Counting Linear Extensions of Sparse Posets*

Kustaa Kangas Teemu Hankala Teppo Niinimäki Mikko Koivisto
University of Helsinki, Department of Computer Science,
Helsinki Institute for Information Technology HIIT, Finland
{jwkangas,tjhankal,tzniinim,mkhkoivi}@cs.helsinki.fi

Abstract

We present two algorithms for computing the number of linear extensions of a given n-element poset. Our first approach builds upon an $O(2^n n)$-time dynamic programming algorithm by splitting subproblems into connected components and recursing on them independently. The recursion may run over two alternative subproblem spaces, and we provide heuristics for choosing the more efficient one. Our second algorithm is based on variable elimination via inclusion–exclusion and runs in time $O(n^{t+4})$, where t is the treewidth of the cover graph. We demonstrate experimentally that these new algorithms outperform previously suggested ones for a wide range of posets, in particular when the posets are sparse.

1 Introduction

Determining the number of linear extensions of a given poset (equivalently, topological sorts of a directed acyclic graph) is a fundamental problem in order theory, with applications in areas such as sorting [Peczarski, 2004], sequence analysis [Mannila and Meek, 2000], convex rank tests [Morton *et al.*, 2009], preference reasoning [Lukasiewicz *et al.*, 2014], and learning probabilistic models from data [Wallace *et al.*, 1996; Niinimäki and Koivisto, 2013].

Brightwell and Winkler [1991] showed that exact counting of linear extensions is #P-complete and therefore not tractable for general posets unless P = NP. By dynamic programming over the lattice of upsets (see, e.g., De Loof *et al.* [2006]) linear extensions can be counted in time $O(|\mathcal{U}| \cdot w)$, where $\mathcal{U}$ is the set of upsets and w is the poset width. This implies the worst case bound $O(2^n n)$ for a poset on n elements, which to our knowledge is the best to date. Though exponential in n, the algorithm can be very fast for posets with few upsets. In particular, it holds that $|\mathcal{U}| = O(n^w)$, since every poset can be partitioned into w chains and an upset can be specified by the number of elements it contains from each chain. Hence the algorithm runs in polynomial time for bounded width. Conversely, the set U can be very large when the order relation is sparse, which raises the question if sparsity can be exploited for counting linear extensions faster.

In this work we present two approaches to counting linear extensions that target sparse posets in particular. In Section 2 we augment the dynamic programming algorithm by splitting each upset into connected components and then computing the number of linear extensions by recursing on each component independently. We also survey previously proposed recursive techniques for comparison.

In Section 3 we show that the problem is solvable in time $O(n^{t+4})$, where t is the treewidth of the cover graph. While our result stems from a well-known method of nonserial dynamic programming [Bertelè and Brioschi, 1972], known as *variable elimination* and by other names [Dechter, 1999; Koller and Friedman, 2009, Chap. 9] global constraints in the problem hamper its direct, efficient use. To circumvent this obstacle, we apply the inclusion–exclusion principle to translate the problem into multiple problems without such constraints, which are then solved by variable elimination.

In Section 4 experimental results are presented to compare the two algorithms against previously known techniques. We conclude with some open questions in Section 5.

1.1 Related Work

A number of approaches for breaking the counting task into subproblems have been considered before. Peczarski [2004] reports a significant gain on some families of posets by recursively decomposing them into connected components and so called admissible partitions; however, this procedure has an unknown asymptotic complexity. Li *et al.* [2005] present another algorithm that recursively splits a poset into connected components and so called static sets.

Besides bounded width, polynomial-time algorithms exist for several restricted families of posets such as series-parallel posets [Möhring, 1989], posets whose cover graph is a polytree [Atkinson, 1990], posets with a bounded decomposition diameter [Habib and Möhring, 1987], and N-free posets of a bounded activity [Felsner and Manneville, 2014].

Fully polynomial time randomized approximation schemes are known for estimating the number of linear extensions [Dyer *et al.*, 1991; Bubley and Dyer, 1999].

For *listing* all linear extensions there exist algorithms that spend $O(1)$ time per linear extension on average [Pruesse and Ruskey, 1994] and in the worst case [Ono and Nakano, 2007].

*This work was supported in part by the Academy of Finland, grants 125637, 255675, and 276864.

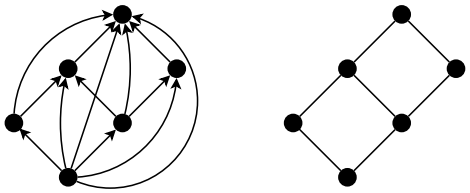

Figure 1: A poset (left) and its cover graph as a Hasse diagram. The reflexive arcs in the poset are omitted for clarity.

