the function $d_t(c)$ given above, $s(c) \geq \frac{r(c)}{w} + d(c)$; equivalently, $r(c) \leq w(s(c) - d(c))$.

As color c is ineligible during the color assignment of element a only if there exists an edge between a and an element colored c, the following inequality holds:

$$\begin{aligned}\sum_{a|c(a)=c} \deg_G(a) &\geq n - r(c)\\ &\geq n - w(s(c) - d(c)).\end{aligned}$$

This sum is also at least $(w-1)s(c)$, since every element assigned to c has been declared incomparable with at least $(w-1)$ other elements. ∎

To formulate a lower bound, we give a name to the quantity in the Fussenegger–Gabow lower bound (Theorem 4.5); i.e., we define

$$g(s) = s - k + \log\left(\binom{s}{k-1}/k\right).$$

Then, for each chain c, we define a *cost* function as follows:

$$\text{cost}(c) = \frac{1}{2}\sum_{a|c(a)=c} \deg_G(a) + \max\left(0, g(s(c))\right).$$

We can now give a lower bound in terms of the cost function.

LEMMA 4.11. *The number of queries required to solve k-selection is at least*

$$\sum_c \text{cost}(c) + \frac{1}{2}\left((w-1)u - \text{DEF}\right), \tag{1}$$

and

$$\sum_c \text{cost}(c) \geq \frac{1}{2}\sum_c \max\left((w-1)s(c), n - w(s(c) - d(c))\right) + \sum_{c|s(c)>k} g(s(c)).$$

Proof of Lemma 4.11. The total number of queries is the number of edges that have been placed in G in the course of the algorithm (i.e., the number of pairs that have been declared incomparable by the adversary), plus the number of queries required to perform k-selection in each chain. Using Lemma 4.9, we find that the number of edges in G is at least $\frac{1}{2}((w-1)u - \text{DEF} + \sum_c \sum_{a|c(a)=c} \deg_G(a))$. By Theorem 4.5, if $s(c) > k$, then at least $g(s(c))$ queries are needed to determine the k smallest elements of chain c. Hence, quantity (1) is a lower bound on the total number of queries. The lower bound on the sum $\sum_c \text{cost}(c)$ is implied by Lemma 4.10. ∎

To continue with the calculation of our lower bound, we now minimize quantity (1) over all choices of nonnegative integers $s(c)$, u, and DEF, such that $\sum_c s(c) + u = n$ and $\text{DEF} = \sum_c \max(0, k - s(c))$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Noting that $\sum_c \min(d(c), 0) \geq \min_m m(m-w)/2 = -w^2/8$, we obtain the following lower bound on the total number of queries:

$$\sum_c \mathrm{cost}(c) + \frac{1}{2}\left((w-1)u - \textsc{def}\right) \tag{2}$$

$$= \frac{1}{2}\left((w-1)u - \textsc{def} + \sum_c \max\left((w-1)s(c), n - w(s(c) - d(c))\right)\right) + \sum_{c|s(c)>k} g(s(c))$$

$$\geq \frac{(w-1)n}{2} - \frac{\textsc{def}}{2} - \frac{w^3}{8} + \frac{1}{2}\sum_c \max\left(0, n - (2w-1)s(c)\right) + \sum_{c|s(c)>k} g(s(c)).$$

Let $r = \frac{n}{2w-1}$. We now restrict our attention to the case $k \leq r$. First, we show that, at any global minimum of quantity (2), $\textsc{def} = 0$. To see this, consider any choice of $\{s(c)\}$ such that $\textsc{def} > 0$, and let c be a chain such that $\mathrm{def}(c) > 0$. If $s(c)$ is increased by 1, then $\textsc{def}$ decreases by 1 and the net change in the value of quantity (2) is $\frac{1}{2} + \frac{1}{2}(-(2w-1)) = 1 - w$, which is negative.

Thus, in minimizing quantity (2) we may assume that $\textsc{def} = 0$ and hence that $\sum_c s(c) = n$. So, we may rewrite the inequality in line (2) as

$$\sum_c \mathrm{cost}(c) + \frac{1}{2}\left((w-1)u - \textsc{def}\right) \geq \frac{(w-1)n}{2} - \frac{w^3}{8} + \sum_c F(s(c)), \tag{3}$$

where

$$F(s) = \begin{cases} \frac{1}{2}\max(0, n - (2w-1)s) & \text{if } s \leq k, \\ \frac{1}{2}\max(0, n - (2w-1)s) + g(s) & \text{if } s > k. \end{cases}$$

The following lemma, combined with inequality (3) and Lemma 4.11, yields the claim of Theorem 4.7.

