Sorting and Selection in Posets

Constantinos Daskalakis* Richard M. Karp† Elchanan Mossel‡ Samantha Riesenfeld§
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 optimal query complexity $O(n(w + \log n))$. 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 $O(wn)$ query and total complexity. 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.

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 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

*Microsoft Research. Research done while the author was a student in Computer Science at UC Berkeley. Supported by a Microsoft Research Fellowship, NSF grant CCF-0635319, a Yahoo! gift, and a MICRO grant. email:costis@cs.berkeley.edu.

†Computer Science, UC Berkeley. Supported by NSF grant CCF-0515259. email:karp@icsi.berkeley.edu.

‡Statistics and Computer Science, UC Berkeley, and Mathematics and Computer Science, Weizmann Institute of Science. Supported by DOD grant N0014-07-1-05-06, NSF grants DMS 0528488 and DMS 0548249, and a Sloan Fellowship in Mathematics. email:mossel@stat.berkeley.edu.

§Computer Science, UC Berkeley. Supported by NSF grant CCF-0515259. email:samr@eecs.Berkeley.edu.

¶Institute for Theoretical Computer Science, Tsinghua University. Part of this research was done while the author was a student at Tel Aviv University. Supported by the National Natural Science Foundation of China Grant 60553001, and the National Basic Research Program of China Grants 2007CB807900 and 2007CB807901. email:eladv@tsinghua.edu.cn.

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

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., [18]), 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 (e.g. [2, 11, 13, 16, 17]).

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, 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 we 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 *anti-chain* $A \subseteq P$ 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 anti-chain 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 [9]. Furthermore, it is not clear if there exists a polynomial-time implementation of the second algorithm.

Boldi et al. [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

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

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 [19] 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 total and query complexity.

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 et al. [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, given in the full version of this paper [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 optimal query complexity $O(n(logn + w))$. 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 $\mathcal{P} = (P, \succ)$. 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 (see proof in the full version [7]).

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.*

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. *The number $N_w(n)$ of partially ordered sets of n elements and width at most w satisfies $\frac{n!}{w!}\ 4^{n(w-1)}\ n^{-24w(w-1)} \le N_w(n) \le n!\ 4^{n(w-1)}\ n^{-(w-2)(w-1)/2} w^{w(w-1)/2}$.*

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.*

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

Proof. From Theorem 3.1, if $w = o\left(\frac{n}{\log n}\right)$, 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 $w = \Omega\left(\frac{n}{\log n}\right)$, $wn = \Omega(n \log n + wn)$.

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, 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].

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 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 equal to the width w 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. We show (see straightforward proof in the full version of this paper [7]):

Lemma 3.1. (Faigle & 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.*

It follows that, as n scales, the number of queries incurred by the algorithm is by a factor of w larger than the lower bound. 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 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 be not 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 spend fewer queries for insertions that are not informative and more queries for insertions that

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 in $\mathcal{P}'$*/
  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 $e_{i\ell_i} \succ \ldots \succ e_{i2} \succ e_{i1}$;
    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($\mathcal{P}', \mathcal{C}$);

Figure 1: pseudo-code for POSET–BININSERTIONSORT

are informative. In some sense, this corresponds to an *entropy-weighted binary search*. To define this notion precisely, we use the following definition.

DEFINITION 3.1. *Suppose that $\mathcal{P}' = (P', \mathcal{R}')$ is a poset of width at most w, U 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 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 $\mathcal{P}'$, 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 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 3.3. 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 3.2. (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 described above.*

Proof of Lemma 3.2: 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} 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 reaches e_{ij} after at most r steps. 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. ■

It is important to note that the number of queries spent 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, our use of the term entropy-weighted binary search. A parallel of Lemma 3.2 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 $\{e_1, e_2, ..., e_k\}$ and by Z_k the number of width w extensions of poset

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 $e_{i\ell_i} \succ \ldots \succ e_{i2} \succ e_{i1}$;

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 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 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** $j = 0)$
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

Figure 2: Algorithm EntropySort is obtained by substituting Step 4d′ given above for Step 4d of the pseudo-code in Figure 1 for Poset–BinInsertionSort.

$\mathcal{P}_k$ on $P \setminus \{e_1, \ldots, e_k\}$ 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 3.3. 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 3.3: 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

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

Steps ii., iii. and vii. It is not hard to see that the following are satisfied

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

Now, using Lemma 3.2, it follows that the total number of queries required to construct $\mathcal{P}_{k+1}$ from $\mathcal{P}_k$ is at most

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

■

Using Lemma 3.3, the query complexity of Entropy-Sort 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 $2n\log n + 8wn + 2w\log w$.

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.

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 pseudo-code of Poset-Mergesort is given in the full version [7], and its performance is characterized by the following.