1.2 Preliminaries

A *partially ordered set* or a *poset* is a pair $(P, \leq_P)$, where P is a set and $\leq_P$ is an order relation on P, that is, a binary relation that is reflexive, antisymmetric, and transitive. The elements (a, b) of $\leq_P$ are denoted simply $a \leq_P b$. Elements $a, b \in P$ are *comparable* if $a \leq_P b$ or $b \leq_P a$, and otherwise *incomparable*, denoted $a||_P b$. We say that a is a *predecessor* of b, denoted $a <_P b$, if $a \leq_P b$ and $a \neq b$. Dually, b is called a *successor* of a. An element with no predecessors or successors is a *minimal* or *maximal* element, respectively.

We say that b *covers* a, denoted $a \prec_P b$, if $a <_P b$ and there is no $c \in P$ such that $a <_P c <_P b$. A poset is uniquely identified by the *cover relation* $\prec_P$ and can be presented by the *cover graph* (Figure 1), usually drawn as a Hasse diagram, where an edge upwards from a to b implies $a \prec_P b$.

A set of elements $A \subseteq P$ is called a *chain* if all pairs of elements in A are comparable and an *antichain* if no pairs are comparable. The *width* of P, denoted $w(P)$ or simply w, is the size of the largest antichain. A *downset* is a set of elements $D \subseteq P$ such that for all $b \in D, a \in P$ it holds that $a <_P b$ implies $a \in D$. Dually, an *upset* is a set $U \subseteq P$ such that for all $a \in U, b \in P$ it holds that $a <_P b$ implies $b \in U$.

A *linear extension* of an n-element poset P is a bijection $\sigma : P \to [n]$ that respects the order $\leq_P$, that is, $a \leq_P b$ implies $\sigma(a) \leq \sigma(b)$ for all $a, b \in P$. Here and henceforth the bracket notation $[n]$ denotes the set $\{1, \ldots, n\}$. An equivalent condition is that σ respects the cover relation, i.e., $a \prec_P b$ implies $\sigma(a) < \sigma(b)$. The number of linear extensions of P will be denoted $\ell(P)$.

We will typically identify a poset simply with the set of elements P. Further, any subset $A \subseteq P$ will be implicitly treated as a subposet $(A, \leq_A)$ of P, that is, for all $a, b \in A$ it holds that $a \leq_A b$ if and only if $a \leq_P b$.

2 Counting by Recursion

We begin with a brief survey of known methods for counting linear extensions by recursively decomposing the task into subproblems. For the remainder of the section consider an arbitrary non-empty poset P.

First, observe that each linear extension of P begins with some minimal element $x \in P$, and the number of extensions that begin with x equals $\ell(P \setminus x)$. Therefore, we have that

$$\ell(P) = \sum_{x \in \min(P)} \ell(P \setminus x)\,, \tag{1}$$

where $\min(P)$ denotes the set of minimal elements of P.

A direct evaluation of recurrence 1 corresponds to enumeration of all linear extensions. It is easy to see that the sets $U \subseteq P$ for which $\ell(U)$ is computed are exactly the upsets of P. If we store these intermediate results so that each is computed only once, we obtain the $O(|\mathcal{U}| \cdot w)$ time dynamic programming algorithm. The number of minimal elements is bounded by w and they are found in $O(w)$ time by simple bookkeeping.

By symmetry recurrence 1 still holds if x are taken over the maximal elements instead, in which case the algorithm will run over the *downsets* of P. Since downsets are exactly the complements of upsets, the running time is unchanged.

Second, we consider the *admissible partitions* used by Peczarski [2004], formalized slightly differently here. For an arbitrary element $x \in P$, we say that a partition of $P \setminus x$ into a pair of sets (D, U) is *admissible* if D is a downset that contains all predecessors of x (and U an upset that contains all successors of x). Equivalently, a partition is admissible if and only if there is at least one linear extension σ such that $\sigma(d) < \sigma(x) < \sigma(u)$ for all $d \in D$ and $u \in U$. Choosing a linear extension of P is equivalent to choosing such a partition and ordering D and U independently. Thus, we have that

$$\ell(P) = \sum_{(D,U)} \ell(D) \cdot \ell(U)\,, \tag{2}$$

where (D, U) runs over all admissible partitions.