LEMMA 4.12. *Subject to the constraints that, for all c, $s(c) \geq 0$, and $\sum_c s(c) = n$, the sum $\sum_c F(s(c))$ is at least*

$$n - wk + \min\left((w-2)\log\left(\binom{r}{k-1}/k\right) + \log\binom{rw}{k-1}, \right.$$
$$\left. \frac{n(w-1)(r-k)}{2r} - \log\left(\binom{r}{k-1}/k\right) + \log\binom{n-(w-1)k}{k-1}\right).$$

Proof of Lemma 4.12. First, we observe that $\sum_{c|k<s(c)}(s(c) - k) = n - wk$. To determine the minimum of $\sum_c F(s(c))$, we consider three ranges of the values of $s(c)$: the low range where $s(c) = k$, the medium range where $k < s(c) \leq r$, and the high range where $r < s(c) \leq n$.

As $F(s)$ is either piecewise linear or a sum of piecewise linear functions and the log function, it is straightforward to check that $F(s)$ is strictly concave in the medium range, and concave and strictly increasing in the high range. Therefore, given a pair of values $s(c_1), s(c_2)$ in the high range, the value of the objective function $\sum_c F(s(c))$ is not increased by decreasing $s(c_1)$ by 1 and increasing $s(c_2)$ by 1. All but one of the values in the high range may thus be pushed to r, and hence we may assume that there is exactly one chain whose length is in the high range. By similar reasoning, it follows that all but one of the values in the medium range may be pushed to either

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

k or r, while there may remain one value falling strictly within the medium range. Hence, we may assume that for every $s(c)$ that falls in the medium range, except for one possible exception, $s(c) = r$.

Since $\sum_{c|k\le s(c)\le r} s(c) \le r(w-1)$, the unique value of $s(c)$ in the high range is at least rw. Let $D = r(w-1) - \sum_{c|k\le s(c)\le r} s(c)$. Then the unique value in the high range is $rw+D$. Since every value, except possibly one, in the low and medium ranges is equal to k or r, the number of chains of length r is at least $w-2-\frac{D}{r-k}$. Hence, $\sum_{c|k<s(c)\le r} \log(\binom{s(c)}{k-1}/k) \ge (w-2-\frac{D}{r-k})\log(\binom{r}{k-1}/k)$. Finally, a simple calculation shows that $\frac{1}{2}\sum_c \max(0, n-(2w-1)s(c)) = \frac{nD}{2r}$.

Thus $\sum_c F(s(c))$ is at least

$$n-wk+\min_{0\le D\le (w-1)(r-k)}\left(\frac{nD}{2r}+\left(w-2-\frac{D}{r-k}\right)\log\left(\binom{r}{k-1}/k\right)+\log\binom{rw+D}{k-1}\right).$$

Since this is a concave function in D, it is minimized either at $D=0$ or $D=(w-1)(r-k)$. Substituting and simplifying gives the lower bound claimed. ∎

4.2.2. Lower bounds in the randomized query model. We also give lower bounds on the number of queries used by randomized k-selection algorithms. We conjecture that the randomized algorithm for finding the minimal elements given in the proof of Theorem 4.2 essentially achieves the lower bound, though the lower bound we prove here is a factor 2 different from that upper bound.

THEOREM 4.13. *The expected query complexity of any algorithm solving the k-selection problem is at least*

$$\frac{w+3}{4}n - wk + w\left(1-\exp\left(-\frac{n}{8w}\right)\right)\left(\log\left(\binom{n/(2w)}{k-1}/k\right)\right).$$

Proof. We consider a distribution $D(n,w)$ on partial orders of width w over a set $P=\{x_1,\dots,x_n\}$. The distribution $D(n,w)$ is defined as follows:

- The support of $D(n,w)$ is the set of partial orders consisting of w chains, where any two elements from different chains are incomparable.
- Each element belongs independently to one of the w chains with equal probability.
- The linear order on each chain is chosen uniformly.

