Department of Computer Science
Series of Publications A
Report A-2019-9

Counting and Sampling Directed Acyclic Graphs for Learning Bayesian Networks

Topi Talvitie

Doctoral dissertation, to be presented for public criticism with the permission of the Faculty of Science of the University of Helsinki in Auditorium CK112, Exactum building, Pietari Kalmin katu 5, on November 21st, 2019 at twelve o'clock noon.

University of Helsinki
Finland

Supervisors
Mikko Koivisto, University of Helsinki, Finland
Valentin Polishchuk, Linköping University, Sweden

Pre-examiners
Mark Huber, Claremont McKenna College, USA
Jack Kuipers, ETH Zürich, Switzerland

Opponent
Kuldeep Meel, National University of Singapore

Custos
Mikko Koivisto, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: https://www.helsinki.fi/en/computer-science
Telephone: +358 2941 911

ISSN 1238-8645
ISBN 978-951-51-5609-9 (paperback)
ISBN 978-951-51-5610-5 (PDF)
Helsinki 2019
Unigrafia

Counting and Sampling Directed Acyclic Graphs for Learning Bayesian Networks

Topi Talvitie

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
topi.talvitie@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2019-9
Helsinki, November 2019, 70+54 pages
ISSN 1238-8645
ISBN 978-951-51-5609-9 (paperback)
ISBN 978-951-51-5610-5 (PDF)

Abstract

Bayesian networks are probabilistic models that represent dependencies between random variables via directed acyclic graphs (DAGs). They provide a succinct representation for the joint distribution in cases where the dependency structure is sparse. Specifying the network by hand is often unfeasible, and thus it would be desirable to learn the model from observed data over the variables. In this thesis, we study computational problems encountered in different approaches to learning Bayesian networks. All of the problems involve counting or sampling DAGs under various constraints.

One important computational problem in the fully Bayesian approach to structure learning is the problem of sampling DAGs from the posterior distribution over all the possible structures for the Bayesian network. From the typical modeling assumptions it follows that the distribution is modular, which means that the probability of each DAG factorizes into per-node weights, each of which depends only on the parent set of the node. For this problem, we give the first exact algorithm with a time complexity bound exponential in the number of nodes, and thus polynomial in the size of the input, which consists of all the possible per-node weights. We also adapt the algorithm such that it outperforms the previous methods in the special case of sampling DAGs from the uniform distribution.

We also study the problem of counting the linear extensions of a given partial order, which appears as a subroutine in some importance sampling

methods for modular distributions. This problem is a classic example of a #P-complete problem that can be approximately solved in polynomial time by reduction to sampling linear extensions uniformly at random. We present two new randomized approximation algorithms for the problem. The first algorithm extends the applicable range of an exact dynamic programming algorithm by using sampling to reduce the given instance into an easier instance. The second algorithm is obtained by combining a novel, Markov chain-based exact sampler with the Tootsie Pop algorithm, a recent generic scheme for reducing counting into sampling. Together, these two algorithms speed up approximate linear extension counting by multiple orders of magnitude in practice.

Finally, we investigate the problem of counting and sampling DAGs that are Markov equivalent to a given DAG. This problem is important in learning causal Bayesian networks, because distinct Markov equivalent DAGs cannot be distinguished only based on observational data, yet they are different from the causal viewpoint. We speed up the state-of-the-art recursive algorithm for the problem by using dynamic programming. We also present a new, tree decomposition-based algorithm, which runs in linear time if the size of the maximum clique is bounded.

Computing Reviews (2012) Categories and Subject Descriptors:

Mathematics of computing → Probabilistic algorithms
Theory of computation → Design and analysis of algorithms
Theory of computation → Generating random combinatorial structures
Theory of computation → Random walks and Markov chains
Computing methodologies → Bayesian network models

General Terms:
algorithms, approximation, sampling, directed acyclic graphs

Additional Key Words and Phrases:
linear extensions, modular distributions, Markov equivalence

Acknowledgements

I would like to start by expressing my deepest gratitude to my advisor Mikko Koivisto for his guidance throughout my PhD studies. His optimistic attitude and deep insight of the field has been an invaluable driving force behind the research in this thesis. I also thank my co-authors Teppo Niinimäki, Kustaa Kangas, and Aleksis Vuoksenmaa, with whom it has been a great pleasure to work. I am grateful to my co-advisor Valentin Polishchuk for his guidance during my early steps into algorithm research and his instrumental role in initiating this PhD project.

I would also like to thank the pre-examiners Mark Huber and Jack Kuipers as well as all the anonymous reviewers of the papers that make up this thesis for their efforts to ensure the quality of the work with their constructive feedback.

During these years, the Department of Computer Science has become like a second home to me. Many people in the department deserve thanks for helping me along the way. Particularly, I would like to mention Antti Laaksonen, who has always been there with advice and encouragement, and Paul Saikko, whose peer support during our concurrent PhD studies I greatly appreciate. I also thank the community of the computer science student room Gurula for all the fun distractions.

I am grateful for the financial support provided by the University of Helsinki Research Foundation that has allowed me to concentrate on my research full-time. Additional funding was provided by the Doctoral Programme in Computer Science (DoCS) and the Academy of Finland. The experimental part of the research was made possible by the computational resources provided by the department and the IT for Science group.

Finally, I want to thank my parents Virpi Talvitie and Lauri Manninen for all their love and support over the years.

Helsinki, November 2019
Topi Talvitie

Contents

Original publications

This thesis is based on the following original publications, referred to as Papers I–V in the text. The papers are reprinted in the end of the thesis.

I. Topi Talvitie, Teppo Niinimäki, and Mikko Koivisto. The mixing of Markov chains on linear extensions in practice. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 524–530, 2017.

II. Topi Talvitie, Kustaa Kangas, Teppo Niinimäki, and Mikko Koivisto. Counting linear extensions in practice: MCMC versus exponential Monte Carlo. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 1431–1438, 2018.

III. Topi Talvitie, Kustaa Kangas, Teppo Niinimäki, and Mikko Koivisto. A scalable scheme for counting linear extensions. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 5119–5125, 2018.

IV. Topi Talvitie, Aleksis Vuoksenmaa, and Mikko Koivisto. Exact sampling of directed acyclic graphs from modular distributions. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

V. Topi Talvitie and Mikko Koivisto. Counting and sampling Markov equivalent directed acyclic graphs. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7984–7991, 2019.

Chapter 1

Introduction

In machine learning, one of the main tasks is to design algorithms that can learn a model of a given phenomenon based on data obtained from it. The model should capture the essential features of the phenomenon so that it can be used to predict its future behavior or reason about its structure. Many recent practical advancements in artificial intelligence are due to the development of machine learning, along with the increasing availability of real-world data and computational resources. In this thesis, our goal is to learn a *probabilistic model*, that is, a model for the joint probability distribution of a set of random variables.

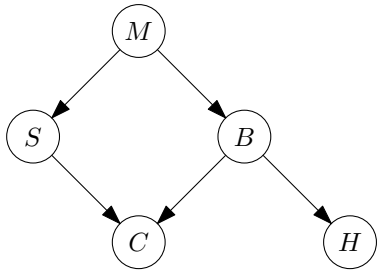
In any machine learning task, the practical performance depends heavily on the chosen model and learning method. For modeling the joint distribution of a set of discrete variables, the simplest probabilistic model is the full probability table, where we specify the probability for every configuration of values for the random variables as a *parameter* for the model. To learn such a model, we can just estimate each parameter by looking at the frequency of that configuration in the data. The problem with this model is that the size of the probability table grows exponentially in the number of random variables, which means that storing the table will often be impossible. From the learning perspective, we also have the problem that the learned model does not generalize well: unless there are only a handful of random variables, most of the configurations will not be observed in the data because the probability table is far larger than the set of training data points.

While in theory full probability tables have the minimum possible number of parameters to express general unstructured joint distributions, most practical joint distributions have a special internal structure: there are many *conditional independencies* between the random variables. The more conditional independencies we have, the fewer parameters we need to ex-

press the distribution. The set of conditional independencies is typically encoded to a *(probabilistic) graphical model*, which is a graph where nodes correspond to random variables and edges are used to express dependencies between them. The most common types of graphical models are *Bayesian networks* and *Markov random fields*, in which the independencies are encoded by directed graphs and undirected graphs, respectively. Each of them are equivalent to the full-table model if the graph is complete, but to reduce the number of parameters, the model should be as sparse as possible, meaning that it encodes as many conditional independencies as possible. Because the types of graphical models differ in the sets of conditional independencies they can express, they also differ in the number of parameters required to express a given distribution.

The contributions of this thesis are related to Bayesian networks, which use *directed acyclic graphs* (DAGs) on the set of random variables to encode the conditional independencies (see the example in Figure 1.1). The marginal joint probability factorizes into a product of conditional probabilities of each random variable conditioned by its set of parent variables in the DAG. Because of this, we can parameterize the probability distribution by specifying these conditional probabilities in probability tables. As the size of a probability table grows exponentially in the number of involved variables, this parameterization will have significantly fewer parameters than the full probability table if the DAG is sparse in the sense that all nodes have only few parents.

While Bayesian networks can be and often are constructed manually by a domain expert, we are interested in the machine learning approach to constructing them: Given a set of training data points, each being a joint assignment of the random variables, we should learn the DAG structure and the parameters of the Bayesian network. The learned network should generalize the training data, which means that sampling from the joint distribution of the network should generate similar data as the training data. The two main paradigms in Bayesian network learning are constraint-based and score-based learning; most methods use either one of these or a hybrid of the two. In constraint-based learning, we detect conditional independencies in the distribution by performing statistical independence tests on the data, and based on them construct the DAG structure. In score-based learning, each network structure is assigned a score based on how well it fits the data and how simple the model is. The learning method then finds one or more structures with large scores. Learning the DAG structure is typically the difficult part: after that, the parameters can be straightforwardly obtained from statistics collected from the data.



$\Pr(M = \mathrm{T})$
0.9

m	$\Pr(S = \mathrm{T} \mid M = m)$
T	0.2
F	0.05

m	$\Pr(B = \mathrm{T} \mid M = m)$
T	0.8
F	0.2

s	b	$\Pr(C = \mathrm{T} \mid S = s, B = b)$
T	T	0.8
T	F	0.8
F	T	0.8
F	F	0.05

b	$\Pr(H = \mathrm{T} \mid B = b)$
T	0.8
F	0.6

$$\Pr(M = m, S = s, B = b, C = c, H = h) = \Pr(M = m) \cdot \Pr(S = s \mid M = m) \cdot \Pr(B = b \mid M = m) \cdot \Pr(C = c \mid S = s, B = b) \cdot \Pr(H = h \mid B = b)$$

(a) Bayesian network

$\Pr(M=m, S=s, B=b, C=c, H=h)$					
m	s	b	c	h	
T	T	T	T	T	0.09216
T	T	T	T	F	0.02304
T	T	T	F	T	0.02304
T	T	T	F	F	0.00576
T	T	F	T	T	0.01728
T	T	F	T	F	0.01152
T	T	F	F	T	0.00432
T	T	F	F	F	0.00288
T	F	T	T	T	0.36864
T	F	T	T	F	0.09216
T	F	T	F	T	0.09216
T	F	T	F	F	0.02304
T	F	F	T	T	0.00432
T	F	F	T	F	0.00288
T	F	F	F	T	0.08208
T	F	F	F	F	0.05472
F	T	T	T	T	0.00064
F	T	T	T	F	0.00016
F	T	T	F	T	0.00016
F	T	T	F	F	0.00004
F	T	F	T	T	0.00192
F	T	F	T	F	0.00128
F	T	F	F	T	0.00048
F	T	F	F	F	0.00032
F	F	T	T	T	0.01216
F	F	T	T	F	0.00304
F	F	T	F	T	0.00304
F	F	T	F	F	0.00076
F	F	F	T	T	0.00228
F	F	F	T	F	0.00152
F	F	F	F	T	0.04332
F	F	F	F	F	0.02888

(b) Full table

Figure 1.1: An artificial example of a Bayesian network model of five binary variables [53] is shown in (a), consisting of the DAG structure and the conditional probability tables that make up the factorization of the joint probability. Each variable indicates the presence (T) or absence (F) of a medical condition (**M**etastatic cancer, **S**erum calcium increased, **B**rain tumor, **C**oma, **H**eadaches) in a patient. For example, the only parent of B is M in this structure, which means that for the factorization we need the probability table of B conditional on M; the DAG encodes conditional independencies such as the independence of B and S given M. The full probability table of the same distribution is shown in (b) for comparison. For this particular DAG structure, we only need to specify 11 probabilities as parameters in the conditional probability tables, compared to $2^5 - 1 = 31$ parameters in a full-table model.

In this thesis, we consider computational problems related to score-based learning, particularly in the case that the score defines a probability distribution over the DAG structures. One important score like this is the Bayesian score [52], which is obtained as the posterior probability after applying the Bayesian formula to update our earlier distribution on DAGs (which encodes our prior knowledge about the structure) by introducing the training data as evidence. We assume that the score is *modular*, meaning

that it factorizes into a product of node-wise factors that only depend on the set of parents of the node. This assumption makes it possible to solve certain computational problems on the score faster. One important and well-studied [15, 87, 92] problem like this is the problem of finding the DAG that maximizes the modular score; this optimum DAG is then typically used as the structure for the learned Bayesian network.

While the structure that maximizes the score is the best guess we have, relying solely on it may lead to bias: while it has the largest probability, the large number of possible DAGs means that its probability may still be insignificantly small. A practical way to take into account the rest of the probability mass is to draw samples from the posterior distribution. After this, the sampled structures can be used as representatives of the whole distribution. One approach to sampling DAGs from a modular distribution first samples DAGs from a biased distribution using a Markov chain, after which the bias is corrected [69]. The bias correction phase requires a subroutine for counting the topological orderings of a DAG, that is, the orderings of the set of nodes of the DAG such that the parent nodes of each node appear before it in the ordering. This problem, known as *counting linear extensions*, along with the related sampling problem, is the first one of the computational problems we study:

Problem A. Count and sample the linear extensions of a given partial order.

We also consider the DAG sampling problem directly without the reduction to linear extension counting:

Problem B. Sample DAGs from a modular distribution given in the factorized form.

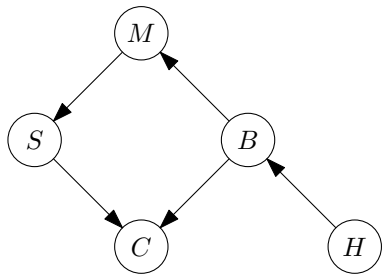
One advantage of Bayesian networks over undirected models, such as Markov random fields, is that they can be used to express causal relationships between variables in a natural way. This is done by encoding the causal mechanism of the phenomenon into the network: the parents of a variable are its direct causes, and the parameters in its probability table specify the mechanism of determining the value of the variable from the values of the parents. While Bayesian networks may be used to represent causal relationships, not all Bayesian networks are causal. For example, while we consider the DAG structure in Figure 1.1 to be causally correct, the DAG in Figure 1.2 can encode exactly the same joint distributions while having the direction of two direct causal relationships flipped. We call this kind of equivalence between DAGs *Markov equivalence*.

When learning the network structure based on training data gathered by passively observing the phenomenon, we cannot distinguish between Markov equivalent DAGs [78]. Thus it makes sense to consider all the DAGs in the equivalence class to be equally possible [31]. As the number of DAGs in the equivalence class can be huge, a practical way to get a small number of representative structures is to sample DAGs from the equivalence class uniformly at random. If this does not give us enough certainty about the causal relationships for the task at hand, we need to supplement the observational data with *interventional data*, that is, data where we have actively intervened on the values of some of the variables. Sampling DAGs from the equivalence class is also useful in the problem of minimizing the number of interventional experiments required to single out the correct DAG [26]. Motivated by these applications, we study this sampling problem along with the related counting problem:

Problem C. Count and sample DAGs that are Markov equivalent to a given DAG.