While this holds for any choice of $x \in P$, not all choices are equally efficient. In general, it is preferable to choose an x such that the number of admissible partitions is minimized, which is exactly when $P \setminus x$ has the maximum number of elements comparable with x.

Third, we consider the decomposition into *static sets*. We say that a non-empty set of elements $S \subset P$ is a *static set* if every element in S is comparable with every element in $P \setminus S$ and if no proper subset of S has this property. It is known [Li *et al.*, 2005] that either P has no static sets, or there exists a unique partition of P into static sets $S_1, \ldots, S_k$. If the partition exists, a linear extension is obtained by ordering each S_i independently, and therefore it holds that

$$\ell(P) = \prod_{i=1}^{k} \ell(S_i)\,. \tag{3}$$

Li *et al.* give an efficient algorithm that either finds the partition or determines that it does not exist.

Finally, if the graph representation of P is disconnected, i.e., if P can be partitioned into sets A and B such that $a||_P b$ for all $a \in A$ and $b \in B$, then taking a linear of extension of P is equivalent to ordering A and B independently and then interleaving them. Thus, in this case we have

$$\ell(P) = \ell(A) \cdot \ell(B) \cdot \binom{|P|}{|A|}\,. \tag{4}$$

Each of the rules 1–4 breaks the poset into one or more subposets whose linear extensions are counted recursively. It is easy to see that some rules are more effective for breaking certain kinds of subposets into parts than others, which suggests an algorithm that uses a combination of multiple rules.

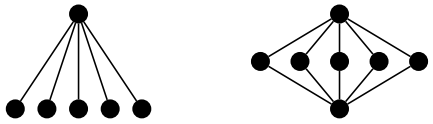

Figure 2: Left: A poset where all upsets are connected. Right: A poset where all upsets *and* downsets are connected.

One such algorithm was given by Peczarski [2004] who applies rule 4 whenever the poset is disconnected and rule 2 otherwise. Li *et al.* [2005] propose another algorithm, which applies rules 4 and 3 (in this order) when possible and falls back to enumeration when neither rule is applicable.

We propose an algorithm that augments the dynamic programming over upsets by applying rule 4 whenever the poset is disconnected and rule 1 in all other cases. When combining these two rules, we observe that it can make an exponential difference whether rule 1 is applied to minimal or maximal elements. For instance, consider a poset on n elements with $n-1$ minimal elements (Figure 2, left) and notice that all of its upsets are connected. Therefore, if rule 1 is applied to minimal elements, rule 4 will never be applicable and the algorithm needs to consider $O(2^n)$ upsets separately. On the other hand, if we remove the lone maximal element, the remainder of the poset breaks into singletons, and linear extensions are efficiently counted by applying rule 4.

Given an arbitrary poset P, it is not obvious which choice of rule 1 leads to a smaller number of subproblems that need to be solved. A very simple heuristic is to remove minimal elements if P has less minimal than maximal elements, and vice versa. We also propose a second heuristic that computes an estimate $e(P)$ of the number of subproblems and makes the choice for which the estimate is smaller. The estimate is computed recursively as follows. If P is connected, we set $e(P) = 2^{|M|} + e(P \setminus M)$, where M is the set of minimal (respectively, maximal) elements of P. If P has the connected components $P_1, \ldots, P_k$, we set $e(P) = e(P_1) + \cdots + e(P_k)$. The intuition here is that for a connected poset we must (roughly) consider all subsets of minimal elements and then solve the subproblems in the remaining poset.

In certain cases (e.g. Figure 2, right) it can be effective to alternate the choice between minimal and maximal elements. However, our preliminary experiments suggest that on most posets it is preferable to make the choice once only and then apply it consistently. Intuitively this is because changing the choice breaks the property that all subproblems are either upsets or downsets, thus expanding the total space of possible subproblems. In practice this means that the recursion cannot reuse already computed subproblems as often, thus requiring a greater number of subproblems to be solved.

We also consider augmenting the algorithm further by applying rule 3 when possible. This can also be seen as an improvement of the algorithm of Li *et al.* where the raw enumeration is replaced by rule 1.

In Section 4 we present an experimental comparison of these proposals against other recursive algorithms. We will show in particular that on randomly generated posets both of the suggested heuristics almost always pick the better choice.

3 Counting by Variable Elimination

In this section we show the following result.

Theorem 1 *Given a poset P, the number $\ell(P)$ can be computed in time $O(n^{t+4})$, where $n = |P|$ and t is the treewidth of the cover graph of P.*

We give a proof in the form of an algorithm. The key idea is to formulate the counting problem as a sum of products that factorize over the edges in the cover graph, and then apply the variable elimination scheme (see, e.g., Dechter [1999]) for computing the sum.