Theorem 3.4. 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 $O(w^2 n\log(n/w))$.*

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 [5]. 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 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.

Theorem 3.5. *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 of Theorem 3.5: 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 1a, 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 2a. 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

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 re-index the modified original chains from 1 to $q - 1$;
return the current chain decomposition, containing w chains

Figure 3: pseudo-code for the PEELING Algorithm

$q - 1$ 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 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 if 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 to 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 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

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

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. Due to lack of space, all proofs are given in the full version [7].

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

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

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 $O(w^2 n\log(2k))$.*

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 $O(wn + poly(k,w)\log n)$.*

4.2 Lower bounds We obtain lower bounds for the k-selection problem both for adaptive and non-adaptive adversaries. Some of our proofs use the following lower bound on finding the k-th smallest element of a total order on n elements:

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

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 implies that the theorem also holds for randomized algorithms.

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 w with which this response and all previous responses are consistent.

The adversarial algorithm for Theorem 4.6 below 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 so that if many chains are very short, then the number of pairs declared incomparable must be large. Achieving this goal is technically challenging; the rather involved details of this argument are given in the full version [7]. If, on the other hand, few of the chains are very short, then the Fussenegger-Gabow Theorem implies that the number of queries required to select the k smallest elements in each chain must be large.

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

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*

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

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 [7] achieves the lower bound.

THEOREM 4.8. *The expected query complexity of any algorithm solving the k-selection problem is at least $\frac{w+3}{4}n - wk + w\left(1 - \exp\left(-\frac{n}{8w}\right)\right)\left(\log\binom{n/(2w)}{k-1}\right)$.*

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

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

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. We give the proof in the full version [7].

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 $O(n \log n + wn)$.*

THEOREM 5.2. *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 $O(wn \log n)$.*

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. We provide the proof of the following in [7].

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 $O(nw^2 \log w \log(n/w))$.*

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) and are therefore relevant to a broader set of applications. See [7] for details.

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.*

Acknowledgments We thank Mike Saks for the reference to [9].

References

[1] P. Boldi and F. Chierichetti and S. Vigna. "Pictures from Mongolia — Partial Sorting in a Partial World," *FUN* 2007.

[2] G. Brightwell. "Balanced Pairs in Partial Orders," *Discrete Mathematics* 201(1–3): 25–52, 1999.

[3] G. Brightwell and S. Goodall. "The Number of Partial Orders of Fixed Width," *Order* 20(4): 333–345, 2003.

[4] G. Brightwell and P. Winkler. "Counting Linear Extensions is #P-Complete," *STOC* 1991.

[5] Y. Chen. "Decomposing DAGs into Disjoint Chains," *Database and Expert Systems Applications* 2007.

[6] T. M. Cover and J. A. Thomas. *Elements of information theory.* New York: John Wiley & Sons Inc., 1991.

[7] C. Daskalakis, R. M. Karp, E. Mossel, S. Riesenfeld, E. Verbin. "Sorting and Selection in Posets," ArXiv Report, 2007.

[8] M. E. Dyer, A. M. Frieze and R. Kannan. "A Random Polynomial Time Algorithm for Approximating the Volume of Convex Bodies," *Journal of the ACM,* 38(1): 1–17, 1991.

[9] U. Faigle and Gy. Turán. "Sorting and Recognition Problems for Ordered Sets," *SIAM J. Comput.* 17(1): 100–113, 1988.

[10] L. R., Jr., Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[11] M. Fredman. "How good is the information theory bound in sorting?" *Theor. Comput. Sci.* 1(4): 355-361, 1976.

[12] F. Fussenegger and H. N. Gabow. "A Counting Approach to Lower Bounds for Selection Problems," *Journal of the ACM* 26(2): 227–238, 1979.

[13] J. Kahn and J. H. Kim. "Entropy and Sorting," *STOC*, 178 – 187, 1992.

[14] J. Kahn and M. Saks. "Balancing poset extensions," *Order* 1(2): 113–126,1984.

[15] S. S. Kislitsyn, "A finite partially ordered set and its corresponding set of permutations," *Matematicheskie Zametki* 4(5): 511–518, 1968.

[16] D. Knuth. *The Art of Computer Programming: Sorting and Searching*, Massachusetts: Addison-Wesley, 1998.

[17] N. Linial. "The Information theoretic bound is good for merging," *SIAM J. Comput.* 13(4): 795-801, 1984.

[18] M. E. J. Newman, *SIAM Review* 45: 167–256, 2003.

[19] K. Onak and P. Parys. Generalization of Binary Search: Searching in Trees and Forest-Like Partial Orders. *FOCS* 2006.

[20] W. Trotter and S. Felsner, "Balancing pairs in partially ordered sets," *Combinatorics, Paul Erdos is Eighty* I: 145–157, 1993.

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