In order to provide a lower bound on the number of queries, we provide a lower bound on the number of queries of incomparable elements and then use the classical bound to bound the number of queries of comparable elements.

First, we note that for each element a the algorithm must make either at least one query where a is comparable to some other element b or at least $w-1$ queries where a is incomparable to all elements queried. (The latter may suffice in cases where a is the unique element of a chain and it is compared to all minimal elements of all other chains.)

We let $Y_t(i)$ denote the number of queries involving x_i *before* the first query for which the response is that x_i is comparable to an element. Also for each of the chains $C_1,\dots,C_w$ we denote by Z_α the number of queries involving two elements from the same chain.

Letting T denote the total number of queries before the algorithm terminates, we obtain

$$\mathbf{E}[T] \ge \sum_{i=1}^{n}\frac{1}{2}\mathbf{E}[Y_T(i)] + \sum_{\alpha=1}^{w}\mathbf{E}[Z_\alpha].$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

We claim that for all $1 \leq i \leq n$ we have $\mathbf{E}[Y_T(i)] \geq \frac{w-1}{2}$. This follows by conditioning on the chains that all other elements but x_i belong to. With probability $1/w$, the first query will give a comparison; with probability $1/w$, the second query, etc.

On the other hand, by the classical lower bound in Theorem 4.5, we have for each $1 \leq \alpha \leq w$ that

$$Z_\alpha \geq |C_\alpha| - k + \log\left(\binom{|C_\alpha|}{k-1}/k\right).$$

Taking the expected value, we obtain

$$\mathbf{E}[Z_\alpha] \geq \frac{n}{w} - k + \mathbf{E}\left[\log\left(\binom{|C_\alpha|}{k-1}/k\right)\right].$$

A rough bound on the previous expression may be obtained by using the fact that by standard Chernoff bounds, except with probability $\exp(-\frac{n}{8w})$, it holds that C_α is of size at least $n/(2w)$. Therefore

$$\mathbf{E}\left[\log\left(\binom{|C_\alpha|}{k-1}/k\right)\right] \geq \left(1 - \exp\left(-\frac{n}{8w}\right)\right)\log\left(\binom{n/(2w)}{k-1}/k\right).$$

Summing all of the expressions above, we obtain

$$\frac{(w-1)n}{4} + w\left(\frac{n}{w} - k\right) + \left(1 - \exp\left(-\frac{n}{8w}\right)\right) w \log\left(\binom{n/(2w)}{k-1}/k\right),$$

and simplifying gives the desired result. ∎

5. Computing linear extensions and heights. We provide upper bounds for two problems that are closely related to the problem of determining a partial order: given a poset, compute a linear extension, and compute the heights of all elements. A total order $(P, >)$ is a *linear extension* of a partial order $(P, \succ)$ if, for any two elements $x, y \in P$, $x \succ y$ implies $x > y$.

Our algorithms are analogous to Quicksort, and are based on a *ternary* search tree, an extension of the well-known binary search tree for maintaining elements of a linear order.

THEOREM 5.1. *There is a randomized algorithm that, given a poset of size n and width at most w, computes a linear extension of the poset and has expected total complexity $O(n \log n + wn)$.*

Proof. A ternary search tree for a poset $\mathcal{P} = (P, \succ)$ consists of a root, a left subtree, a middle subtree, and a right subtree. The root contains an element $x \in P$, and the left, middle, and right subtrees are ternary search trees for the restrictions of $\mathcal{P}$ to the sets $\{y \mid x \succ y\}$, $\{y \mid x \not\succ y\}$, and $\{y \mid y \succ x\}$, respectively. The ternary search tree for the empty poset consists of a single empty node.

We give a simple randomized algorithm to construct a ternary search tree for $\mathcal{P}$ as follows: The algorithm assigns a random element of P to the root, compares each of the $n-1$ other elements to the element at the root to determine the sets associated with the three children of the root, and then, recursively, constructs a ternary search tree for each of these three sets.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Define the weight of an internal node x of a ternary search tree as the total number of internal nodes in its three subtrees and the weight of a ternary search tree as the sum of the weights of all internal nodes. Then the number of queries required to construct a ternary search tree is exactly the weight of the tree.