Let $(P, \leq_P)$ be a poset on n elements. To simplify notation, we assume without loss of generality that $P = [n]$. On all (not necessarily bijective) mappings of form $\sigma : [n] \to [n]$, define the function

$$\Phi(\sigma) = \prod_{\substack{i,j \in [n] \\ i \prec_P j}} \varphi(\sigma(i), \sigma(j)) \,,$$

where we define $\varphi(x, y) = 1$ if $x < y$ and $\varphi(x, y) = 0$ otherwise. Recall that a linear extension of P is a bijection $\sigma : [n] \to [n]$ that respects the cover relation $\prec_P$. Therefore, for all bijections $\sigma : [n] \to [n]$ it holds that $\Phi(\sigma) = 1$ if σ is a linear extension and $\Phi(\sigma) = 0$ otherwise. As a consequence, we have that

$$\ell(P) = \sum_{\substack{\sigma:[n]\to[n] \\ \text{bijection}}} \Phi(\sigma) \,.$$

This form does not immediately admit variable elimination due to the global constraint that σ must be bijective. By applying the inclusion–exclusion principle, we can rewrite the sum as

$$= \sum_{X \subseteq [n]} (-1)^{n-|X|} \sum_{\tau:[n]\to X} \Phi(\tau) \,,$$

which removes the undesired constraint from the inner sum but introduces a summation over all subsets of $[n]$. We now make use of the property of Φ that for all $X \subseteq [n]$ of fixed size $k = |X|$ there are equally many $\tau : [n] \to X$ such that $\Phi(\tau) = 1$. Hence, it suffices to sum over all possible k:

$$= \sum_{k=1}^{n} \binom{n}{k} (-1)^{n-k} \sum_{\tau:[n]\to[k]} \Phi(\tau) \,.$$

The inner sum is now of the desired form

$$\sum_{\tau_1,\ldots,\tau_n} \prod_{i \prec_P j} \varphi(\tau_i, \tau_j) \,, \tag{5}$$

where we have expressed the summation over $\tau : [n] \to [k]$ as a sum over the variables $\tau_1, \ldots, \tau_n$, each of which runs over the values $1, \ldots, k$. A summation problem of this form is associated with an *interaction graph*, an undirected graph on the variables $\tau_1, \ldots, \tau_n$, where two variables are joined by an edge if there is at least one function $\varphi(\tau_i, \tau_j)$ in the product that depends on both of them. It is well known that with variable elimination such a sum can be computed in time polynomial in n and exponential in the treewidth t of the interaction graph, which in this case is exactly (the undirected variant of) the cover graph of P.

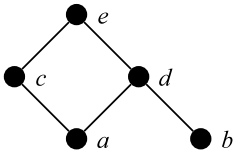

Figure 3: A poset with a cover graph of treewidth 2.

To establish the exact running time of our algorithm, we illustrate variable elimination with a simple example. Consider a poset on the elements $\{a, b, c, d, e\}$ with a cover graph as shown in Figure 3. In this example the summation task is

$$\sum_{d,e,a,b,c} \varphi(a,c)\,\varphi(a,d)\,\varphi(b,d)\,\varphi(c,e)\,\varphi(d,e)\;,$$

where we have picked the *elimination ordering* d, e, a, b, c for the variables. Variables are summed out in the reverse order from right to left. To eliminate c, we organize the sum as

$$= \sum_{d,e,a,b} \varphi(a,d)\,\varphi(b,d)\,\varphi(d,e) \sum_{c} \varphi(a,c)\,\varphi(c,e)\;,$$

placing every function that does not depend on c outside the inner sum. The set of functions that do depend on c is called the *bucket* of c. Carrying out the inner summation produces a new function $\lambda_c(a,e) = \sum_c \varphi(a,c)\,\varphi(c,e)$ that depends on every variable appearing in the bucket except c.

We put the new function in place of the sum and continue eliminating variables in this manner,

$$\begin{aligned}
&= \sum_{d,e,a} \varphi(a,d)\,\varphi(d,e)\,\lambda_c(a,e) \sum_{b} \varphi(b,d)\\
&= \sum_{d,e} \varphi(d,e)\,\lambda_b(d) \sum_{a} \varphi(a,d)\,\lambda_c(a,e)\\
&= \sum_{d} \lambda_b(d) \sum_{e} \varphi(d,e)\,\lambda_a(d,e)\\
&= \sum_{d} \lambda_b(d)\,\lambda_e(d)\\
&= \lambda_d\,,
\end{aligned}$$

until we are left with a constant function whose value equals expression 5.