Problems A, B, and C are all problems where we count or sample objects that satisfy a set of constraints. For problems of this type, a wide variety of algorithmic tools have been developed. Selecting the best tools for the task at hand is aided by the classification provided by computational complexity theory. The complexity class of #P-complete problems [84] consists of counting problems that are unlikely to be solvable exactly in polynomial time, analogously to the class of NP-complete decision problems. Counting problems are often #P-complete even if the corresponding decision problem can be solved in polynomial time. For example, the problem of counting linear extensions (Problem A) is #P-complete [8], but a linear extension can be found in linear time by topological sorting. For many #P-complete problems, efficient approximation is still possible. The computational difficulties of randomized approximation and almost uniform sampling are closely related [46]; many #P-complete problems have randomized approximation schemes that are obtained by reducing the problem into the corresponding sampling problem [8, 21, 45], which is then solved by simulating rapidly mixing Markov chains [21, 76].

The asymptotic polynomiality of the time complexity does not directly translate into practical feasibility. Polynomial-time algorithms may be unfeasible in practice due to large constant factor or high degree in the polynomial; for example, this is the case in the existing Markov chain-based polynomial randomized approximation schemes for counting linear extensions [8, 9]. On the other hand, exact algorithms with exponential worst-case time complexity may still achieve good performance in practice by



$\Pr(H = \mathrm{T})$
0.748

h	$\Pr(B = \mathrm{T} \mid H = h)$
T	0.79144
F	0.58730

b	$\Pr(M = \mathrm{T} \mid B = b)$
T	0.97297
F	0.69231

b	$\Pr(S = \mathrm{T} \mid M = m)$
T	0.2
F	0.05

s	b	$\Pr(C = \mathrm{T} \mid S = s, B = b)$
T	T	0.8
T	F	0.8
F	T	0.8
F	F	0.05

$$\begin{aligned}\Pr(M = m, S = s, B = b, C = c, H = h) = \\ \Pr(H = h) \cdot \Pr(B = b \mid H = h) \cdot \Pr(M = m \mid B = b) \\ \cdot \Pr(S = s \mid M = m) \cdot \Pr(C = c \mid S = s, B = b)\end{aligned}$$

Figure 1.2: A Bayesian network that corresponds to the same distribution as the network in Figure 1.1. The DAG structures are Markov equivalent, which means that for any assignment of parameters for the other DAG, we can always find values for the parameters for this DAG such that the distributions are equal. In the shown parameterization, the probability values are rounded to five digits after the decimal point.

exploiting the special properties of the instances the algorithm is applied to. While these special properties can sometimes be included as additional parameters in the time complexity [16, 20], proving bounds that fully capture the instance-specific time complexity is often difficult. Thus it makes sense to implement the algorithms and measure their actual running times on practical instances.

Counting satisfying assignments of a Boolean formula given in the conjunctive normal form is a prime example of a #P-complete problem with practical algorithms whose time complexity bounds are very loose. The corresponding decision problem is NP-complete, but algorithms for it can often solve real-world instances with millions of variables, and improved solvers are empirically compared to each other in yearly competitions [43]. These solvers can be modified to allow exact counting of satisfying assignments [67, 82]. Furthermore, various approximate counting and sampling algorithms [12, 30, 54, 77] have been proposed based on the idea of repeatedly applying a solver for the decision problem in instances where the space

of satisfying assignments is bisected using a randomly chosen hash function over the assignments [81, 85]. These methods can be used to solve any constrained counting or sampling problem in a declarative fashion, by encoding the constraints as a Boolean formula. This approach significantly reduces the amount of effort that has to be spent on each problem, and allows constraints to be easily added and removed without requiring a redesign of the algorithm. For instance, Problems A and C can be obtained from the uniform case of Problem B by adding further constraints, because linear extensions can be encoded as maximal DAGs that contain a given DAG as a subgraph, and the set of DAGs in a Markov equivalence class can be characterized by required and prohibited subgraphs.

Even though algorithms for general problems such as Boolean satisfiability are good at exploiting instance-specific structure, they are unlikely to understand the special properties of the problem, which sometimes allow very efficient solutions. For example, the problem of counting acyclic orientations in chordal graphs, from which Problem C is obtained by adding additional constraints, admits a polynomial-time algorithm due to its somewhat surprising connection to the chromatic polynomial [68, 79]. For this reason, even small differences in the problem statement might necessitate completely different solutions. In general, the existing algorithmic toolbox does not directly give us a recipe for solving any problem; rather, the best algorithms are obtained through combining existing methods, mathematical observations, and experimentation.

1.1 Contributions and organization

This thesis presents advancements towards solving the problems A, B, and C, introduced in the previous section, all of which facilitate practical graphical model structure learning, as well as advance the algorithmic theory of acyclic structures. The contributions are in the form of five conference papers.

In Papers I–III, we consider Problem A particularly from the viewpoint of randomized approximation schemes and Markov chains. The previous methods [8, 9, 21, 48] have been developed with focus on theoretical results such as asymptotic time complexity. In this thesis, motivated by the application of structure learning, we take a more practical approach, placing emphasis on the actual running time of the algorithms on modern hardware. In Paper I, we develop a method for measuring the practical performance of a Markov chain, as it is often better than the available theoretical bound, and apply it to various chains. In Paper II, we improve the per-

formance of Markov chain-based methods by improving the reduction [8] from the counting problem into the problem of sampling linear extensions uniformly at random. In addition, we present an approximation version of an exact linear extension counting algorithm [47], which outperforms the polynomial-time approximation schemes for moderate-sized instances. Finally, in Paper III, we present an improved randomized approximation scheme in which we express the problem in a continuous space and use a Markov chain for sampling in the space.

In Paper IV, we address Problem B directly without reducing it to Problem A first. We present an algorithm with a time complexity bound that is exponential in the number of nodes. In addition, we devise a polynomial-time algorithm for the special case of symmetric modular distributions, which also improves the best known bound [55] for the case where the distribution is uniform. We take particular care in making the algorithm exact by considering the often overlooked effect of the time complexity of sufficient-precision numerical operations.

Problem C can be reduced to the simpler problem of counting and sampling *moral acyclic orientations* in chordal graphs, which we consider in Paper V. We augment the state-of-the-art method [35] with dynamic programming, achieving exponential running time bound along with practical speedup. We also devise another dynamic programming algorithm that works in the clique tree decomposition structure of the chordal graph. This algorithm runs in linear time if the maximum size of a clique in the graph is bounded by a constant, and thus it is the fastest method in sparse instances.

The remainder of the thesis is structured as follows: In Chapter 2, we introduce the basic theory and problems in Bayesian network structure learning that lead up to our computational problems. After establishing the motivation, we present the algorithmic contributions to Problems A, B, and C in their pure combinatorial form, without the language of Bayesian networks, in Chapters 3, 4, and 5, respectively. We summarize and discuss the results in Chapter 6. The original publications are reprinted in the end of the thesis.

1.2 Author contributions

All the publications were jointly written by all of their authors. The other main contributions by the present author are as follows:

Paper I: The present author devised the mixing time estimation scheme jointly with Teppo Niinimäki and implemented all the algorithms.

Paper II: The present author developed the structure decomposition algorithm, proved its time complexity and implemented it.

Paper III: The present author came up with the perfect sampler, proved its correctness and implemented the algorithm.

Paper IV: The present author developed the performance optimizations of the symmetric case adaptation of the general algorithm and carried out the numerical precision analysis.

Paper V: The present author devised the algorithm based on clique trees and implemented all the algorithms.

Chapter 2

Bayesian networks

We consider the problem of modeling the joint distribution of random variables $(X_v)_{v \in V}$, where V is an index set of size n. Our main interest is in the case where each variable X_v takes values from a finite set Ω_v. The simplest way to model this joint distribution would be the full probability table model, parameterized by $\theta : \prod_{v \in V} \Omega_v \to [0, 1]$ such that $\theta((x_v)_{v \in V})$ specifies the probability that $X_v = x_v$ for all $v \in V$. As the probabilities should add up to 1, one of the parameters is redundant, and thus the model has $\prod_{v \in V} |\Omega_v| - 1$ free parameters. Figure 1.1b shows an example of a full probability table model.

As in practice we typically have tens or hundreds of variables, the number of parameters in full-table models is prohibitively large. To reduce the number of parameters, we use our prior knowledge that practical joint distributions are likely to contain *conditional independencies* between the variables, that is, for some $a, b \in V$ and $U \subseteq V \setminus \{a, b\}$, if we know the values of X_v for all $v \in U$, then knowing the value of X_b provides no additional information on the distribution of X_a. This allows us to reduce the number of parameters in the model. We focus on one particular type of model, Bayesian networks, which uses DAGs on V to encode the independencies.

To reason about the structure of a DAG G and independencies, we will need the following notation:

- For all $v \in V$, G_v is the set of parents of v in G.
- For all $v \in V$, $\mathrm{Desc}(G, v)$ is the set of descendants of v in G, where node $x \in V$ is a descendant of v if there is a directed path from v to x in G. We also consider any node to be a descendant of itself.
- If for some symbol z we have defined z_v for all $v \in U \subseteq V$, we use the notation z_U as a shorthand for $(z_v)_{v \in U}$. For example, $X_U = x_U$ means that $X_v = x_v$ for all $v \in U$.

- We denote the conditional independence of random variables A and B given C as $A \perp\!\!\!\perp B \mid C$. We can replace A, B, or C with multiple random variables by considering tuples of multiple variables as a single random variable. For example, if $a \in V$ and $U, U' \subseteq V$, the notation $X_a \perp\!\!\!\perp X_{U'} \mid X_U$ means that X_a is independent of all the variables X_v where $v \in U'$ given the values of X_v for all $v \in U$.
- We choose to index the random variables X_v by a separate index set of DAG nodes $v \in V$ instead of using them directly as nodes. This will improve readability in the subsequent sections, as there we consider the structure of the DAG to also be a random variable. We make an exception in the examples (such as Figures 1.1 and 1.2), where for brevity we use the same notation for the DAG node and the random variable.

The independencies implied by the Bayesian network DAG structure G all follow from the *local Markov property*, which states that for all $v \in V$

$$X_v \perp\!\!\!\perp X_{V \setminus \mathrm{Desc}(G,v)} \mid X_{G_v},$$

in other words, X_v is conditionally independent of all of its non-descendant variables in G given the values of its parent variables. Many other independencies follow by combining these independencies. All the conditional independencies implied by DAG G can be characterized by the *d-separation* criterion [18].

For example, consider the DAG in Figure 1.2. The local Markov property states that $C \perp\!\!\!\perp (M, S, B, H) \mid (S, B)$, from which it follows that $C \perp\!\!\!\perp H \mid (S, B)$. In fact also a stronger statement $C \perp\!\!\!\perp H \mid B$ holds, but it does not follow directly from the local Markov property for this DAG. Instead, we have to use the d-separation criterion or manually prove it by applying the properties of independence known as *graphoid axioms* [18, 73] to the conditional independencies that follow from the local Markov property.

To use the local Markov property to factorize the joint probability distribution of X_V, we use the fact that because G is acyclic, it has a topological ordering $(v_i)_{i=1}^n$, that is, an ordering of V such that if there is an edge from v_i to v_j, then $i < j$. By the chain rule of probabilities, we can factorize the joint probability as follows:

$$\Pr(X_V = x_V) = \prod_{i=1}^{n} \Pr(X_{v_i} = x_{v_i} \mid X_{v_{1:i-1}} = x_{v_{1:i-1}}),$$

where $v_{i:j}$ is the set of v_k such that $i \le k \le j$. Since $(v_i)_{i=1}^n$ is a topological ordering, $G_{v_i} \subseteq v_{1:i-1} \subseteq V \setminus \mathrm{Desc}(G, v_i)$ for all $1 \le i \le n$. Thus we may

use the local Markov property to reduce the conditioning set $v_{1:i-1}$ to G_{v_i}, yielding the factorization

$$\Pr(X_V = x_V) = \prod_{v \in V} \Pr(X_v = x_v \mid X_{G_v} = x_{G_v}). \tag{2.1}$$

Now, if the DAG structure G is known, we can parameterize the joint distribution X_V by specifying $\Pr(X_v = x_v \mid X_{G_v} = x_{G_v})$ for all $v \in V$ and $x_{G_v \cup \{v\}} \in \prod_{u \in G_v \cup \{v\}} \Omega_u$ as a parameter. We denote this parameter by $\theta_v(x_{G_v \cup \{v\}})$. Thus the complete Bayesian network model is the pair (G, θ) of the DAG structure G and the parameterization $\theta = \theta_V$, where $\theta_v : \prod_{u \in G_v \cup \{v\}} \Omega_u \to [0, 1]$ is the conditional probability table for v. The number of free parameters in θ when the structure G is fixed is

$$\sum_{v \in V} (|\Omega_v| - 1) \prod_{u \in G_v} |\Omega_u|,$$

where we subtract 1 from $|\Omega_v|$ due to the fact that for each v and x_{G_v} the sum of $\theta_v(x_{G_v \cup \{v\}})$ over $x_v \in \Omega_v$ is 1, and thus one of the parameters in the sum can be determined from the others. As we can see, the number of parameters is small if the graph is sparse in the sense that the nodes have few parents. If we choose G to be a complete directed graph, then the local Markov property implies no independencies, and the number of parameters will actually match the number of parameters $\prod_{v \in V} |\Omega_v| - 1$ of the probability table model. This means that the Bayesian network model can be viewed as a generalization of the full probability table model.

2.1 Score-based learning

We consider the problem of Bayesian network learning: we are given training data points $x_V^{(j)}$ for $j = 1, 2, \ldots, N$ that are independent and identically distributed samples from an unknown *generating distribution*, and we should find DAG G on V and parameters θ such that the Bayesian network (G, θ) entails a distribution as close as possible to the generating distribution. The most difficult part of this learning problem is learning the structure, because after the structure G is fixed, the probability tables that make up θ are typically obtained simply by collecting statistics from the data. Our focus is in the score-based approach for this learning problem: each possible network structure G is assigned a score, which quantifies how well the data fits the structure, and typically also penalizes the complexity in the model. The learning method then either finds the structure maximizing the score, or outputs a mixture of multiple models weighted by their scores.

One natural way to define a score for a structure is through probabilities and Bayesian inference. In this approach, we consider the Bayesian network model to be a random variable, and its distribution quantifies for each possible model our current belief on how likely it is the correct model. For clarity, we use the notation $(\mathcal{G}, \Theta)$ to refer to this random uncertain network and (G, θ) for a fixed network. As the parameters of a Bayesian network consists of probabilities, Θ is a continuous random variable, while the graph structure $\mathcal{G}$ is discrete. Because the dimensions of the probability table Θ_v depend on the parent set $\mathcal{G}_v$, we can only write its probability distribution conditionally when the parent set $\mathcal{G}_v$ is fixed.