LEMMA 5.2. *The expected weight of a ternary search tree for any poset of size n and width w is $O(n \log n + wn)$.*

Proof of Lemma 5.2. Consider the path from the root to a given element x. The number of edges in this path from a parent to a middle subtree is at most w. The expected number of edges from a parent to a left or right subtree is $O(\log n)$, since, at every step along the path, the probability is at least 1/2 that the sizes of the left and right subtrees differ by at most a factor of 3. It follows that the expected contribution of any element to the weight of the ternary search tree is $w + O(\log n)$. ∎

Once a ternary search tree for a poset has been constructed, a linear extension can be constructed by a single depth-first traversal of the tree. If x is the element at the root, then the linear extension is the concatenation of the linear extensions of the following four subsets, corresponding to the node and its three subtrees: $\{y \mid x \succ y\}$, $\{x\}$, $\{y \mid x \not\sim y\}$, and $\{y \mid y \succ x\}$. The claim follows. ∎

THEOREM 5.3. *There is a randomized algorithm that, given a poset of size n and width at most w, determines the heights of all elements and has expected total complexity $O(wn \log n)$.*

Proof. We establish first the following lemma.

LEMMA 5.4. *There is a deterministic algorithm that, given a linear extension of a poset of size n and width at most w, computes the heights of all elements and has total complexity $O(wn \log n)$.*

Proof of Lemma 5.4. Let $h(x) = h$ be the height of element x in $(P, \succ)$. Given a linear extension $x_n > \cdots > x_2 > x_1$, it is easy to compute $h(x)$ for each element x by binary search using the following observation: Let

$$S(i, h) = \{x_j \mid j \leq i, h(x_j) = h\}$$

be the set of elements of index at most i in the linear extension and of height h in $(P, \succ)$. Then $|S(i, h)| \leq w$ (as the elements of $S(i, h)$ are pairwise incomparable), and $h(x_{i+1}) > h$ if and only if there exists $x \in S(i, h)$ such that $x_{i+1} \succ x$. Thus, given the sets $S(i, h)$, for all h, we can determine $h(x_{i+1})$ and the sets $S(i+1, h)$, for all h, in time $O(w \log i)$ using binary search. Summing over i yields the claim. ∎

Combining the algorithms of Theorem 5.1 and Lemma 5.4 yields the claim. ∎

6. Variants of the poset model. We discuss sorting in two variants of the poset model that occur when different restrictions are relaxed. First, we consider posets for which a bound on the width is not known in advance. Second, we allow the irreflexivity condition to be relaxed, which leads to transitive relations. We show that with relatively little overhead in complexity, sorting in either case reduces to the problem of sorting posets.

6.1. Unknown width. Recall from section 3 that $N_w(n)$ is the number of posets of width at most w on n elements.

CLAIM 6.1. *Given a set P of n elements and access to an oracle for poset $\mathcal{P} = (P, \succ)$ of unknown width w, there is an algorithm that sorts $\mathcal{P}$ using at most $\log w\,(2 \log N_{2w}(n) + 8wn) = \Theta(n \log w\,(\log n + w))$ queries, and there is an efficient algorithm that sorts P using at most $8nw \log w \log(n/(2w))$ queries with total complexity $O(nw^2 \log w \log(n/w))$.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Proof. We use an alternate version of EntropySort that returns fail if it cannot insert an element (while maintaining a decomposition of the given width) and an alternate version of Poset-Mergesort that returns fail if the Peeling algorithm cannot reduce the size of the decomposition to the given width. The first algorithm of the claim is, for $i = 1, 2, \ldots$, to run the alternate version of algorithm EntropySort on input set P, the oracle, and width upper bound 2^i until the algorithm returns without failing. The second algorithm is analogous but uses the alternate version of Poset-Mergesort. The claim follows from Theorems 3.5 and 3.8 and from the fact that we reach an upper bound of at most $2w$ on the width of $\mathcal{P}$ in $\log w$ rounds. ∎

6.2. Transitive relations. A partial order is a particular kind of transitive relation. Our results generalize to the case of arbitrary transitive relations (which are not necessarily irreflexive or antisymmetric) and are therefore relevant to a broader set of applications. Formally, a transitive relation is a pair $(P, \trianglerighteq)$, where P is a set of elements and $\trianglerighteq \subseteq P \times P$ is transitive. The *width* of a transitive relation is defined to be the maximum size of a set of mutually incomparable elements.