We first analyze the time required to eliminate a single variable x. Let m be the number of functions in the bucket of x and let q be the number of variables at least one of the functions depends on. From a computational point of view every function is simply an array that contains a value for each instantiation of its domain. Hence, eliminating x to produce λ_x involves iterating over all k^q instantiations of the q variables, and for each instantiation the product of all m functions is computed. Thus eliminating x requires $O(k^q \cdot m)$ time.

It is immediate that $m = O(n)$, since in the beginning at most $n-1$ functions depend on x and variable elimination produces at most $n-1$ other such functions. On the other hand, the maximum value of q over all variables depends on the elimination ordering and is called its *induced width*. An elimination ordering is called *optimal* if it has the minimum induced width among all possible orderings. It is known that the induced width of an optimal elimination ordering is exactly $t+1$ [Dechter, 1999]. Thus, given such an ordering, eliminating all n variables takes $O(n^2 \cdot k^{t+1})$ time. Due to the inclusion–exclusion, expression 5 is evaluated by variable elimination for all $k = 1, \ldots, n$, which brings the final running time to $O(n^{t+4})$.

It remains to note that an optimal elimination ordering can be found in $O(n^{t+2})$ time [Arnborg *et al.*, 1987].

4 Experiments

We have implemented all algorithmic techniques described in Sections 2 and 3 for experimental evaluation. The following five (combinations of) techniques were considered:

- R1: Dynamic programming over upsets (rule 1 only).
- R14: Our proposal (applies rules 1 and 4).
- R24: The algorithm of Peczarski (rules 2 and 4).
- R134: Applies rules 1, 3, and 4.
- VEIE: Variable elimination via inclusion–exclusion.

For R14 we consider the following two variants:

- R14-a: The simple heuristic is used to decide whether rule 1 is applied to minimal or maximal elements.
- R14-b: The recursive heuristic is used instead.

For comparison, we also consider R14-best and R14-worst, two hypothetical variants of R14 that always make the best or the worst choice, respectively.

All posets used in these experiments were produced by randomly sampling directed acyclic graphs (DAGs) and taking for each DAG the corresponding partial order. The program `LEcount`,[1] comprising all implementations, was written in C++ and run on machines with Intel Xeon E5540 CPUs. Hashing was used in all implementations for storing the computed subproblems. All algorithms were given up to 20 minutes of CPU time and 30 GB of RAM on each poset.

4.1 Recursive Algorithms

In the first set of experiments we compare the first four algorithms against each other.

We generated two classes of sparse DAGs, parameterized by the number of vertices $n \in \{30, 32, 34, \ldots, 100\}$ and a density parameter $k \in \{2, 3, 4, 5, 6\}$. In both classes a DAG was generated by picking a random ordering on the vertices and then adding edges compatible with the ordering. In the first class k is the expected average degree, achieved by adding each possible edge with probability $k/(n-1)$. In the second class k is the maximum indegree, achieved by choosing for each vertex at most k parents among the preceding vertices in the ordering. This class is motivated by applications in learning Bayesian networks [Niinimäki and Koivisto, 2013], where the indegree is typically bounded.

We also generated a third class of dense bipartite graphs on $n \in \{30, 32, 34, \ldots, 60\}$ vertices and a density parameter $p \in \{0.2, 0.5\}$. These were produced by splitting the vertices

[1] The `LEcount` program and all experiment posets are available at `www.cs.helsinki.fi/u/jwkangas/lecount/`.