The marginal density of $(\mathcal{G}, \Theta)$ at (G, θ), denoted by $p_{\mathcal{G},\Theta}(G, \theta)$, describes our prior belief of (G, θ) being the correct model. The prior is typically specified as the product of the *structure prior* $\Pr(\mathcal{G} = G)$ and the *parameter prior* $p_{\Theta|\mathcal{G}=G}(\theta)$. The prior should be set using domain knowledge; if nothing is known in advance, the structure prior could be set to be uniform over all DAGs, or if we expect the model to be simple, it should assign higher probabilities to DAGs with fewer edges. After this, we observe the training data $D = (x_V^{(j)})_{j=1}^N$, which we think of as a realization of a random variable $\mathcal{D}$. Based on the training data, we update our belief on the likelihood of each possible model, obtaining the *posterior* distribution, that is, the distribution of $(\mathcal{G}, \Theta)$ conditioned on the data observation $\mathcal{D} = D$. To compute the posterior, we use the Bayes formula

$$p_{\mathcal{G},\Theta|\mathcal{D}=D}(G, \theta) = \frac{\Pr(\mathcal{D} = D \mid \mathcal{G} = G, \Theta = \theta) p_{\mathcal{G},\Theta}(G, \theta)}{\Pr(\mathcal{D} = D)}.$$

The divisor $\Pr(\mathcal{D} = D)$ depends only on the data we have observed, not the model (G, θ), and thus we consider it to be a constant and omit it from the score. Hence the score $s(G, \theta)$ for the Bayesian network model (G, θ) is given by $l(G, \theta) p_{\mathcal{G},\Theta}(G, \theta)$, where $l(G, \theta)$ is the *likelihood* of the data $\Pr(\mathcal{D} = D \mid \mathcal{G} = G, \Theta = \theta)$. The likelihood is simply the product of the probabilities assigned by the model (G, θ) to each of the independent data points $x_V^{(j)}$, which means that by applying (2.1) we get the formula

$$l(G, \theta) = \prod_{j=1}^{N} \prod_{v \in V} \Pr(X_v = x_v^{(j)} \mid X_{G_v} = x_{G_v}^{(j)}) = \prod_{j=1}^{N} \prod_{v \in V} \theta_v(x_{G_v \cup \{v\}}^{(j)}). \quad (2.2)$$

To score only the graph structure G without fixing the parameters θ, we integrate out θ, essentially averaging over all the possible assignments for the parameters:

$$s(G) = \int s(G, \theta) \, \mathrm{d}\theta = \int l(G, \theta) p_{\mathcal{G},\Theta}(G, \theta) \, \mathrm{d}\theta.$$

By splitting the network prior $p_{\mathcal{G},\Theta}(G,\theta)$ into the structure prior $\Pr(\mathcal{G}=G)$ and the parameter prior $p_{\Theta|\mathcal{G}=G}(\theta)$, and moving the structure prior outside the integral, we can write the score as

$$s(G) = \Pr(\mathcal{G}=G) \int l(G,\theta) p_{\Theta|\mathcal{G}=G}(\theta)\,\mathrm{d}\theta. \tag{2.3}$$

This formula only gives a general framework for defining the score in a Bayesian way. For practical feasibility, we need to set the form of the parameter prior $p_{\Theta|\mathcal{G}=G}$ such that the integral in (2.3) can be computed. In addition, the score should have some internal structure that enables solving structure learning problems, such as optimizing the score or sampling weighted by the score, faster than considering every one of the $2^{O(n^2)}$ possible DAGs separately. Next, we will introduce a class of score functions that has a suitable internal structure: modular functions.

2.1.1 Modularity

We say that a function f which maps each DAG G on V to a nonnegative number $f(G)$ is *modular* if it can be factored into functions f_V such that

$$f(G) = \prod_{v\in V} f_v(G_v),$$

or in other words, the value of f is obtained as a product of the values of the per-node functions $f_v : 2^{V\setminus\{v\}} \to \mathbb{R}$ when each of them is given the parent set of v in the DAG as the argument. A distribution over DAG structures is modular if its probability mass function is modular. Our aim is to ensure that the structure score function $s(G)$ is modular, because it makes it possible to speed up structure learning, such as optimizing the score or sampling DAGs from the distribution defined by the score. We will next describe the additional assumptions we need about the priors to ensure the modularity of the structure score $s(G)$; the structure learning algorithms for the case of modular score are described in Section 2.1.2.

The data likelihood $l(G,\theta) = \Pr(\mathcal{D}=D \mid \mathcal{G}=G, \Theta=\theta)$ already has a suitable decomposition into local factors, because from (2.2) we see that $l(G,\theta) = \prod_{v\in V} l_v(G_v,\theta_v)$, where

$$l_v(P,\theta_v) = \prod_{j=1}^{N} \theta_v(x^{(j)}_{P\cup\{v\}}).$$

We will also require that the structure prior $\Pr(\mathcal{G}=G)$ and the parameter prior $p_{\Theta|\mathcal{G}=G}(\theta)$ have factorizations $\prod_{v\in V} a_v(G_v)$ and $\prod_{v\in V} b_v(G_v;\theta_v)$, respectively. Note the second argument of b_v is θ_v, because the rest of θ has

no meaning unless the whole DAG structure G is fixed. By substituting the modular factorizations to the formula (2.3) for the score, we get that

$$s(G) = \left(\prod_{v \in V} a_v(G_v) \right) \int \prod_{v \in V} l_v(G_v, \theta_v) b_v(G_v; \theta_v) \, \mathrm{d}\theta.$$

Each factor in the product inside the integral depends only on one component θ_v of the integration variable θ, and thus by splitting the integral into componentwise integrals and exchanging the order of the product and the integrals, we get a modular factorization $s(G) = \prod_{v \in V} s_v(G_v)$, where

$$s_v(P) = a_v(P) \int l_v(P, \theta_v) b_v(P; \theta_v) \, \mathrm{d}\theta_v.$$

To enable the practical computation of the *parent set scores* $s_v(P)$ for all $v \in V$ and $P \subseteq V \setminus \{v\}$, the parameter prior component $b_v(P; \theta_v)$ is typically chosen such that the integral can be computed analytically. Various popular scores, such as K2 [14] and the BDe family of scores [10,36], achieve this by using a Dirichlet distribution as the parameter prior, because then the posterior will also be a Dirichlet distribution, which is known to be integrable [18].

Remark. For simplicity of presentation, we only considered cases where each variable X_v takes values from a finite discrete set. However, the theory can be generalized to continuous variables as well. This is typically done by assuming that the continuous data is sampled from a multivariate Gaussian distribution. By choosing the parameter priors appropriately, one can ensure that the resulting structure score is modular. One example of such a score for the Gaussian case is the BGe score [25, 57].

2.1.2 Optimization and sampling

From the strictly Bayesian point of view, the assignment of structure scores $s(G)$ to all the possible DAGs G already answers the Bayesian network structure learning problem, as the scores constitute our updated belief on the correct DAG structure based on the data. However, in practical applications, we should be able to point out a single structure or a small number of structures as the most likely ones.

The most likely structure is naturally defined as the DAG G that maximizes the score $s(G)$. In the case of the Bayesian score, it is known as the *maximum a posteriori* (MAP) structure, as it maximizes the posterior probability. The problem of finding the MAP structure has been proved

to be NP-complete for typical scores [13]. The fastest exact modular score optimization algorithms in terms of asymptotic time complexity run in $O(2^n n^2)$ time [51,70], assuming that each parent set score is given in $O(1)$ time by a black box in the sense that we cannot make any assumptions about the values of the scores. This is very close to the theoretical lower bound $\Omega(2^n n)$ that follows simply from the fact that there are $2^{n-1}n$ parent set scores $s_v(P)$, and the algorithm has to consider all of them to find the exact optimum DAG. If exactness is not necessary, we may use a local search method [71] and avoid computing all the parent set scores. In order to solve larger instances exactly, we can impose an upper limit d on the parent set size, setting the structure prior $a_v(P)$ and thus also the score $s_v(P)$ to zero if $|P| > d$. The scores may be pruned further by finding cases where $s_v(A) \geq s_v(B)$ for some $A \subset B \subseteq V \setminus \{n\}$ and in each of them setting $s_v(B) = 0$, because parent set B can always be replaced by A without violating the acyclicity constraint. The resulting sparse case of the optimization problem where most parent set scores are zero can be solved faster than the general case by constraint optimization and heuristic methods [15,87,92].

If we have only limited amount of training data available, the MAP structure will typically represent only a very small portion of the posterior probability mass [23]. In these cases, it makes sense to obtain representative DAGs from the posterior distribution by sampling structures weighted by the structure score. This problem is more difficult than optimization in the sense that we cannot use techniques such as pruning or branch-and-bound, because even the smaller scores affect the distribution. Paper IV presents the first exact modular DAG sampling algorithm, which is described in more detail in Chapter 4. The previous research into the problem has focused on approximate methods based on running Markov chains in the space of DAGs [29,56,61] or related spaces [69]. Even though these methods are useful in practice, their accuracy guarantees are weak: we know that if we run the Markov chain for long enough, we get close to the correct distribution, but we do not have any useful bounds on the sufficient number of steps.

The computational difficulty of sampling from modular distributions has led to the use of *order-modular* distributions, which are distributions that can be obtained from a modular distribution by multiplying the probability weight of each DAG by the number of topological orderings of the DAG. Using an order-modular distribution instead of a modular distribution allows for faster sampling both exactly [33,69] and approximately [23]. This speedup comes at a cost of biasing the distribution: we effectively

choose a structure prior which favors DAGs that have more topological orderings, which is usually not justified by anything else than computational convenience. To fix this bias, we can use *importance sampling* [33,69]: each sampled DAG from the order-modular distribution is given a weight which is the inverse of the number of topological orderings of that DAG. In practice, importance sampling can be used similarly to unweighted sampling by taking the weights into account. To implement this method, we need a subroutine that can count the number of topological orderings of a given DAG, which is equivalent to counting the linear extensions of a given partial order. Our contributions to this problem are described in Chapter 3.

2.2 Markov equivalence

When learning Bayesian networks from data, it often happens that even if we have large amounts of training data available, we find two network structures G and G' that seem to fit the data equally well. This is typically caused by the DAG G being *Markov equivalent* to the DAG G', which means that they can express exactly the same set of distributions. More exactly, this means that there is a one-to-one correspondence between parameter assignments θ of G and θ' of G' such that (G, θ) and (G', θ') always correspond to the same distribution. The structures in Figures 1.1 and 1.2 are an example of two Markov-equivalent DAGs. Another equivalent definition states that DAGs G and G' are equivalent if they yield exactly the same set of conditional independencies by the local Markov property. This means that a constraint-based Bayesian network learning method cannot distinguish the two DAGs, because both structures fit the conditional independencies found from the data equally well. Also in score-based learning, the score function is often set such that Markov-equivalent DAGs give the same score; this is the case for example for the BDe [36] score. For the problem of modeling the joint distribution, the existence of multiple indistinguishable DAG structures is not really a problem: we can simply select an arbitrary DAG from the equivalence class. However, Bayesian networks are also used for *causal modeling*, and from that viewpoint Markov equivalent DAGs are not interchangeable.

Causal Bayesian networks are a type of *structural causal models* [72] which encode the mechanism by which the random variables influence each other. The parent variables X_{G_v} of a random variable X_v are its *direct causes*, and parameters θ_v specify the mechanism by which the value of X_v is determined from them. The structure learning methods described earlier are not sufficient for learning the DAG structure G of the causal

Bayesian network (*causal learning*): given enough training data, they will find a DAG G' that is Markov equivalent to G with high probability, but G' may still be very different from G in the causal sense. For example in Figures 1.1 and 1.2, the DAGs disagree on whether B (brain tumor) causes H (headaches) or vice versa. It is possible that we are lucky in the sense that the learned structure G' is the only DAG in its Markov equivalence class: in this case, we have reason to believe that it is also the correct causal structure. However, this is often not the case: even when $V = \{a, b\}$ and the random variables X_a and X_b are dependent, we have two possible Markov equivalent DAG structures $a \to b$ and $a \leftarrow b$. Asymptotically, only about every 14th DAG has no Markov equivalent DAGs [80].

Even if we have more than one DAG in the learned Markov equivalence class, some features of the DAG are already known: the *skeleton*, that is, the undirected graph obtained by removing all directions from the edges, and the *immoralities* (also known as *v-structures*), that are the node triplets (a, v, b) such that there are edges $a \to v \leftarrow b$ but no edge between a and b. Actually, we can check if two DAGs are Markov equivalent by seeing if they have the same skeleton and the same set of immoralities [88]. We can summarize the Markov equivalence class by starting from the skeleton, and then for each immorality (a, v, b) directing edges $a \to v$ and $b \to v$, and finally directing the edges whose directions can be inferred from the acyclicity condition or the fact that there should not be any additional immoralities. The resulting partially directed graph, known as the *essential graph*, uniquely characterizes the Markov equivalence class and can be constructed in polynomial time [63, 88]. The essential graph of the DAGs in Figures 1.1 and 1.2 is shown in Figure 2.1. In this case we may deduce that because there is a directed edge from B to C in the essential graph, B (brain tumor) is a direct cause of C (coma). This deduction relies on the assumption that the model contains all the relevant variables, because if there is a variable Z that is a direct cause of both B and C, then we may have learned the edge from B to C only because we omitted the *confounding* variable Z from the model [72, Chapter 6].

If the essential graph does not reveal enough information about the causal structure, for instance if in the example of Figure 2.1 we want to know whether S (increase in serum calcium) is a direct cause of M (metastatic cancer) or vice versa, then observational training data from the distribution is not enough [78]: we need to augment our training data with *interventional experiments*, in which we control some variables and see how this affects other variables. In real-world applications, gathering interventional data is often very expensive, as it requires interfering with

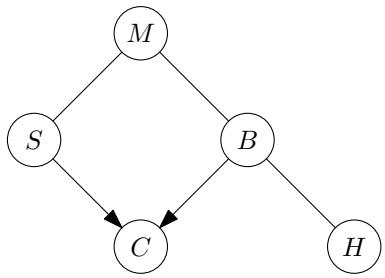


Figure 2.1: The essential graph of the Markov equivalence class that contains the DAGs in Figures 1.1 and 1.2. The DAGs in the equivalence class are exactly the graphs that can be constructed from the essential graph by orienting each undirected edge such that the resulting graph is acyclic and no additional immoralities are created.

the mechanism being modeled. This gives rise to the problem of designing the interventional experiments such that the total cost of the experiments is minimized. Many algorithms have been proposed for different variants of this problem [22,26,32,42,75]. One of these algorithms, the random greedy interventional design algorithm [26], considers the problem of finding a set of experiments that are approximately optimal in maximizing the expected number of discovered edge directions while staying within given cost budget. This algorithm is one application for our problem of sampling DAGs from a Markov equivalence class, because the main performance bottleneck of the algorithm is estimating the average utility of each possible intervention, which works by sampling DAGs from the equivalence class uniformly at random and averaging the utilities of each sampled DAG.

Sometimes it is impossible to carry out interventional experiments, as the phenomenon being modeled is out of our control. If we have no prior knowledge which could be used to direct some of the remaining undirected edges, we have no other choice than to consider each causal structure in the learned Markov equivalence class as equally likely [31]. To estimate the total causal effects between variables (including indirect effects through intermediate variables), we should use the calculus of interventions [72] to estimate the effect in each DAG of the equivalence class, and aggregate all the results. This is computationally infeasible if the size of the equivalence class is too large. If this is the case, we can reduce the number of considered DAGs by sampling them uniformly at random from the equivalence class. Actually, sampling DAGs from the Markov equivalence class of a maximum a posteriori DAG is approximately equivalent to sampling DAGs from the posterior distribution if the chosen score is equal for Markov equivalent

DAGs and the amount of observational training data is very large. This is the case because as the amount of data increases, the total probability of the other DAG structures in the posterior distribution will approach zero. Despite this equivalence, it often is computationally more feasible to find a DAG that optimizes the score, resorting to heuristic methods if necessary, than to sample from the posterior distribution directly. This is another application for uniform sampling of DAGs from a Markov equivalence class, which will be considered in more detail in Chapter 5 and Paper V.

Chapter 3

Linear extensions

In addition to its application in Bayesian network structure learning that was discussed in Section 2.1.2, the problem of counting linear extensions appears as a subproblem in various other problems in artificial intelligence, such as sequence analysis [62], preference reasoning [60], partial order plans [66], and inference of election outcomes [50]. The problem has been proved to be a #P-complete [8], which means that it is unlikely to be solvable in polynomial time. In the sense of worst-case asymptotic time complexity, the fastest exact counting algorithms run in $O(2^n n)$ time [19], where n is the number of elements in the partial order. However, practical optimizations allow the algorithms to run much faster than the time complexity bound on typical instances [47].

Despite the #P-completeness of the problem, one can achieve polynomial time complexity bounds by allowing *randomized approximation* [21], where we only require the output to be within given relative error of the correct value with given probability. This can be done by reducing the problem into a polynomial number of instances of the problem of sampling linear extensions uniformly at random [8], which is then solved in polynomial time using Markov chains [3, 9, 38, 48]. Prior to our results, the approximation schemes were outperformed by the exact counting method in practice [47].