Claim 6.2. *Suppose there is an algorithm $\mathcal{A}$ that, given a set P of n elements, access to an oracle $\mathcal{O}_{\succ}$ for a poset $\mathcal{P} = (P, \succ)$, and an upper bound of w on the width of $\mathcal{P}$, sorts $\mathcal{P}$ using $f(n, w)$ queries and $g(n, w)$ total complexity. Then there is an algorithm $\mathcal{B}$ that, given P, w, and access to an oracle $\mathcal{O}_{\trianglerighteq}$ for a transitive relation $(P, \trianglerighteq)$ of width at most w, sorts $(P, \trianglerighteq)$ using $f(n, w) + 2nw$ queries and $g(n, w) + O(nw)$ total complexity.*

Proof. We say that a poset $(P, \succ)$ is *induced* by a transitive relation $(P, \trianglerighteq)$ if $\succ \subseteq \trianglerighteq$. A poset $(P, \succ)$ is *minimally induced* by $(P, \trianglerighteq)$ if for any relation $(x, y) \in \trianglerighteq \setminus \succ$ the pair $(P, \succ \cup (x, y))$ is not a valid partial order; i.e., its corresponding graph contains a directed cycle.

We require the following lemma, bounding the width of a minimally induced poset.

Lemma 6.1. *Let $(P, \succ)$ be a poset minimally induced by the transitive relation $(P, \trianglerighteq)$. Then the width of $(P, \succ)$ is equal to the width of $(P, \trianglerighteq)$.*

Proof of Lemma 6.1. Suppose otherwise; that is, suppose that there is a pair of distinct elements $x, y \in P$ such that $x \not\succ y$ with respect to the partial order $(P, \succ)$, but x and y have some relation in $(P, \trianglerighteq)$. Without loss of generality, suppose that $x \trianglerighteq y$; it may be simultaneously true that $y \trianglerighteq x$. First, we note that $(P, \succ \cup (x, y))$ is a valid partial order; if it were not, i.e., if the addition of (x, y) introduced a cycle, then it would be the case that $y \succ x$, which is a contradiction to their incomparability. However, the poset $(P, \succ \cup (x, y))$ is also induced by $(P, \trianglerighteq)$, which contradicts the assumption that $(P, \succ)$ is minimally induced. ∎

We assume that the poset sorting algorithm outputs a chain decomposition (such as a ChainMerge); if it does not, the total complexity of the algorithm for sorting a transitive relation increases a bit, but its query complexity does not.

Given an oracle $\mathcal{O}_{\trianglerighteq}$ for the transitive relation $(P, \trianglerighteq)$, we define a special poset oracle $\mathcal{O}$ that runs as follows: Given a query $q(x, y)$, the oracle $\mathcal{O}$ first checks whether the relation between x and y can be inferred by transitivity and irreflexivity from previous responses. If so, it outputs the appropriate inferred response; otherwise, it forwards the query to the oracle $\mathcal{O}_{\trianglerighteq}$. The oracle $\mathcal{O}$ outputs the response of $\mathcal{O}_{\trianglerighteq}$ except if both $x \trianglerighteq y$ and $y \trianglerighteq x$; in this case, $\mathcal{O}$ outputs whichever relation is consistent with the partial order determined by previous responses (if both relations are consistent, then it arbitrarily outputs one of the two). By definition, the responses of $\mathcal{O}$ are consistent with a partial order induced by $(P, \trianglerighteq)$.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

The first step of algorithm $\mathcal{B}$ is to run algorithm $\mathcal{A}$ on input P and w, giving $\mathcal{A}$ access to the special oracle $\mathcal{O}$, which $\mathcal{B}$ simulates using its access to $\mathcal{O}_{\trianglerighteq}$. Since $\mathcal{A}$ completely sorts its input, it reconstructs a poset induced by $(P, \trianglerighteq)$ via $\mathcal{O}$ that has a maximal set of relations. That is, there is a poset $\mathcal{P} = (P, \succ)$ minimally induced by $(P, \trianglerighteq)$ such that the responses of $\mathcal{O}$ to the sequence of queries made by $\mathcal{A}$ are indistinguishable from the responses of $\mathcal{O}_{\succ}$ to the same sequence of queries. Since $\mathcal{P}$ has the same width as $(P, \trianglerighteq)$, it is valid to give $\mathcal{A}$ the upper bound of w. Hence, $\mathcal{A}$ sorts $\mathcal{P}$ and outputs some chain decomposition $\mathcal{C} = \{C_1, \ldots, C_q\}$ of $\mathcal{P}$ such that $q \leq w$.