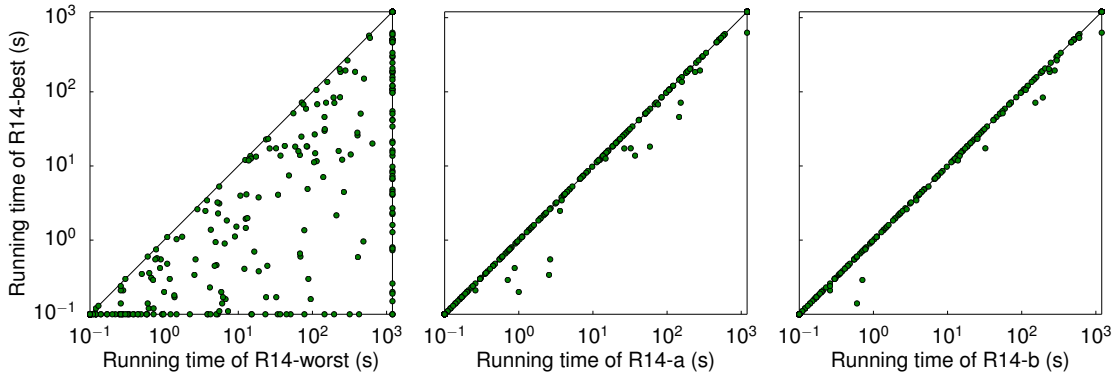

Figure 4: Comparison of running times between the variants of R14 on posets of all three classes. Left: the difference between making the better or worse choice between minimal and maximal elements. In the middle and right is shown how close the two heuristics are to optimal behavior. Cases where the time or memory limit were exceeded are shown as 20 minutes.

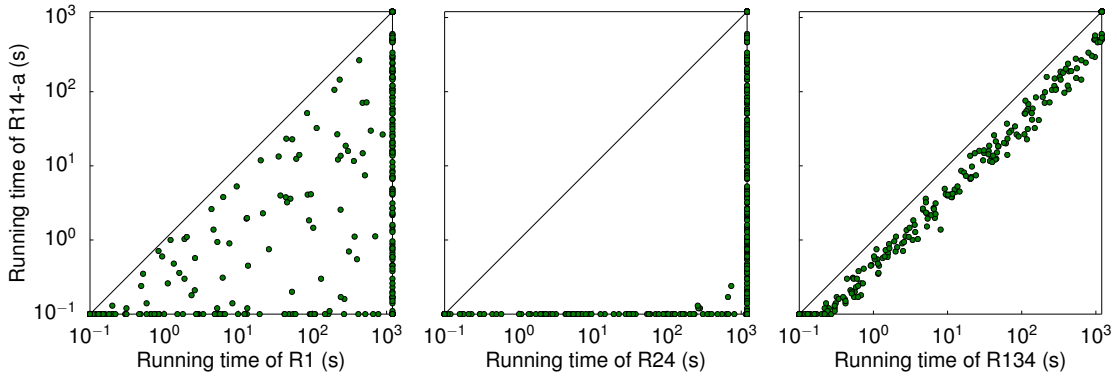

Figure 5: The running time of R14-a compared to R1, R24, and R134 on all three classes of posets.

into two sets A and B of size $n/2$ and adding the edge (a, b) for all $a \in A$ and $b \in B$ with probability p.

A comparison between the variants of R14 is presented in Figure 4. As suggested earlier, the choice between removing minimal or maximal elements has a huge impact on performance. It turns out, however, that even the simple heuristic is able to pick the better choice for a vast majority of random posets. The recursive heuristic improves upon this even further and appears to deviate less from the better choice even when it makes a mistake.

In light of this we only compare the variant R14-a to the other recursive algorithms (Figures 5 and 6). We observe that R14-a outperforms the other algorithms on every poset of the three classes. It beats R24 by a large margin and greatly improves upon the baseline set by R1, suggesting that rule 1 is in general better equipped for breaking a poset into connected components than rule 2. In algorithm R134 we used the simple heuristic for rule 1 to make it directly comparable with R14-a. A closer analysis of this algorithm reveals that rule 3 was applicable only to a handful of subproblems and thus could not compensate for the overhead of detecting static sets. We conclude that the addition of rule 3 does not improve upon the performance on R14.

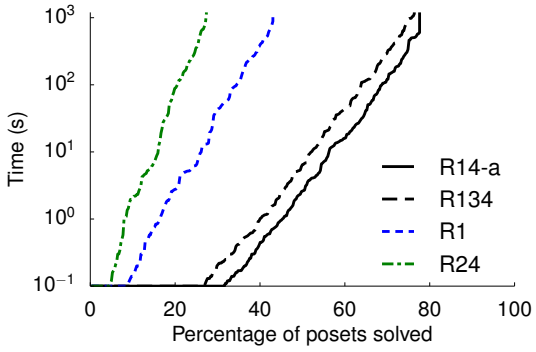

Figure 6: The number of posets on which the recursive algorithms finished computation within a certain amount of time.

The advantage of R14 over the other algorithms is most pronounced on the posets of bounded indegree (Figure 7). For the dense bipartite graphs its behavior is closer to R1 as most of the subposets are connected.