In this chapter, we introduce our contributions to the problems of counting and sampling linear extensions. After defining the essential concepts, in Sections 3.2 and 3.3 we introduce the two practical approximate linear extension counting methods of Paper II. Then in Section 3.4 we switch our focus to sampling linear extensions using Markov chains and present the method of Paper I for empirically evaluating the mixing properties of a chain. We conclude the chapter with the contribution of Paper III: an exact sampler for a continuous-space generalization of linear extensions, which enables faster approximate linear extension counting in large instances.

3.1 Partial orders

Let A be a finite set and $\preceq$ a binary relation on A, that is, a subset of $A \times A$. We use the notation $x \preceq y$ to indicate that $(x, y) \in \preceq$. We define that $\preceq$ is a *partial order* on A if the following properties hold for all $x, y, z \in A$:

- Reflexivity: $x \preceq x$.
- Antisymmetry: if $x \preceq y$ and $y \preceq x$, then $x = y$.
- Transitivity: if $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

We denote $x \not\preceq y$ to indicate that $(x, y) \notin \preceq$ and $x \prec y$ to indicate that $x \preceq y$ and $x \neq y$. Each member (x, y) of the partial order relation is called an *(ordering) constraint*. Elements $x, y \in A$ are *comparable* in the partial order if $x \preceq y$ or $y \preceq x$; otherwise they are *incomparable*. Element $y \in A$ is a *predecessor* of $x \in A$ if $y \prec x$ and *successor* of x if $x \prec y$. An element is *minimal* or *maximal* if it has no predecessors or successors, respectively. The pair $(A, \preceq)$ is known as a *partially ordered set* or *poset* for short—all the concepts defined for the partial order $\preceq$ naturally extend to the poset P. Given a set $B \subseteq A$, the *restriction* of P into B is the poset $(B, \preceq')$ where $\preceq' = \preceq \cap (B \times B)$, and it is denoted by $P[B]$.

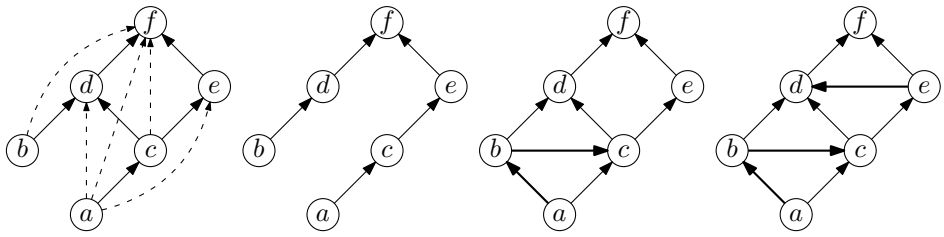
Partial order $\preceq'$ on A is an *extension* of the partial order $\preceq$ if it is a superset of $\preceq$, or in other words, if for all $x, y \in A$ such that $x \preceq y$ it holds that $x \preceq' y$. Conversely, $\preceq'$ is a *relaxation* of $\preceq$ if $\preceq$ is an extension of $\preceq'$. A partial order in which all pairs of elements are comparable is *linear*. Each *linear extension* $\preceq'$ of $\preceq$ can equivalently be represented as an ordered list $\mathrm{List}(A, \preceq') = (x_1, x_2, \ldots, x_n)$ of the set of elements A which is compatible with the partial order in the sense that $x_j \not\preceq x_i$ for all $1 \leq i < j \leq n$. Figure 3.1 shows examples of these concepts.

We denote the set of its linear extensions of a poset P by $\mathcal{L}(P)$ and the number of linear extensions $|\mathcal{L}(P)|$ by $\ell(P)$. In the problem of randomized approximate counting of linear extensions of given poset P with given quality parameters $\epsilon, \delta > 0$, the output $\ell'(P)$ of the randomized algorithm should be an (ϵ, δ)-*approximation* of $\ell(P)$, which means that

$$\Pr((1+\epsilon)^{-1}\ell(P) \leq \ell'(P) \leq (1+\epsilon)\ell(P)) \geq 1 - \delta.$$

For example, $(1, 1/4)$-approximation means that the result $\ell'(P)$ is in range $[\ell(P)/2, 2\ell(P)]$ with probability at least $3/4$.

The problem of counting linear extensions of a poset is equivalent to the problem of counting the topological orderings of a given DAG: the topological orderings of a DAG $G = (V, E)$ correspond to the linear extensions of the poset $(V, \preceq)$ where $x \preceq y$ if there is a directed path from x to y in G.



(a) poset (b) relaxation (c) extension (d) linear extension

Figure 3.1: We visualize partial orders $\preceq$ as DAGs where each edge $x \to y$ indicates the ordering constraint $x \prec y$ in the partial order. We often omit the edges that follow by transitivity. In the partial order of (a), these edges are shown as dashed lines. (b) is a relaxation of (a), as it is obtained by removing the constraints $c \preceq d$ and $a \preceq d$ from the partial order relation. The extension (c) is obtained from (a) by adding the constraints $a \preceq b$ and $b \preceq c$; we also need to add $b \preceq e$ to satisfy transitivity. After adding the constraint $e \preceq d$ to the relation, we get a linear extension (d) of the poset (a) corresponding to the ordered list (a, b, c, e, d, f). This is one of the 7 linear extensions of the poset (a); the other six are (a, b, c, d, e, f), (a, c, b, d, e, f), (a, c, b, e, d, f), (a, c, e, b, d, f), (b, a, c, d, e, f), and (b, a, c, e, d, f).

3.2 Adaptive relaxation Monte Carlo

The exact linear extension counting algorithm due to De Loof et al. [19] is based on the observation that each linear extension of a poset $P = (A, \preceq)$ is obtained by choosing a minimal element x of P as the first element, followed by an arbitrary linear extension of the remainder of the poset $P[A \setminus \{x\}]$. This gives a recurrence for the function $f(S) = \ell(P[S])$:

$$f(S) = \sum_{x \text{ minimal in } P[S]} f(S \setminus \{x\}), \tag{3.1}$$

where the base case is $f(\emptyset) = 1$. The number of linear extensions $\ell(P)$ can then be obtained as the value $f(A)$. As there are 2^n sets $S \subseteq A$, and we need to compute each value $f(S)$ only once, we get time complexity $O(2^n n)$.

It often happens that not all of the values $f(S)$ are needed to compute $f(A)$. If we implement the recurrence using recursion and memoize each result we compute to avoid recomputation, we only need to consider the sets $S \subseteq A$ that are *upsets* of the poset P, that is, sets in which for all $x \in S$ and $y \in A$ such that $x \preceq y$ it holds that $y \in S$. Kangas et al. [47] improved the practical running time of this algorithm further by observing

that it suffices to only consider upsets that are connected: if $S \subseteq A$ can be partitioned into two sets X and Y that are completely independent in the sense that x and y are incomparable for all $x \in X$ and $y \in Y$, then each linear extension of $P[S]$ is obtained by arbitrarily interleaving linear extensions of $P[X]$ and $P[Y]$. This gives us another recursive rule for this case:

$$f(S) = \binom{|S|}{|X|} f(X) f(Y). \tag{3.2}$$

The dynamic programming algorithm based on the rules (3.1) and (3.2) is able to exactly count the linear extensions of many practical poset instances with up to about 60 elements [47]. For the existing approximation schemes [8,9,38], instances larger than 60 elements were practically impossible. This gives rise to the question on whether we can obtain faster linear extension counting algorithms by making the dynamic programming algorithm approximate. In Paper II, we answer the question in the affirmative with the Adaptive Relaxation Monte Carlo (ARMC) method.

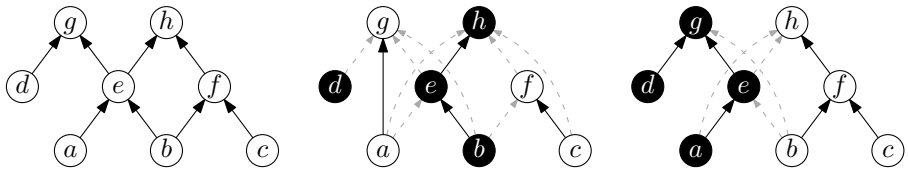
In the ARMC method, we find a relaxation $R = (A, \preceq')$ of the poset P with the aim that it would be easier for the exact dynamic programming algorithm to compute $\ell(R)$ than $\ell(P)$. Because $\mathcal{L}(R) \supseteq \mathcal{L}(P)$, the computed number of linear extensions $\ell(R)$ will be larger than the desired output $\ell(P)$. We correct the overestimation by multiplying $\ell(R)$ with the correction factor $\mu = \ell(P)/\ell(R)$, which we estimate using Monte Carlo sampling: we sample linear extensions from $\mathcal{L}(R)$ uniformly at random, and obtain the estimator as the proportion of the samples that are also in $\mathcal{L}(P)$.

Even though in the method we need to sample linear extensions of the relaxation poset R, it is actually is much easier for us than in the general case, because we already used the dynamic programming algorithm to count the linear extensions $\ell(R)$, and we chose R to be an easy instance for the algorithm. If we retain the values $f(S) = \ell(R[S])$ computed by the counting algorithm, then we can use them to sample linear extensions efficiently by tracing back the recursive steps made by the algorithm. This is done by a recursive routine that samples a linear extension for the restriction $R[S]$ given $S \subseteq A$. If the counting algorithm used the rule (3.1) to compute $f(S)$, then we can first sample the first element x of the linear extension from the set of minimal elements of $R[S]$ weighted by $f(S \setminus \{x\})$, and then recursively sample the linear extension for the remainder $R[S \setminus \{x\}]$. Otherwise, the algorithm found a partition of independent sets X and Y for S and used the rule (3.2) to compute $f(S)$: in this case, we sample the linear extension of $R[S]$ by recursively sampling linear extensions of $R[X]$ and $R[Y]$ and randomly interleaving them.

The more samples we draw from $\mathcal{L}(R)$, the more likely we are to get a close estimate for μ. To ensure that we get an (ϵ, δ)-approximation, we use the adaptive Monte Carlo scheme by Dagum et al. [17] in which we stop sampling when we have seen $O(\epsilon^{-2} \log \delta^{-1})$ positive samples, that is, samples that are in $\mathcal{L}(P)$. The expected number of samples from $\mathcal{L}(R)$ we need to draw to find a single member of $\mathcal{L}(P)$ is μ^{-1}, which means that the expected number of samples we draw in total is $O(\mu^{-1}\epsilon^{-2} \log \delta^{-1})$. Because of the linear dependence on $\mu^{-1} = \ell(R)/\ell(P)$, it is important that we do not increase the number of linear extensions too much when choosing the relaxation of R for P. For example, if we use $R = (A, \emptyset)$ in a case where P has $O(1)$ linear extensions, then $\mu^{-1} = \Omega(n!)$, which is far larger than the worst-case running time bound $O(2^n n)$ of the exact dynamic programming algorithm.

In ARMC, we choose the relaxation R to be a *partition relaxation*, which is based on a partition of the set of elements A into sets $(A_i)_{i=1}^m$ such that each set A_i has size k, where k is a tunable parameter (the last set A_m may be smaller). Then we obtain the relaxation R from P by making each set A_i independent of the rest of the elements $A \setminus A_i$. More exactly, we set $R = (A, \preceq')$, where $\preceq' = \preceq \cap \bigcup_{i=1}^m (A_i \times A_i)$. Now if k is small, computing $\ell(R)$ should be much faster than computing $\ell(P)$, because the exponential-time dynamic programming algorithm will consider each component A_i separately, and $|A_i| \leq k$. However, decreasing k also means that we remove more constraints from $\preceq$ to obtain $\preceq'$, which makes $\mu = \ell(P)/\ell(R)$ smaller and as a consequence, we need more samples to estimate it. Thus the parameter k controlling the tradeoff must be set carefully to optimize the total running time. In addition, the elements we pick to the sets A_i in the partition affects the quality of the partition. We should select it such that $\ell(R)$ is minimized, but to make it computationally feasible, we use the number of removed constraints $|\preceq \setminus \preceq'|$ as the heuristic we minimize. As even the heuristic optimization problem is NP-complete [24], we settle for a local optimum obtained by swapping elements between the sets in the partition such that on each such move, the heuristic value decreases. Figure 3.2 shows an example of this heuristic optimization.

To select the value of the parameter k adaptively, we start from a small value and increase it until the running times of the two phases of the algorithm are balanced. For each k, we find the relaxation, count its linear extensions using the dynamic programming algorithm, and finally estimate μ by sampling. When it becomes clear that the sampling phase will take significantly more time than the dynamic programming algorithm, then we give up and try a larger value of k instead. If we increment k in large



(a) poset (b) initial relaxation (c) optimal relaxation

Figure 3.2: To find the relaxation to use for the poset in (a) with parameter $k = 4$, we start from an arbitrarily chosen initial partition $A_1 = \{b, d, e, h\}$ and $A_2 = \{a, c, f, g\}$ of sets of size k. The relaxation is obtained by removing all the ordering constraints between different sets of the partition, which are shown in (b) as dashed arrows. Note that we have to add the constraint arrow $a \to g$ to the graphical representation as it no longer follows by transitivity. Initially, the heuristic value, given by the number of removed constraints, is 8. The greedy local search swaps g and h, improving the heuristic to 5, and then a and b, further improving the heuristic to 4. This leads to the case in (c), which cannot be improved further. By minimizing the heuristic value, we have successfully reduced the number of linear extensions of the relaxation from 1680 in (b) to 420 in (c).

enough steps, then the failed attempts will not introduce notable overhead because the time complexity of each attempt grows exponentially in k.

In Paper II, we implemented the ARMC algorithm and compared it to the LEcount implementation [47] of the exact dynamic programming algorithm. We tested the algorithms on both artificially generated posets and posets derived from Bayesian networks. Figure 3.3 summarizes the results of the experiments. We see that ARMC extends the practical range of linear extension counting to over 200 elements by allowing approximation.

3.3 Structure decomposition in counting

Let us consider linear extension counting algorithms that are based on Monte Carlo sampling operations of the following type: Given a poset $Q = (A, \preceq)$ and its extension $Q' = (A, \preceq')$, estimate $\mu = \ell(Q')/\ell(Q)$ as the proportion of linear extensions of Q' when sampling linear extensions of Q uniformly at random. Each operation like this allows us to find the number of linear extension of a poset if we know the number of linear extensions of a relaxation or extension of the poset. The extension or relaxation should be close in the sense that μ is not too small, because we need $\Theta(\mu^{-1})$ samples to get a good approximation of μ. In the previous section, we introduced the

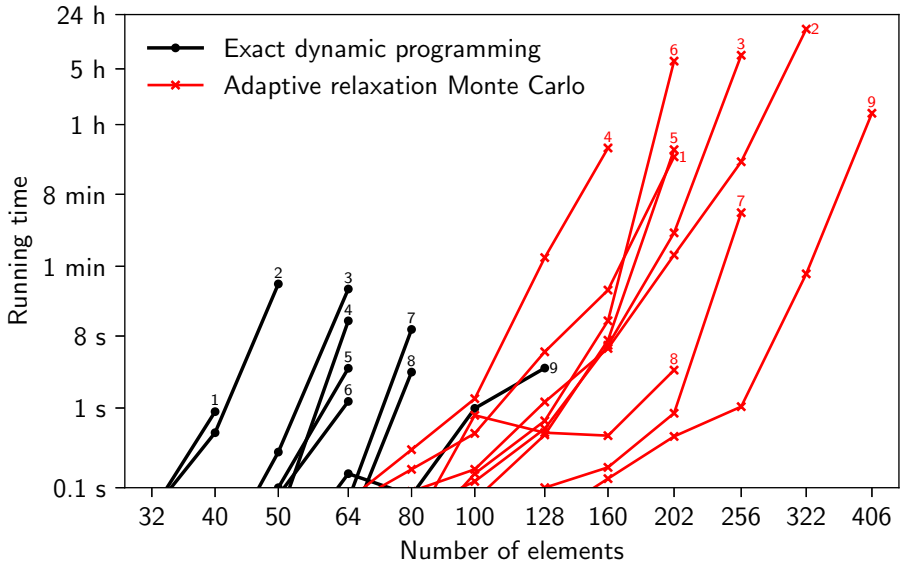


Figure 3.3: The running times of the algorithms as functions of the number of elements in the poset. Poset types 1–4 are randomly generated, and types 5–9 are derived from benchmark Bayesian networks by first obtaining the poset as the reachability relation in the DAG, and then restricting the set of elements into a randomly chosen subset of given size. The quality parameters for ARMC are set such that it produces an $(1, 1/4)$-approximation. Each plot line ends when the algorithm exceeds the time limit of 24 hours or the memory limit of 8 gigabytes. In these experiments, the exact dynamic programming algorithm always runs out of memory before running out of time.