The second step of algorithm $\mathcal{B}$ is to make a sequence of queries to the oracle $\mathcal{O}_{\trianglerighteq}$ to recover the relations in $\trianglerighteq \setminus \succ$. It is similar to building a ChainMerge data structure: for all i, j, $1 \leq i, j \leq q$, for every element $x \in C_i$, we store the index of x in chain C_i and the index of the largest element $y \in C_j$ such that $x \trianglerighteq y$. An analysis similar to the one for ChainMerge (see section 2) shows that it takes at most $2nq$ queries to the oracle $\mathcal{O}_{\trianglerighteq}$ and total complexity $O(nq)$ to find all the indices. The relation in $(P, \trianglerighteq)$ between any pair of elements can then be looked up in constant time. $\square$

7. Future directions. One of the most interesting issues brought forward but left unresolved by this work is the precise total complexity of sorting, as it is not at all clear whether optimal query complexity may be achieved efficiently. One avenue we have contemplated is the use of approximate-counting techniques, similar to those described in Dyer, Frieze, and Kannan [8], to create an efficient version of the EntropySort algorithm. Related open problems include finding the precise query complexity and total complexity of k-selection and of sorting when a bound on the width is not known in advance.

We are also intrigued by the possibility of other problems related to partial orders that parallel classical theory, including the design of efficient static and dynamic data structures analogous to heaps and binary search trees. Exploration through the application of the algorithms in this paper to practical settings may be a good means of spurring ideas for future work.

Acknowledgments. We thank Mike Saks for the reference to the related work by Faigle and Turán [9]. We also thank the referees for helpful comments.

REFERENCES

[1] P. Boldi, F. Chierichetti, and S. Vigna, *Pictures from Mongolia—partial sorting in a partial world*, in Proceedings of FUN, 2007, pp. 66–77.

[2] G. Brightwell, *Balanced pairs in partial orders*, Discrete Math., 201 (1999), pp. 25–52.

[3] G. Brightwell and S. Goodall, *The number of partial orders of fixed width*, Order, 20 (2003), pp. 333–345.

[4] G. Brightwell and P. Winkler, *Counting linear extensions is #P-Complete*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 175–181.

[5] G. R. Brightwell, S. Felsner, and W. T. Trotter, *Balancing pairs and the cross product conjecture*, Order, 12 (1995), pp. 327–349.

[6] Y. Chen, *Decomposing DAGs into Disjoint Chains*, Lecture Notes in Comput. Sci. 4653, Springer, Berlin, 2007.

[7] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, 1991.

[8] M. E. Dyer, A. M. Frieze, and R. Kannan, *A random polynomial time algorithm for approximating the volume of convex bodies*, J. Assoc. Comput. Mach., 38 (1991), pp. 1–17.

[9] U. Faigle and Gy. Turán, *Sorting and recognition problems for ordered sets*, SIAM J. Comput., 17 (1988), pp. 100–113.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[10] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
[11] M. Fredman, *How good is the information theory bound in sorting?*, Theoret. Comput. Sci., 1 (1976), pp. 355–361.
[12] F. Fussenegger and H. N. Gabow, *A counting approach to lower bounds for selection problems*, J. Assoc. Comput. Mach., 26 (1979), pp. 227–238.
[13] J. Kahn and J. H. Kim, *Entropy and sorting*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 178–187.
[14] J. Kahn and M. Saks, *Balancing poset extensions*, Order, 1 (1984), pp. 113–126.
[15] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison–Wesley, Reading, MA, 1998.
[16] N. Linial, *The information-theoretic bound is good for merging*, SIAM J. Comput., 13 (1984), pp. 795–801.
[17] M. E. J. Newman, SIAM Rev., 45 (2003), pp. 167–256.
[18] K. Onak and P. Parys, *Generalization of binary search: Searching in trees and forest-like partial orders*, in Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, 2006, pp. 379–388.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.