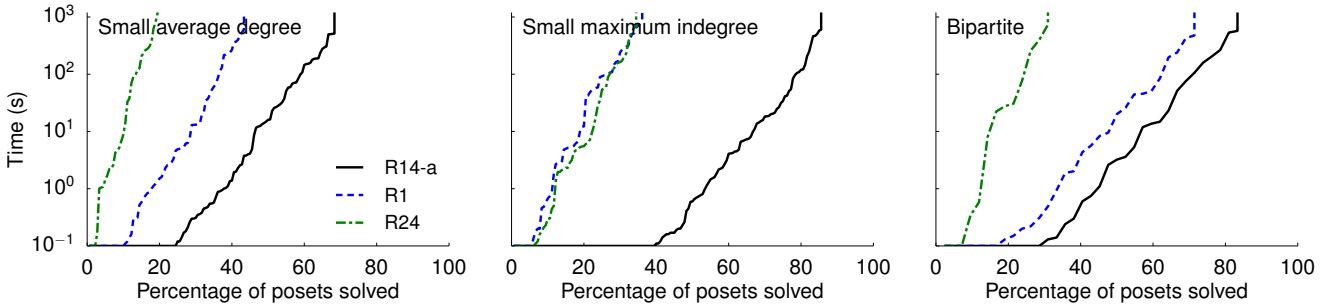

Figure 7: The number of posets on which R14, R1, and R24 finished within a certain time, on each class of posets separately.

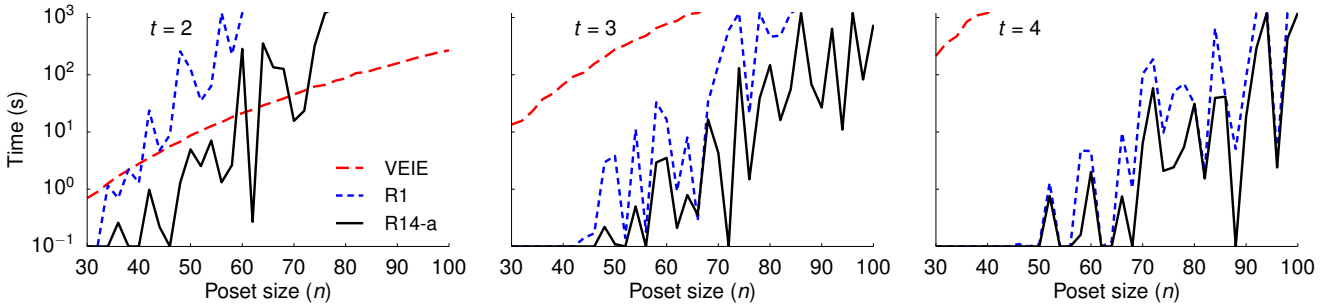

Figure 8: The running time of VEIE on grid trees with respect to the number of elements n, compared to R14 and R1.

4.2 Variable Elimination

In the second set of experiments we compare the variable elimination algorithm VEIE against R14 and R1 on a set of grid trees on $n \in \{30, 32, 34, \ldots, 100\}$ vertices and treewidth $t \in \{2, 3, 4\}$. Such a grid tree is constructed by randomly joining t by t grid posets along the edges, orienting the edges so that no directed cycles are introduced. Finding an optimal elimination ordering for a grid tree is easy and this step is omitted from the evaluation.

For a fixed value of t the running time of VEIE exhibits the expected polynomial behavior with respect to n (Figure 8). By contrast, the recursive algorithms are highly sensitive to other features of the poset structure and therefore display more erratic running times. On average their behavior is still exponential, allowing VEIE to surpass them on sufficiently large posets of low treewidth. We can observe this happening for $t = 2$, but for larger treewidth even R1 remains faster within the 20-minute time limit. We remark that the recursive algorithms typically run out of memory around this point, thus making VEIE the most viable option thereafter.

5 Conclusion

We have proposed two algorithms for counting linear extensions of posets, exploiting recursive decomposition into connected components and low treewidth, respectively. We demonstrated with experiments that the recursive algorithm beats previously proposed methods on a range of both sparse and dense posets, and that for large posets of low treewidth our second algorithm can be even faster. We also showed that simple heuristics often suffice to determine the better variant of the recursive algorithm.

As a conclusion, we raise some questions for future work. First, we note that one can easily construct specific examples where our recursive algorithm performs poorly compared to the other techniques. For instance, the poset in Figure 2, right, and larger posets with a similar structure, can be effectively decomposed into admissible partitions or static sets, whereas our algorithm requires an exponential time. While this is a very extreme example, it is natural to ask if there are notable classes of posets where the counting problem is nontrivial but solved effectively by applying multiple recursive techniques together. In particular, does any such class benefit from using both variants of rule 1 together, an approach that we rejected in general? Can further generalizations of rule 1 (e.g. Edelman *et al.* [1989]) yield even faster algorithms?