ARMC algorithm that uses only one operation of this type. While ARMC is useful in instances with up to roughly 200 elements, we see in Figure 3.3 that the algorithm quickly becomes infeasible when increasing the number of elements further, because it can no longer find close relaxations of the poset that are easy enough for the exact dynamic programming algorithm. Thus, in order to improve the scalability of the algorithm, we have to break the task into smaller operations.

The polynomial-time randomized approximate counting scheme presented by Brightwell and Winkler [8] breaks the problem into multiple smaller parts by using a sequence of intermediate partial orders $(P_i)_{i=0}^k$ between $P_0 = P$ and $P_k = L \in \mathcal{L}(P)$ which is increasing in the sense that for all $1 \leq i \leq n$, P_i is an extension of P_{i-1}. The algorithm estimates $\mu = \ell(L)/\ell(P) = \ell(P)^{-1}$ by decomposing it into factors $(\mu_i)_{i=1}^k$ defined

by $\mu_i = \ell(P_i)/\ell(P_{i-1})$, and estimating each of them using a Monte Carlo sampling operation. When we multiply all the factors, everything except $\ell(P_k)/\ell(P_0) = \mu$ cancels out. This approach, which we refer to as the *telescopic product* method due to the telescopic cancellation, is more generally applicable to other combinatorial counting problems with self-reducible structure [44, 46].

The sequence of posets $(P_i)_{i=0}^k$ should be chosen such that the factors μ_i are not too small, because otherwise we need a lot samples to estimate them. Brightwell and Winkler [8] achieve this by choosing the sequence such that the factors are bounded from below by a positive constant. This is done by using an iterative method that obtains the next poset $P_i = (A, \preceq)$ from P_{i-1} by choosing a pair of incomparable elements (a, b) and adding one of the constraints $a \preceq b$ and $b \preceq a$ along with the constraints that follow from it by transitivity. Making the right choice among the two possible directions for the added constraint is critical, as one of them might lead to μ_i being very small. Denote the two possible values of (P_i, μ_i) for the two different choices of the added constraint by (P_i', μ_i') and (P_i'', μ_i''). The sets $\mathcal{L}(P_i')$ and $\mathcal{L}(P_i'')$ partition the set of linear extensions of P_{i-1} based on the ordering of a and b, and thus $\mu_i' + \mu_i'' = 1$. We can get an estimate for μ_i' by sampling, and if the estimate is larger than $1/2$, we choose $P_i = P_i'$, and otherwise we choose $P_i = P_i''$. Either way, if we use enough samples to estimate μ_i', then we will get the desired property that $\mu_i \geq 1/4$ with high probability.

A naive implementation of this method may result in a sequence of posets of length $k = O(n^2)$, as we can add at most $n(n-1)/2$ constraints before reaching a linear poset. However, this can be improved by observing that the way we form the sequence of posets is analogous to comparison sorting [8]: on the ith step, P_{i-1} is the poset currently discovered by the algorithm, and then the algorithm compares elements a and b, obtaining the augmented poset P_i. Thus, by emulating an $O(n \log n)$-time comparison sorting algorithm for the choice of a and b on each iteration, we get sequence length $k = O(n \log n)$. Figure 3.4 shows an example of a sequence of posets obtained this way.

While Brightwell and Winkler [8] were already able to prove a polynomial time complexity bound for this approximate linear extension counting algorithm, their analysis is somewhat loose, and the more recent improvements on the time complexity bound for linear extension sampling [9,38] directly translate to further improvements to the time complexity bound. The best known bound for the number of linear extension samples we need for estimating each factor μ_i to obtain an (ϵ, δ)-approximation of $\ell(P) = \mu^{-1}$ is

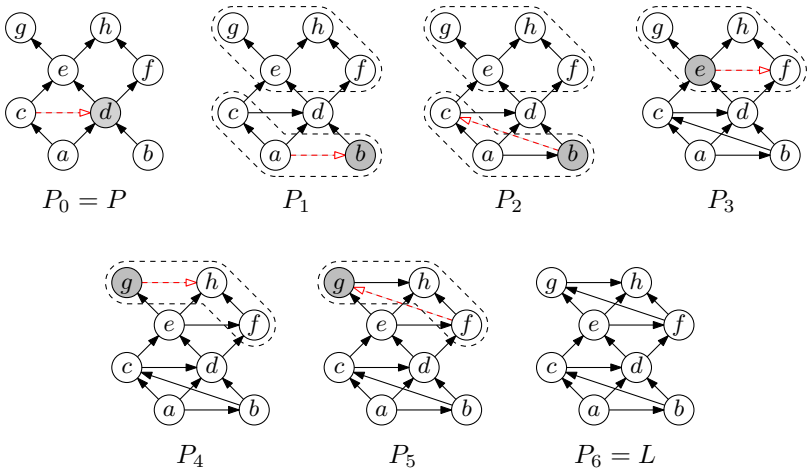


Figure 3.4: An example of the increasing sequence of posets $(P_i)_{i=0}^{6}$ obtained by simulating the Quicksort algorithm. The algorithm begins with the original poset $P_0 = P$, and picks a pivot element d. The only element that is incomparable with d is c, and by sampling linear extensions of P_0, the algorithm decides that adding the constraint in the direction $c \to d$ instead of direction $d \to c$ to form the next poset P_1 results in smaller reduction in the number of linear extensions. After this, the pivot d has partitioned the poset into its predecessors $\{a, b, c\}$ and successors $\{e, f, g, h\}$. The algorithm goes on to recursively order the set of predecessors, choosing b as the pivot and comparing it with the other elements, and the set of successors, first using pivot e and then g, eventually reaching a linear extension L. This sequence gradually reduces the number of linear extensions from $\ell(P) = 29$ to 1, because $(\ell(P_i))_{i=0}^{6} = (29, 15, 10, 5, 3, 2, 1)$.

$O(\epsilon^{-2} k \log \delta^{-1})$; the details on how to obtain this bound are in Theorem 1 of Paper I. Combining this with the bound $O(n \log n)$ for the number of factors k and the expected time complexity $O(n^3 \log n)$ of the best known exact linear extension sampler [38], we get that the total time complexity of the approximate linear extension counting algorithm is $O(\epsilon^{-2} n^5 \log^3 n \log \delta^{-1})$. As this bound is polynomial in both ϵ^{-1} and n, the algorithm is a *fully polynomial randomized approximation scheme* (FPRAS).

Even though the algorithm has a polynomial running time bound, it is still not very practical: the degree of the polynomial bound is very high, and the algorithm rigidly uses the expensive sampling operation for all steps, missing opportunities for exploiting special structure in the poset. Next, we will describe the *structure decomposition* idea that we presented in Paper II to mitigate these shortcomings in the algorithm.

If we look at the way the sequence of intermediate posets is formed by using the Quicksort algorithm as the comparison sorting algorithm, we observe that it creates a particular kind of special structure in the poset: if the algorithm chose $p \in A$ as the first pivot element, it will add constraints concerning it until eventually we get a poset $P_i = (A, \preceq)$ in which we the predecessors A' and successors A'' of p form a partition of the set of other elements $A \setminus \{p\}$. By using transitivity we get that $x \preceq y$ for all $x \in A' \cup \{p\}$ and $y \in A'' \cup \{p\}$. Thus the A' part is completely independent from the A'' part in P_i (and all the subsequent posets): each linear extension of P_i is obtained by concatenating an arbitrary linear extensions of $P_i[A']$, the pivot element p and an arbitrary linear extension of $P_i[A'']$.

We improve the algorithm by allowing it decompose the poset into parts and handle each part independently. More exactly, when forming the sequence of posets, if it happens that the set of elements S of the poset $P_i = (S, \preceq)$ can be decomposed into two sets X and Y such that for all $x \in X$ and $y \in Y$ it holds that $x \preceq y$, then we actually split the sequence into two branches, starting from $P_i' = P_i[X]$ and $P_i'' = P_i[Y]$. The total product μ will include the factors μ_j from both of the branches. This gives us the correct result, because due to the independence property it holds that $\ell(P_i) = \ell(P_i')\ell(P_i'')$. We can also branch using the reduction rule (3.2) from the exact linear extension counting algorithm for the case that each element of X is incomparable with each element of Y; in this case, we need to multiply μ by the factor $\binom{|S|}{|X|}^{-1}$. These decomposition rules can be used to further decompose already decomposed posets, which means that we effectively get a tree of posets instead of a sequence. Figure 3.5 shows an example of the decomposition tree structure.

We implemented the algorithm and compared it to ARMC and the original telescopic product algorithm without structure decompositions. In addition to the state-of-the-art linear extension sampler with $O(n^3 \log n)$ expected time complexity [38], we tested a more recent sampler due to Huber [39] that is based on Gibbs sampling. While the sampler is known to be efficient only for posets in which there are no elements x, y, z such that $x \prec y \prec z$, it works correctly for other posets too.

The results for the experiments from Paper II are summarized in Figure 3.6. We see that the decomposing the structure and switching to the Gibbs sampler each result in a speedup of roughly an order of magnitude. The running times of the telescopic product methods are much less varied than those of ARMC. In these experiments, ARMC outperforms the other algorithms in every instance, but by extrapolation we presume that if we increased the time limit, the fastest telescopic product method would

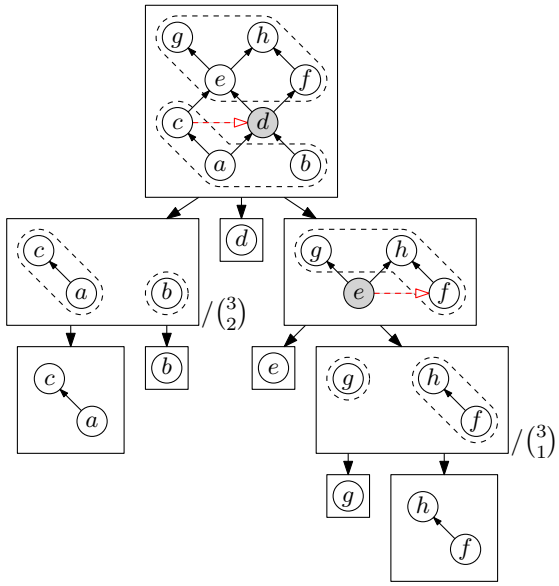


Figure 3.5: The tree obtained when using the Quicksort algorithm and the decomposition rules to the same poset P as in the example of Figure 3.4. After adding the constraint $c \to d$, the poset decomposes into three parts. The first one decomposes further into incomparable parts $\{a, c\}$ and $\{b\}$, which are linear posets. The second part is also linear, as it consists of only the pivot d. In the third part we have to add the constraint $e \to f$, but after that everything decomposes into linear parts without the need for additional sampling. Compared to the telescopic product algorithm without the decomposition rules, this method reduces the number of factors we need to estimate by sampling from six to two, and in the second sampling operation, the number of elements in the poset is 4 instead of 8.

eventually outperform ARMC. Furthermore, the telescopic product method should parallelize better than ARMC, as it spends most of the time drawing independent linear extension samples, while ARMC uses half of the time running the exact dynamic programming algorithm, which is harder to parallelize.

In addition to the practical improvements, we were able to prove that the algorithm runs in $O(\epsilon^{-2} n^5 \log^2 n \log \delta^{-1})$ time, which means that the structure decomposition idea improves the time complexity bound by a factor of $O(\log n)$. To achieve this bound, we use a variant of Quicksort that always uses the median element as the pivot, because then after adding $O(n)$ constraints, the poset has been split into exactly half, and this halving

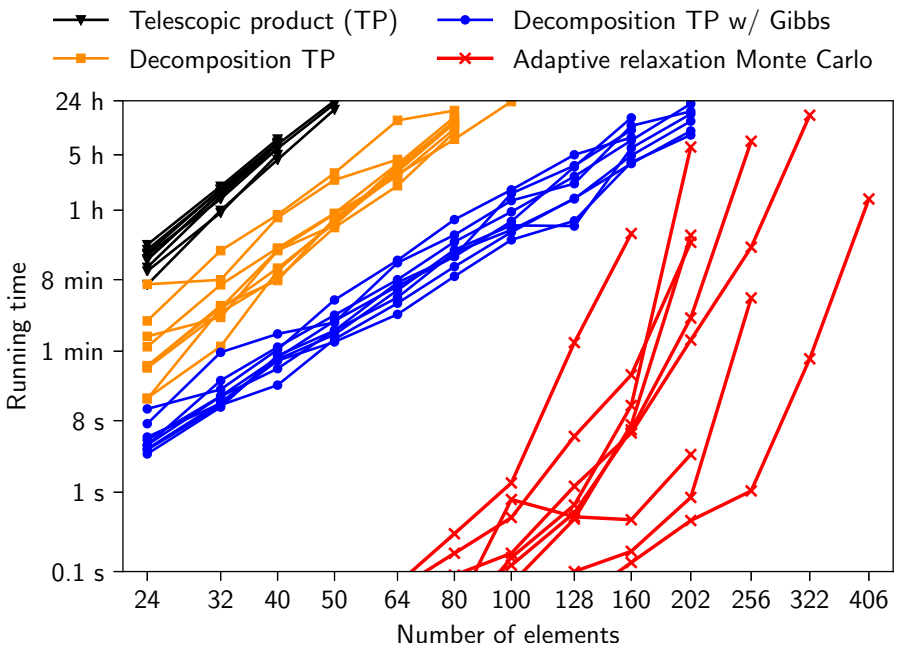


Figure 3.6: The running times of the telescopic product (TP) algorithms and ARMC as functions of the number of elements in the same 9 types of posets as in Figure 3.3, one plot line per algorithm and type of poset.

process recursively continues on both branches after adding $O(n)$ additional constraints. As the time complexity of linear extension sampling is super-linear and the poset sizes decrease geometrically, the time complexity of the first $O(n)$ of the $O(n \log n)$ sampling operations dominate the total time complexity. We believe that this time complexity bound is the best known worst-case bound for approximate linear extension counting. However, in cases where the poset is sufficiently dense, that is, $\ell(P) = \exp(o(n\sqrt{\log n}))$, the output-sensitive bound $O(\epsilon^{-2}(\log \ell(P))^2 n^3 \log n \log \delta^{-1})$ of the algorithm due to Banks et al. [3] is better.

3.4 Mixing times of Markov chains

In Section 3.2, we showed how to sample linear extensions uniformly at random by using the exact linear extension counting algorithm based on dynamic programming. While this approach worked well for sampling linear extensions of the relaxation poset in ARMC, it is feasible for only a very limited class of posets, and even in those it requires a long precomputation phase for every poset we use it for. For linear extension sampling in more

scalable approaches (such as the telescopic product method), we sample linear extensions using a *Markov chain*, that is, a random walk in the set of linear extensions. By carefully designing the transition process for the chain, we can ensure that if we simulate the chain for sufficient number of transitions, the resulting state will give us a sample from a distribution that is close to the desired distribution.