Second, the derivation of our $O(n^{t+4})$ time algorithm required the trick of running variable elimination through inclusion–exclusion to avoid global constraints. Is this the natural best way to deal with the constraints or can a direct dynamic programming over a tree decomposition yield an equally or more efficient algorithm?

Lastly, in terms of parameterized complexity [Downey and Fellows, 2012], the running time of form $n^{f(t)}$ places the problem of counting linear extensions in the class XP when parameterized by the treewidth of the cover graph. A strictly better running time of form $f(t) \cdot n^{O(1)}$ would place the problem in the class FPT or *fixed-parameter tractable*. Does such an algorithm exist for the treewidth of the cover graph or some other poset parameter?

Acknowledgments

We would like to thank the anonymous reviewers for expanding our knowledge on related literature and other valuable comments for improving the presentation.

References

[Arnborg *et al.*, 1987] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[Atkinson, 1990] M. D. Atkinson. On computing the number of linear extensions of a tree. *Order*, 7(1):23–25, 1990.

[Bertelè and Brioschi, 1972] U. Bertelè and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[Brightwell and Winkler, 1991] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.

[Bubley and Dyer, 1999] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201(13):81–88, 1999.

[De Loof *et al.*, 2006] K. De Loof, H. De Meyer, and B. De Baets. Exploiting the lattice of ideals representation of a poset. *Fundamenta Informaticae*, 71(2,3):309–321, 2006.

[Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(12):41–85, 1999.

[Downey and Fellows, 2012] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 2012.

[Dyer *et al.*, 1991] M. Dyer, A. Frieze, and R. Kannan. A random polynomial-time algorithm for approximating the volume of convex bodies. *J. ACM*, 38(1):1–17, 1991.

[Edelman *et al.*, 1989] P. Edelman, T. Hibi, and R. P. Stanley. A recurrence for linear extensions. *Order*, 6(1):15–18, 1989.

[Felsner and Manneville, 2014] S. Felsner and T. Manneville. Linear extensions of N-free orders. *Order*, 32(2):147–155, 2014.

[Habib and Möhring, 1987] M. Habib and R. H. Möhring. On some complexity properties of N-free posets and posets with bounded decomposition diameter. *Discrete Mathematics*, 63(2):157–182, 1987.

[Koller and Friedman, 2009] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT press, 2009.

[Li *et al.*, 2005] W. N. Li, Z. Xiao, and G. Beavers. On computing the number of topological orderings of a directed acyclic graph. *Congressus Numerantium*, 174:143–159, 2005.

[Lukasiewicz *et al.*, 2014] T. Lukasiewicz, M. V. Martinez, and G. I. Simari. Probabilistic preference logic networks. In *Proc. of the 21st European Conference on Artificial Intelligence (ECAI)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 561–566. IOS Press, 2014.

[Mannila and Meek, 2000] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proc. of the Sixth International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 161–168. ACM, 2000.

[Möhring, 1989] R. H. Möhring. *Algorithms and Order*, chapter Computationally tractable classes of ordered sets, pages 105–193. Springer, 1989.

[Morton *et al.*, 2009] J. Morton, L. Pachter, A. Shiu, B. Sturmfels, and O. Wienand. Convex rank tests and semigraphoids. *SIAM J. on Discrete Mathematics*, 23(3):1117–1134, 2009.

[Niinimäki and Koivisto, 2013] T. Niinimäki and M. Koivisto. Annealed importance sampling for structure learning in Bayesian networks. In *Proc. of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI, 2013.

[Ono and Nakano, 2007] A. Ono and S.-I. Nakano. Constant time generation of linear extensions. In *Proc. of the First Workshop on Algorithms and Computation (WALCOM)*, pages 151–161. Bangladesh Academy of Sciences, 2007.

[Peczarski, 2004] M. Peczarski. New results in minimum-comparison sorting. *Algorithmica*, 40(2):133–145, 2004.

[Pruesse and Ruskey, 1994] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM J. Computing*, 23(2):373–386, 1994.

[Wallace *et al.*, 1996] C. S. Wallace, K. B. Korb, and H. Dai. Causal discovery via MML. In *Proc. of the 13th International Conference on Machine Learning (ICML)*, pages 516–524. Morgan Kaufmann, 1996.