A Markov chain is defined as a random sequence of states $(X_t)_{t=0}^{\infty}$ in a domain Ω, in which for all $t > 0$ the probability distribution of the state X_t depends only the previous state X_{t-1}, or in other words, we have the conditional independence $X_t \perp\!\!\!\perp (X_s)_{s=0}^{t-2} \mid X_{t-1}$ for all $t > 0$. In the case of linear extension sampling, the domain Ω is typically the set of linear extensions $\mathcal{L}(P)$ for some poset P. We limit our consideration to *homogeneous* Markov chains, in which the way the transition from X_{t-1} to X_t works does not depend on t; more exactly, for all $x, y \in \Omega$ and $t > 0$

$$\Pr(X_t = y \mid X_{t-1} = x) = \Pr(X_1 = y \mid X_0 = x).$$

This probability is known as the transition probability from x to y. Designing the Markov chain means setting the transition probabilities such that the chain behaves as desired. The transition is typically implemented as an algorithm that computes the new state X_t from the previous state X_{t-1} using a random generator.

The Karzanov–Khachiyan chain [48] in the set of linear extensions of a poset $P = (A, \preceq)$ uses the following kind of algorithm for the transition from X_{t-1} to X_t: With probability 1/2, the chain does nothing, which means that $X_t = X_{t-1}$. In the other case, let $\mathrm{List}(X_{t-1}) = (x_1, x_2, \ldots, x_n)$. We sample an index $1 \le j \le n-1$ uniformly at random, and attempt to swap the elements x_j and x_{j+1}. This is allowed only if $x_j \not\preceq x_{j+1}$; in that case, we set $\mathrm{List}(X_t) = (x_1, \ldots, x_{j-1}, x_{j+1}, x_j, x_{j+2}, \ldots, x_n)$. Otherwise, the chain does nothing.

To prove that the chain has the desired property that the distribution of X_t converges to the uniform distribution over $\Omega = \mathcal{L}(P)$ as $t \to \infty$ for any starting state $X_0 = s \in \Omega$, we need to prove three properties: uniform stationary distribution, irreducibility, and aperiodicity [58]. The chain has uniform stationary distribution if it holds that whenever X_0 is distributed uniformly in Ω, then X_1 is also. This property follows from the fact that the chain is symmetric: for all $x, y \in \Omega$, the transition probability from x to y is the same as the transition probability from y to x. The irreducibility and aperiodicity properties ensure that we can reach all states at all times $t \ge t'$ for some t'; they follow from the fact that a sequence of swaps between any pair of states can be constructed explicitly and the chain stays in the same state with positive probability.

This convergence result alone does not tell us how many transitions we need to simulate to get sufficiently close to the uniform distribution. While we could simply run the chain for a long time and hope that it is enough, it is sometimes possible to prove a bound on the number of transitions we need, which is captured by the notion of the *mixing time* of the chain. Denote the distribution of X_t when the initial state X_0 is fixed to $s \in \Omega$ by p_s^t, that is, for all $S \subseteq \Omega$

$$p_s^t(S) = \Pr(X_t \in S \mid X_0 = s).$$

To measure the distance of the distribution p_s^t to the desired stationary distribution π of the chain (in the uniform case, $\pi(S) = |S|/|\Omega|$), we use the *total variation distance*

$$||p_s^t - \pi|| = \max_{S \subseteq \Omega} |p_s^t(S) - \pi(S)|,$$

that is, the worst case absolute error in the probability of an event S. The mixing time $\tau_{\text{mix}}(\epsilon)$ for distance $\epsilon > 0$ of the chain is defined as the smallest t such that $||p_s^t - \pi|| \leq \epsilon$ for all $s \in \Omega$. Without specific distance, the mixing time τ_{mix} is defined as $\tau_{\text{mix}}(1/4)$. Simulating the chain for a small multiple of the mixing time will make it quickly reach smaller distances too, because it holds that $\tau_{\text{mix}}(\epsilon) \leq \lceil \log_2 \epsilon^{-1} \rceil \tau_{\text{mix}}$ for all $\epsilon > 0$ [58].

If we want to estimate a probability of an event $S \subseteq \Omega$ using Monte Carlo sampling, then the mixing time $\tau_{\text{mix}}(\epsilon)$ tells us directly how long we should simulate the chain such that the absolute error of the probability of the event given by the chain $p_s^t(S)$ compared to the correct probability $\pi(S)$ is at most ϵ. For instance, Brightwell and Winkler [8] were able to show polynomial time complexity bound for the original telescopic product algorithm using the mixing time bound $\tau_{\text{mix}} = O(n^6 \log n)$ for the Karzanov–Khachiyan chain [48], which was the best known bound at the time. We can also adapt our improved version of the algorithm introduced in Section 3.3 to directly use a Markov chain, yielding time complexity bound $O(\epsilon^{-2} \tau_{\text{mix}}\, n^2 \log n \log(n/\epsilon) \log \delta^{-1})$. This, combined with the improved mixing time bound $\tau_{\text{mix}} = O(n^3 \log n)$ [91], introduces only a factor $O(\log(n/\epsilon))$ of overhead to compared to the bound $O(\epsilon^{-2} n^5 \log^2 n \log \delta^{-1})$ that we achieved using an exact sampler based on the Karzanov–Khachiyan chain [38]. Thus, finding a Markov chain in $\mathcal{L}(P)$ with uniform stationary distribution and better mixing time bound than the Karzanov–Khachiyan chain will give us faster approximate linear extension counting algorithm.

Unfortunately, mixing time bounds are often hard to prove. For instance, the mixing time bound $O(n^3 \log n)$ for the Karzanov–Khachiyan chain, which is known to be tight in the worst case [91], is a result of

research spanning a long period of time [9, 48, 91]. For this reason, in Paper I we use experimental methods to evaluate the mixing times of Markov chains. The results of these experiments can then be used to guide the development of new chains and bounds.

Generally, computing the mixing time of a given chain is difficult [5,37], having time complexity that is at least linear in the number of states. Even in the subproblem of testing whether the distribution of a given sampler is close to uniform, the required number of samples is of the order $|\Omega|^c$, where c is a positive constant [4,86]. In the case of linear extension sampling, $|\Omega|$ can be as large as $\Theta(n!)$, which means that these methods are not feasible. Using fewer samples is sometimes possible if the sampler satisfies certain special properties. For instance, Chakraborty and Meel [11] formulated mild non-adversariality assumptions under which they could experimentally determine whether the distribution of a sampler is close to uniform over the set of satisfying assignments of a Boolean formula. However, methods like this are based on domain-specific structure, making them difficult to adapt to other domains. We opt for a simpler approach, in which we modify the definition of mixing time to make it easier to estimate, and use the resulting quantity as a proxy for understanding the behavior of the mixing time.

The modified variant of mixing time, which we call the *relative mixing time* $\tau_{\text{mix}}^{\mathcal{A}}$ with respect to a set of events $\mathcal{A} \subseteq 2^{\Omega}$, is obtained by replacing the total variation distance in the definition of mixing time by the *relative distance*

$$||p_s^t - \pi||_{\mathcal{A}} = \max_{S \in \mathcal{A}} |p_s^t(S) - \pi(S)|.$$

Because the relative distance is a lower bound for the total variation distance $||p_s^t - \pi||$, the relative mixing time $\tau_{\text{mix}}^{\mathcal{A}}$ is also a lower bound for the standard mixing time τ_{mix}, and equality in both of them hold in the case of the maximal set of events $\mathcal{A} = 2^{\Omega}$.

The choice of the set of events $\mathcal{A}$ is specific to the domain we use the method for. In the experiments of Paper I, we consider Markov chains on linear extensions of a poset $P = (A, \preceq)$. In this case, we choose the set such that it contains for all pairs (a, b) of distinct elements the event that element a occurs before element b in the linear extension, or in other words,

$$\mathcal{A} = \{\{(A, \preceq') \in \mathcal{L}(P) \,:\, a \preceq' b\} \,:\, a, b \in A, a \neq b\}.$$

The relative mixing time with respect to this set of events is directly applicable to the telescopic product algorithm, because it contains exactly the types of events for which the algorithm estimates probabilities using Monte Carlo sampling. In addition, we hope that this set of events is extensive enough in the sense that the relative mixing time will give us a

good idea about the standard mixing time too. In our method, we will also need to be able to compute the probability in the stationary distribution π for each event $S \in \mathcal{A}$. For this set of events we can do this, because if $P' = (A, \preceq')$ is the poset obtained from P by adding the constraint $a \preceq' b$, then $\pi(S) = \ell(P')/\ell(P)$, and by limiting our consideration to posets of at most 50 elements, we can compute $\ell(P')$ and $\ell(P)$ using the exact dynamic programming algorithm due to Kangas et al. [47].

The method estimates the relative mixing time $\tau^{\mathcal{A}}_{\text{mix}}(\epsilon)$ by using binary search to find a value of t such that the relative distance $\mathcal{D}^t_s(\mathcal{A}) = ||p^t_s - \pi||_{\mathcal{A}}$ reaches ϵ. The use of binary search is justified, because the total variation distance $||p^t_s - \pi||$ decreases as t increases [64], and we confirmed in our preliminary experiments that the relative distance, which approximates the total variation distance, also exhibits similar behavior. To estimate each relative distance $\mathcal{D}^t_s(\mathcal{A})$ queried by the binary search, we use Monte Carlo sampling: we carry out many independent simulations of the chain for t transitions, and compute statistics about how many of the resulting states are members of each event $S \in \mathcal{A}$. This gives us an estimate $\tilde{p}^t_s(S)$ of $p^t_s(S)$ for each $S \in \mathcal{A}$, and then by computing $\max_{S \in \mathcal{A}} |\tilde{p}^t_s(S) - \pi(S)|$ we get an estimate of the relative distance $\mathcal{D}^t_s(\mathcal{A})$.

In the experiments of Paper I, we estimated relative mixing times for the Karzanov–Khachiyan chain and two alternative chains: the *insertion chain*, which was mentioned by Bubley and Dyer [9] as an alternative for the Karzanov–Khachiyan chain, and the novel *shuffle chain*. The chains use the following algorithms for the transitions from X_{t-1} to X_t, where $\text{List}(X_{t-1}) = (x_1, x_2, \ldots, x_n)$:

- Insertion chain: First, we draw indices i, j from the uniform distribution on $\{1, 2, \ldots, n\}$. The idea is to attempt to move element x_i to the position of x_j, moving other elements out of the way. For example, if $j \geq i$, we first check that none of the elements $x_{i+1}, x_{i+2}, \ldots, x_j$ are successors of x_i in P, and if it holds, we set $\text{List}(X_t) = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_j, x_i, x_{j+1}, \ldots, x_n)$. Otherwise we do nothing, setting $X_t = X_{t-1}$. The case $j < i$ is symmetric.

- Shuffle chain: We begin by drawing endpoints $1 \leq i < j \leq n$ for an interval of elements $I = \{x_i, x_{i+1}, \ldots, x_j\}$ from a distribution that favors short intervals: the probability for interval length $2 \leq l \leq n$ is $n/[l(l-1)(n-1)]$, which yields that the expected length is $\Theta(\log n)$. We then partition the poset restricted to the interval $P[I]$ into mutually incomparable sets of elements, and randomly reinterleave them to form a sequence $(x'_i, x'_{i+1}, \ldots, x'_j)$ such that within each set of the par-

tition, the elements stay in the same order as in X_{t-1}. The new state is then given by $\mathrm{List}(X_t) = (x_1, \ldots, x_{i-1}, x'_i, \ldots, x'_j, x_{j+1}, \ldots, x_n)$.

Both of these chains have polynomial mixing time bounds: Bubley and Dyer [9] proved the mixing time bound $O(n^4 \log n \log \ell(P))$ for the insertion chain, and we proved the mixing time bound $O(n^4 \log^2 n)$ for the shuffle chain by using the fact that it often does the same transition as the Karzanov–Khachiyan chain. However, the results of our experiments showed that these bounds are very loose in practice: in all the randomly generated types of poset instances we tried, the growth of the relative mixing times of the insertion and shuffle chains as functions of the number of elements n resembled that of a quadratic function, though for insertion chain we were able to create a poset where the growth is cubic. In contrast, the growth of the relative mixing times of the Karzanov–Khachiyan chain seemed to follow the $O(n^3 \log n)$ bound.

In the alternative chains, the time complexity of simulating one transition is higher: the worst-case expected time complexity is $O(n)$ and $O(n \log n)$ for the insertion and shuffle chain, respectively, while a transition in the Karzanov–Khachiyan chain runs in constant time. Still, the slowdown in the transition does not completely negate the improved mixing times, because the time complexity bound $O(n)$ is very pessimistic for the insertion chain in typical instances and the constant factor in the time complexity bound for one shuffle chain transition is very small in a bit-parallel implementation.

The success of the insertion chain in these experiments led us to investigate exact samplers based on similar Markov chains, resulting in improved methods for linear extension counting. For instance, in Paper II we used the exact sampler due to Huber [39], which lacks general-case complexity bounds, to speed up the telescopic product method by an order of magnitude. Moreover, in the next section we will present the main contribution of Paper III: an exact sampler based on a chain similar to the insertion chain, enabling faster linear extension counting in large instances.

3.5 Continuous-space order constraints

In Section 3.3, we saw how we can use the telescopic product approach [46] to count linear extensions approximately by sampling them uniformly at random. Huber and Schott [41] developed an alternative method known as the *Tootsie Pop algorithm* (TPA) for this problem of estimating the ratio of sizes of two nested sets by sampling. In TPA, we need the two sets $S_0 \subseteq S_1$ to be measurable sets in a continuous measurable space. To estimate the

ratio of their measures $\mu(S_1)/\mu(S_0)$, the algorithm utilizes an increasing continuum of intermediate sets between S_0 and S_1, in which we have a set S_β for each $\beta \in (0,1)$ such that for all $0 \leq \beta \leq \beta' \leq 1$ it holds that $S_\beta \subseteq S_{\beta'}$ and the measure $\mu(S_\beta)$ grows continuously as a function of $\beta \in [0,1]$.

The TPA algorithm consist of multiple independent runs, each of which iteratively constructs a decreasing sequence $(\beta_i)_{i=0}^k$ of numbers in $[0,1]$, which corresponds to decreasing sequence of sets $(S_{\beta_i})_{i=0}^k$ in the continuum. The sequence starts from the outer set S_1 and ends as soon as the inner set S_0 is reached. We obtain β_i from β_{i-1} by sampling an element $x \in S_{\beta_{i-1}}$ uniformly at random, and setting $\beta_i = \inf\{\beta \in [0,1] : x \in S_\beta\}$. This means that if $x \in S_0$, we end the sequence by setting $\beta_i = 0$, and otherwise we choose β_i such that x is on the boundary of S_{β_i}.

The ratio of interest $\mu(S_1)/\mu(S_0)$ affects the sequence length k, because in the sequence we decrease $\mu(S_\beta)$ on each step by a factor that is uniformly distributed in $[0,1]$ until reaching $\mu(S_0)$. This is equivalent to decreasing $\log(\mu(S_\beta))$ by subtracting exponentially distributed random values from it, and thus $k-1$ is Poisson distributed with mean $r = \log(\mu(S_1)/\mu(S_0))$. To obtain an (ϵ, δ)-approximation of $\mu(S_1)/\mu(S_0)$, we estimate r by taking an average of $k-1$ over multiple runs; the expected number of samples we need to draw in total is $O(\epsilon^{-2} r^2 \log \delta^{-1})$ [41].

TPA can be seen as a complementary approach compared to the telescopic product estimator: while the telescopic product scheme uses a fixed monotone sequence of sets between the sets of interest and estimates the ratio in each step using sampling, in TPA the step ratios all have a known distribution, and they are used to form dynamic monotone sequences from S_1 to S_0 whose lengths give us the estimator.

For the problem of approximately counting linear extensions of a given poset P, there exists a TPA algorithm due to Banks et al. [3], which has polynomial time complexity bound $O(\epsilon^{-2}(\log \ell(P))^2 n^3 \log n \log \delta^{-1})$. The algorithm embeds the discrete problem into a continuous space by augmenting the set of linear extensions $\mathcal{L}(P)$ with a continuous dimension. More exactly, the outer set S_1 is $\mathcal{L}(P) \times [0,1]$. In this section, we present the main contribution of Paper III: a novel TPA algorithm for counting linear extensions based on a completely different embedding of the problem into continuous space.

We base the algorithm on the observation that if $P = (A, \preceq)$ is the input poset and we define

$$S_0 = \{x \in [0,1]^A : x_a \leq x_b \text{ for all } (a,b) \in \preceq\},$$

then we can compute $\ell(P)$ as $\mu(S_0)n!$, where $\mu(S_0)$ is the n-dimensional volume of the polytope S_0 [21]. We estimate $\mu(S_0)$ using TPA by comparing

it to $\mu(S_1) = 1$, where S_1 is the whole hypercube $[0,1]^A$. The sets S_0 and S_1 are connected by a continuum of sets S_β obtained by adding β as a slack variable to the inequality constraints as follows:

$$S_\beta = \{x \in [0,1]^A : x_a - x_b \leq \beta \text{ for all } (a,b) \in \preceq\}.$$

To complete the TPA algorithm, we need an algorithm that can sample from the set S_β for any $0 < \beta \leq 1$ uniformly at random. This problem is subsumed by the problem of sampling from the hypercube $[0,1]^n$ under *generalized order constraints*, that is, from the subset

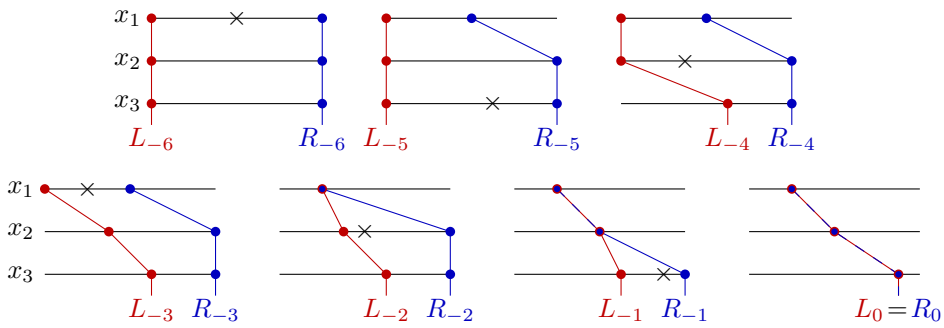
$$\Omega = \{x \in [0,1]^n : x_i - x_j \leq s_{ij} \text{ for all } 1 \leq i < j \leq n\},$$

where $s_{ij} \geq 0$ for all $1 \leq i < j \leq n$. To do this, we use a very simple Markov chain which we call the *continuous relocation chain*. In the chain, the next state y is obtained from the previous state x is using the following transition: We sample dimension $1 \leq i \leq n$ and coordinate $p \in [0,1]$ uniformly at random, and define the point y' by setting $y'_i = p$ and $y'_j = x_j$ for all $1 \leq j \leq n$ such that $j \neq i$. Now if $y' \in \Omega$, we set $y = y'$, and otherwise we do nothing, setting $y = x$.

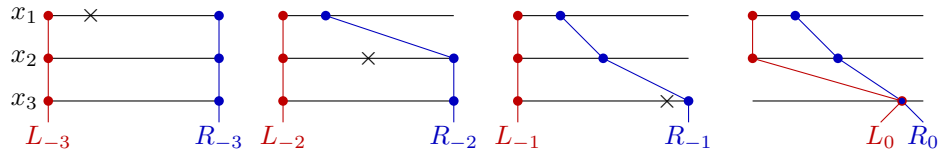
The uniform distribution is a stationary distribution for this chain, because the transition densities are symmetric. The traditional way of using the chain would be to prove a mixing time bound for it and simulate the chain for a number of transitions given by the bound to obtain a single sample. However, we have not been able to prove a mixing time bound for this chain. Furthermore, even if we had a mixing time bound, we would still only get samples from an approximately uniform distribution, and in TPA we need samples exactly from the uniform distribution in order to get guarantees on the correctness of the result [41]. For this reason, we need *perfect simulation* [40], which means using the chain to obtain samples exactly from the stationary distribution.

The continuous relocation chain is similar to the Markov chain due Huber [39], which can be used to sample points uniformly at random from Ω in the special case where $s_{ij} \in \{0,1\}$ for all $1 \leq i < j \leq n$. For this chain, perfect simulation was achieved using a technique known as *monotone coupling from the past* due to Propp and Wilson [74]. We will now show that we can use the same technique also for the continuous relocation chain for sampling from Ω in the more general case of arbitrary weights $s_{ij} \geq 0$.

To obtain a sample exactly from the stationary distribution, we need the chain to be *coupled* at the end of the simulation, which means that the resulting state does not depend on the initial state we chose. However, we cannot do this simply by simulating the chain until it has coupled, as



(a) Successful coupling in the case $N = 6$



(b) Incomplete coupling in the case $N = 3$

Figure 3.7: We visualize the bounding chains $(L_t)_{t=-N}^{0}$ and $(R_t)_{t=-N}^{0}$ when using monotone coupling from the past with the continuous relocation chain for sampling from $\Omega = \{x \in [0,1]^3 : x_1 \leq x_2 \leq x_3\}$. We use the symbol $\times$ to mark the random numbers (the dimension $i \in \{1,2,3\}$ and coordinate $p \in [0,1]$) for each transition. In (a), we see that if we use $N = 6$ iterations, we achieve coupling and obtain a sample $L_0 = R_0$, whereas in (b) we see that $N = 3$ is not sufficient for coupling with these random numbers.

changing the number of iterations based on the coupling status will bias the distribution. For this reason, we use *coupling from the past*: we index the chain using nonpositive numbers $(X_t)_{t=-\infty}^{0}$, and consider X_0 be the final state of the chain that is obtained after an arbitrary long simulation. Now if we simulate the chain from X_{-N} up to X_0 for some $N > 0$, then either the chain is coupled and we get an unbiased sample X_0, or we need to run the chain starting from further into past, in which case we double N and retry, always using the same random numbers (in our case, the dimension $i = i_t$ and coordinate $p = p_t$) for the same transition from X_{t-1} to X_t.

To implement this algorithm, we need a fast routine for detecting cases where the chain has coupled. In monotone coupling from the past, this is done by using a partial order $\sqsubseteq$ that is preserved by the transition in the sense that for all $x, y \in \Omega$ such that $x \sqsubseteq y$, the same ordering $x' \sqsubseteq y'$ holds for the states x' and y' we get from x and y by applying the transition using the same random numbers. There must also exist elements $a, b \in \Omega$

that satisfy $a \sqsubseteq x \sqsubseteq b$ for all $x \in \Omega$. For this chain, the partial order

$$\sqsubseteq = \{(x, y) \in \Omega \times \Omega \,:\, x_i \leq y_i \text{ for all } 1 \leq i \leq n\}$$

with the elements $a = (0, \ldots, 0)$ and $b = (1, \ldots, 1)$ satisfies the conditions. To detect coupling when simulating the chain from X_{-N} to X_0, we run two parallel *bounding chains* $(L_t)_{t=-N}^{0}$ and $(R_t)_{t=-N}^{0}$ that use the same random numbers as (X_t). The extremal elements a and b are used as the initial states L_{-N} and R_{-N}. Now for any choice of initial state $X_{-N} \in \Omega$, it holds that $L_{-N} \sqsubseteq X_{-N} \sqsubseteq R_{-N}$, and by repeatedly applying the monotonicity property we get that $L_0 \sqsubseteq X_0 \sqsubseteq R_0$. Now, if it happens that $L_0 = R_0$, then necessarily also $X_0 = L_0$, which means that the chain has coupled. Figure 3.7 shows an example of the algorithm.

We know that the expected time complexity of this sampling algorithm is finite, because it follows from the fact that coupling is detected with positive probability for some $N > 0$. One advantage of coupling from the past compared to directly simulating the chain is that even though we do not have any bounds on the number of transitions N needed for coupling, we can still use the algorithm in practice, as it knows when to stop.

To optimize this TPA algorithm further, we augmented it with the idea of the ARMC algorithm of using a close relaxation $R = (A, \preceq')$ of P to make the estimated ratio smaller. We do this by using

$$S_1 = \{x \in [0,1]^A \,:\, x_a \leq x_b \text{ for all } (a, b) \in \preceq'\},$$

as the outer set in TPA instead of $[0,1]^A$, which means that we estimate the ratio $\mu(S_1)/\mu(S_0) = \ell(R)/\ell(P)$. To connect S_0 and S_1, we use the following continuum of sets

$$S_\beta = \{x \in [0,1]^A \,:\, a \preceq' b \Rightarrow x_a \leq x_b \text{ and } a \preceq b \Rightarrow x_a - x_b \leq \beta \;\; \forall a, b \in A\}.$$

For this optimized algorithm that we call *relaxation Tootsie Pop*, choosing the relaxation R to be close to P is far less critical than for ARMC, because the dependence of the running time on the ratio $\ell(R)/\ell(P)$ is logarithmic instead of linear. Thus in the optimized algorithm we employ a heuristic algorithm to find the relaxation from a class of posets in which we can count linear extensions in polynomial time, that is, *series-parallel* posets [65] and trees [2].

In Paper III, we evaluated the practical performance of the algorithm by comparing it to ARMC and the telescopic product method with the structure decomposition optimization by once more using the same experimental setup as Section 3.2 and 3.3. The results are shown in Figure 3.8.

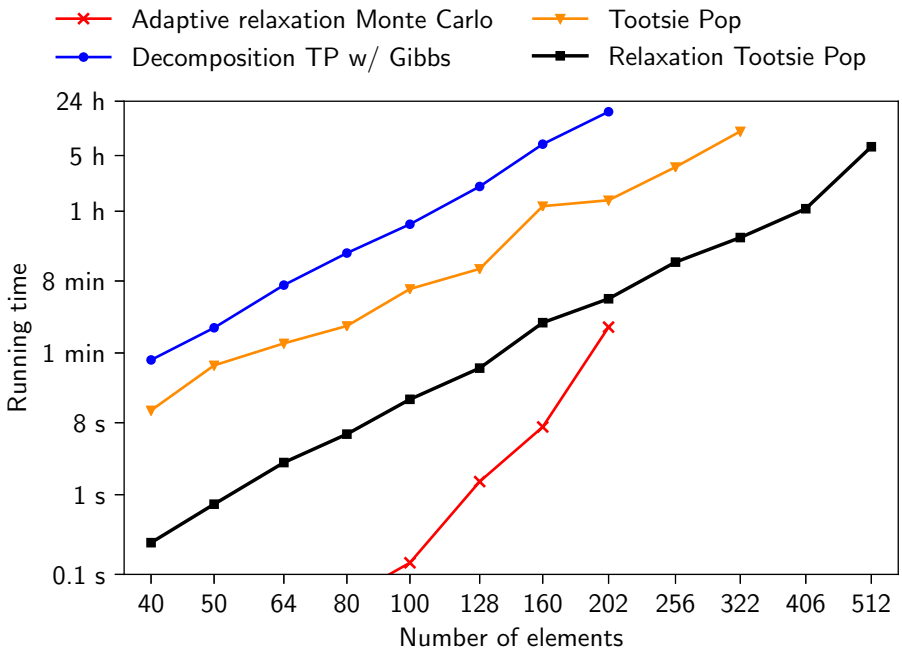


Figure 3.8: The running times of the TPA algorithms are shown as function of the number of elements in the poset along with the running times shown in Figure 3.6 for the ARMC method and the fastest telescopic product method. For readability, only the medians over all the 9 datasets are shown—a more detailed breakdown is shown in Paper III.

From the results we see that the basic TPA algorithm already outperforms the telescopic product method and can be used to solve larger instances than ARMC. After adding relaxation optimization, the running times improve by an order of magnitude, which extends the range up to $n = 512$ for our time limit of 24 hours. Even though we do not have a polynomial time complexity bound, the steady growth of the running times suggests that the algorithm could scale to even larger instances if the algorithm is parallelized or allowed to use more time.

Chapter 4

Sampling DAGs from modular distributions

In this chapter, we consider the problem of sampling DAGs from a modular distribution, motivated by its direct application in Bayesian network learning described in Section 2.1. In the problem, we are given a set V of nodes and a *parent set weight* $w_v(P)$ for all $v \in V$ and $P \subseteq V \setminus \{v\}$, and the task is to sample DAGs from the set $\mathcal{D}_V$ of all DAGs on V such that the probability of outcome $G \in \mathcal{D}_V$ is proportional to

$$w(G) = \prod_{v \in V} w_v(G_v),$$

where G_v denotes the set of parent nodes of v in G. Because the input consists of $2^{n-1}n$ weight values and each of them affects the result, we have a lower bound $\Omega(2^n)$ for the running time of any exact algorithm. Because $|\mathcal{D}_V| \geq n!$, obtaining an exponential time complexity bound is nontrivial.

Tian and He [83] gave a dynamic programming algorithm for the related problem of computing the normalizing constant $\sum_{G' \in \mathcal{D}_V} w(G')$ in $\tilde{O}(3^n)$ time. Here the $\tilde{O}$ notation suppresses logarithmic factors, that is, $g(n) = \tilde{O}(f(n))$ if there exists $k \geq 0$ such that $g(n) = O(f(n) \log^k f(n))$. While in Section 3.2 we successfully used a single run of a similar dynamic programming algorithm to create a polynomial-time sampler for linear extensions, we cannot directly replicate the idea in this case, because the sums in the recurrences used by the dynamic programming algorithm have both negative and positive terms.

In the remainder of this chapter, we present the contributions of Paper IV to the problem. We begin by describing the central idea of splitting the sampling process into two phases, in which the *root layering* of the DAG is sampled first and only then the DAG itself. Based on this, we present

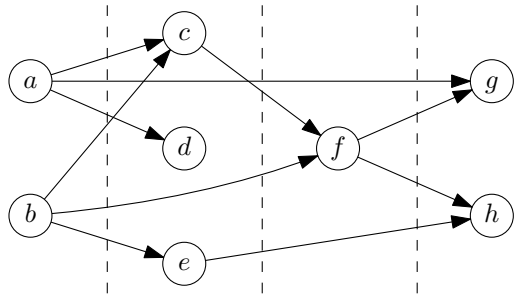


Figure 4.1: A DAG with root layering $(\{a, b\}, \{c, d, e\}, \{f\}, \{g, h\})$. The set of DAGs with this root layering consists of the directed graphs in which each node has parents only in the preceding layers and each node that is not in the first layer has at least one parent in the adjacent previous layer.

a preprocessing scheme that runs in $\tilde{O}(4^n)$ time and allows us to sample DAGs from the distribution in polynomial time. After this, we give an alternative preprocessing scheme that uses inclusion–exclusion recurrences inspired by those of Tian and He [83] to speed up preprocessing time to $\tilde{O}(3^n)$ while still enabling sampling in $\tilde{O}(2^n)$ time. Then we conclude the chapter by adapting the algorithms to the symmetric case in which $w_v(P)$ depends only on $|P|$, resulting in a sampler with a polynomial running time.

4.1 Root layerings

We define that a node $v \in V$ is a *root* of DAG $G = (V, E)$ if $G_v = \emptyset$. The set of roots of G is denoted by $\rho(G)$. The *root layering* of G is defined as the ordered partition $(R_i)_{i=1}^{l}$ of V we get by repeatedly removing the roots of the DAG until all the nodes have been removed. Figure 4.1 shows an example of the root layering of a DAG. To sample a DAG from the modular distribution, we use a two-phase algorithm in which we first sample the root layering of the DAG from its marginal distribution, and then the DAG conditionally on the sampled root layering. This can be seen as a generalization of the algorithm due to Kuipers and Moffa [55], which used this two-phase approach for sampling DAGs from the uniform distribution.

In this two-phase algorithm, most of the difficulty is in the first phase, because in the second phase, the fact that the root layering $(R_i)_{i=1}^{l}$ is fixed implies that the choices of parent sets G_v for nodes $v \in V$ are mutually independent. For each $1 \leq i \leq l$ and $v \in R_i$, we sample the parent set G_v from the set $\mathcal{C}(R_{i-1}, \bigcup_{j=1}^{i-1} R_j)$, where we define that $R_0 = 0$ and

$$\mathcal{C}(R, T) = \{S \subseteq T \,:\, S \cap R \neq \emptyset \text{ or } R = \emptyset\}.$$

The parent set has to be a member of $\mathcal{C}(R_{i-1}, \bigcup_{j=1}^{i-1} R_j)$, because otherwise node v would be in some other layer of the root layering. The probability weight of outcome G_v is given by the parent set weight $w_v(G_v)$.

We implement the first phase of the algorithm by sampling the root layering $(R_i)_{i=1}^{l}$ iteratively, starting from the first layer. After we have sampled R_j for all $j < i$, we sample the next layer R_i conditional on the choices of the previous layers. It turns out that the correct conditional probability for outcome $R_i \subseteq U := V \setminus \bigcup_{j=1}^{i-1} R_j$ is proportional to

$$f(R_i, U) \prod_{v \in R_i} \hat{w}_v(R_{i-1}, V \setminus U), \tag{4.1}$$

where $\hat{w}_v(R, T) = \sum_{S \in \mathcal{C}(R,T)} w_v(S)$, and for all $R \subseteq U$ we define that

$$\begin{aligned} \mathcal{D}_V(R, U) &= \{G \in \mathcal{D}_V \,:\, \rho(G) \supseteq V \setminus (U \setminus R), \rho(G[U]) = R\}, \\ f(R, U) &= \sum_{G \in \mathcal{D}_V(R,U)} \prod_{v \in U \setminus R} w_v(G_v). \end{aligned} \tag{4.2}$$

The high level idea on why the formula (4.1) gives us the conditional distribution over R_i is that it essentially sums over the weights of all the DAGs that are compatible with the chosen prefix $(R_j)_{j=1}^{i}$ of the root layering. Fixing the prefix makes the parent sets of the nodes in $R_i \cup (V \setminus U)$ mutually independent. Each $\hat{w}_v$-factor accounts for the weights of the possible choices of the parent set of $v \in R_i$. We could also similarly account for the parent sets of nodes in $V \setminus U$, but the resulting factor would not depend on the choice of R_i, and thus it can be omitted. The factor $f(R_i, U)$ sums over the possible DAG structures and accounts for the parent set weights of the nodes in $U \setminus R$. As the parent sets of these nodes are independent of the parent sets of the rest of the nodes, we may simply enforce that the nodes in $V \setminus (U \setminus R)$ have no parents, which means that the summation is over the DAGs in $\mathcal{D}_V(R_i, U)$.

If we have precomputed all the values of $\hat{w}_v$ and f, then a straightforward implementation of the two-phase sampling algorithm will run in $\tilde{O}(2^n)$ time per sample. To precompute $\hat{w}_v(R, T)$ for all $v \in V$ and $R \subseteq T \subseteq V$ in $\tilde{O}(3^n)$ time, we first compute it in the cases where $|R| \leq 1$ directly by using the summation in the definition, and then for each case in which $|R| \geq 2$, we pick an element $x \in R$ and use the recurrence

$$\hat{w}_v(R, T) = \hat{w}_v(\{x\}, T) + \hat{w}_v(R \setminus \{x\}, T) \setminus \{x\}).$$

We can precompute $f(R, U)$ for all $\emptyset \neq R \subseteq U \subseteq V$ using the recurrence

$$f(R, U) = \sum_{\emptyset \neq R' \subseteq U \setminus R} f(R', U \setminus R) \prod_{v \in R'} \hat{w}_v(R, V \setminus (U \setminus R)) \tag{4.3}$$

with the base case $f(U,U) = 1$. The recurrence considers every possible set of roots R' for $G[U \setminus R]$, and in each case uses the same idea as (4.1) to compute the sum of weights over the DAGs. As there are $O(4^n)$ triplets (R', R, U) we need to consider, the precomputation runs in $\tilde{O}(4^n)$ time.

Within this precomputation time budget of $\tilde{O}(4^n)$, we can actually create structures that allow us to sample DAGs in polynomial time. This is done by preprocessing a structure for sampling $R_i \subseteq U$ weighted by (4.1) for any $U \subseteq V$ and $R_{i-1} \subseteq V \setminus U$, and another structure for sampling $G_v \in \mathcal{C}(R,T)$ weighted by $w_v(G_v)$ for any $v \in V$ and $R \subseteq T \subseteq V \setminus \{v\}$.

4.2 Inclusion–exclusion recurrences

We will now give an alternative algorithm for precomputing $f(R,U)$ for all $\emptyset \neq R \subseteq U \subseteq V$ that improves the time complexity bound to $\tilde{O}(3^n)$, although with the drawback of deteriorating the numerical stability. To this end, we define for all $R \subseteq X \subseteq U \subseteq V$ the set

$$\bar{\mathcal{D}}_V(R,X,U) = \{G \in \mathcal{D}_V \,:\, \rho(G) \supseteq V \setminus (U \setminus R), \rho(G[U]) \supseteq X\}.$$

By using the inclusion–exclusion principle, we can transform (4.2) into

$$f(R,U) = \sum_{R \subseteq X \subseteq U} (-1)^{|X \setminus R|} \sum_{G \in \bar{\mathcal{D}}_V(R,X,U)} \prod_{v \in U \setminus R} w_v(G_v).$$

As $\bar{\mathcal{D}}_V(R,X,U)$ consists of exactly those DAGs that can be obtained from a DAG in $\bar{\mathcal{D}}_V(\emptyset,\emptyset,U \setminus X)$ by independently replacing the empty parent set of each node $v \in X \setminus R$ by an arbitrary subset of $V \setminus U$, we get that

$$f(R,U) = \sum_{R \subseteq X \subseteq U} (-1)^{|X \setminus R|} g(U \setminus X) \prod_{v \in X \setminus R} \hat{w}_v(\emptyset, V \setminus U), \tag{4.4}$$

where we define for all $U \subseteq V$

$$g(U) = \sum_{G \in \bar{\mathcal{D}}_V(\emptyset,\emptyset,U)} \prod_{v \in U} w_v(G_v).$$

Precomputing all the values of f given the values of g using a straightforward implementation of the formula (4.4) runs in $\tilde{O}(4^n)$ time. However, we can optimize this to $\tilde{O}(3^n)$ as follows. We consider a fixed $U \subseteq V$ at a time, and invert the first argument. This means that we want to compute the value $f_U(S) = f(U \setminus S, U)$ for all $S \subseteq U$. If we make the substitutions $X = U \setminus Y$ and $R = U \setminus S$ to the formula (4.4), we get that

$$f_U(S) = \sum_{Y \subseteq S} (-1)^{|S \setminus Y|} g(Y) \prod_{v \in S \setminus Y} \hat{w}_v(\emptyset, V \setminus U)$$

This is the *subset convolution* $\sum_{Y \subseteq S} g(Y)h(S \setminus Y)$ of functions g and h, where $h(Z) = (-1)^{|Z|} \prod_{v \in Z} \hat{w}_v(\emptyset, V \setminus U)$, and thus we can compute $f_U(S)$ for all $S \subseteq U$ in $\tilde{O}(2^{|U|})$ time by using the fast subset convolution algorithm due to Björklund et al. [6]. By repeating this process for all $U \subseteq V$, we get all the values of f in $\tilde{O}(3^n)$ time. In Section 3.1 of Paper IV, we use the special structure of h to optimize the convolution further, but the time complexity in the sense of $\tilde{O}$-notation remains the same.

We still have the task of computing the value $g(U)$ for all $U \subseteq V$ remaining. In the case $U \neq \emptyset$ it holds that $\bar{\mathcal{D}}_V(\emptyset, \emptyset, U) = \bigcup_{v \in U} \bar{\mathcal{D}}_V(\emptyset, \{v\}, U)$, and thus by applying the inclusion–exclusion principle, we get that

$$g(U) = \sum_{\emptyset \neq R \subseteq U} (-1)^{|R|+1} \sum_{G \in \bar{\mathcal{D}}_V(\emptyset, R, U)} \prod_{v \in U} w_v(G_v).$$

Similarly to (4.4), we see that the DAGs in $\bar{\mathcal{D}}_V(\emptyset, R, U)$ are obtained from the DAGs in $\bar{\mathcal{D}}_V(\emptyset, \emptyset, U \setminus R)$ by choosing a subset of $V \setminus U$ as the parent set for each node in R, and thus we get the recurrence

$$g(U) = \sum_{\emptyset \neq R \subseteq U} (-1)^{|R|+1} g(U \setminus R) \prod_{v \in R} \hat{w}_v(\emptyset, V \setminus U)$$

with base case $g(\emptyset) = 1$. A straightforward implementation of this recurrence computes all the values in $\tilde{O}(3^n)$ time.

In theory, this $\tilde{O}(3^n)$-time preprocessing algorithm is much faster than the $\tilde{O}(4^n)$-time preprocessing algorithm. However, it turns out that in practice, using the $\tilde{O}(4^n)$-time recurrence (4.3) is actually faster. To get an idea on why this happens, we need to look at the time complexities in the sense of O-notation instead of $\tilde{O}$-notation. Let us make the following standard assumption about the machine word, which is the native number type on which we can run arithmetic operations in $O(1)$ time: we assume that it is large enough to fit a single parent weight $w_v(P)$ or an index to the input array, that is, a $\Theta(n)$-bit integer.

If we carry out all the computations using exact integers, then the time complexity of the preprocessing step using the inclusion–exclusion recurrences is $O(3^n n^2)$. One of the $O(n)$-factors in the time complexity comes from the fact that the exact representation of some of the numbers in the computation will require $O(n)$ machine words. While typically algorithms like this are implemented using floating point numbers such that the precision of all the values is truncated to $O(1)$ machine words, in this case we cannot do this, because the subtractions in the inclusion–exclusion formulas might lead to *catastrophic cancellation*. This kind of cancellation occurs in cases where the two operands for a subtraction operation are very close

together, and thus the relative error of the result is much larger than that of the operands. This problem only occurs with the inclusion–exclusion recurrences, as the other formulas only use additions and multiplications of nonnegative numbers. Thus we can use truncated floating point numbers to implement preprocessing based on the recurrence (4.3) in $O(4^n)$ time.

In practice, we can only handle instances where $n < 20$, because the time complexities of the algorithms are bounded from below by $\Omega(3^n)$. In this range $4^n < 3^n n^2$, and thus the algorithm with time complexity $O(4^n)$ is likely to be faster than the algorithm with time complexity $O(3^n n^2)$, as we know that the constant factors hidden by the O-notation are moderate. For this reason, we implemented the $O(4^n)$-time preprocessing algorithm based on (4.3) for the experiments of Paper IV. The largest case the implementation could handle had $n = 15$ elements and took two minutes. In the case $n = 16$, the algorithm ran out of memory, as the space complexity is $\Theta(3^n n)$. By trading speed for lower memory consumption by storing $\hat{w}(R, T)$ only for the cases where $|R| \leq 1$, we were able to handle cases with $n = 17$ in five hours.

Remark. The definitions given in Paper IV for the sets $\mathcal{G}(R, U)$ and $\bar{\mathcal{G}}(X, U)$ that correspond to $\mathcal{D}_V(R, U)$ and $\bar{\mathcal{D}}_V(R, X, U)$, respectively, were errorneously missing the constraint $\rho(G) \supseteq R$ for the DAGs G. We have corrected the error in this chapter. The error concerned only the correctness proof; the algorithm itself was already correct and remains unchanged.

4.3 Symmetric case

Let us consider the special case of *symmetric* modular distributions, in which there exists a function $w_* : \{0, 1, \ldots, n-1\} \to [0, \infty)$ such that each parent set weight $w_v(P)$ is given by $w_*(|P|)$. We show that in this case, our algorithms can be adapted such that they run in time that is polynomial in n. The input for the algorithm consists of $w_*(p)$ for all $0 \leq p \leq n-1$.

In the symmetric case, all the nodes are interchangeable. Thus to adapt the preprocessing algorithm, we can simply replace sets by their sizes, which means that we replace $f(R, U)$ by $f(r, u)$ and $\hat{w}_v(R, T)$ by $\hat{w}_*(r, t)$, where r, u, and t are integers. We base the preprocessing algorithm on the $\tilde{O}(4^n)$-time preprocessing algorithm for the general case due to its better numerical stability. We compute $f(r, u)$ for all $1 \leq r \leq u \leq n$ using the following adaptation of (4.3):

$$f(r, u) = \sum_{r'=1}^{u-r} \binom{u-r}{r'} f(r', u-r)(\hat{w}_*(r, n-u+r))^{r'}, \tag{4.5}$$

and $\hat{w}_*(r,t)$ for all $0 \leq r \leq t \leq n$ by reduction to the case $r \leq 1$ using the recurrence $\hat{w}_*(r,t) = \hat{w}_*(1,t) + \hat{w}(r-1,t-1)$. In the sampling phase, we iteratively sample the sizes of the root layers $(r_i)_{i=1}^{l}$ from distributions given by the adaptation of formula (4.1). After this, we randomly distribute the set V among them to form the root layering $(R_i)_{i=1}^{l}$ such that $|R_i| = r_i$ for all $1 \leq i \leq l$, and proceed similarly to the general-case algorithm.

We can prove that on average, floating point numbers of $O(1)$ machine words suffice for this algorithm. Based on this, a straightforward analysis shows that preprocessing runs in $O(n^3)$ time, after which we can obtain samples in $O(n^2)$ time. However, we can improve the preprocessing time complexity by using a result due to Liskovets [59], which states that in a DAG sampled from the uniform distribution, the number of root nodes is small with a very high probability. Based on this result, Kuipers and Moffa [55] observed that for uniform DAG sampling, we can simply disregard the cases where a root layer is larger than a certain threshold, as they have negligible probability. In Section 4.4 of Paper IV, we show that this optimization can be used in the general symmetric case without sacrificing the exactness of the algorithm. The performance of the generalization depends on the following distance to uniformity:

$$A = \log_2 \big(\max\{w_*(a)/w_*(b) \,:\, a, b \in \{0, 1, \ldots, n-1\}\big).$$

If at least one weight $w_*(p)$ is zero, we define that $A = \infty$. We obtain an improved preprocessing time complexity bound $O(n^2 \min\{A+1, n\})$ by setting $f(r,u) = 0$ for all $r > \ell$ for some threshold ℓ, which means that we can also truncate the sum in the recurrence relation (4.5) to the first ℓ terms. To keep the algorithm exact, we maintain bounds on the induced error and if in the sampling phase we notice that the precision is not sufficient, we rerun the preprocessing with increased ℓ and possibly with increased numerical accuracy. However, this recomputation happens with only a very small probability, and thus the expected running time of the sampling phase remains $O(n^2)$.

In the case of the uniform distribution over DAGs, which can be represented as a symmetric modular distribution by setting all the weights $w_*(p)$ to 1, it holds that $A = 0$. Thus both preprocessing and sampling runs in $O(n^2)$ time for the uniform distribution, which improves on the previous best known bound $O(n^3)$ for exact sampling [55]. Unlike the exponential-time algorithms for the general case, this algorithm can sample DAGs of thousands of nodes in a matter of seconds.

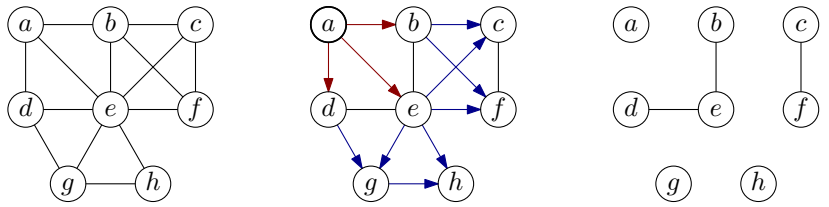
Chapter 5

Moral acyclic orientations

In Section 2.2, we considered causal learning, and arrived at the problems of counting and sampling DAGs in the Markov equivalence class corresponding to a given essential graph $G = (V, E)$. The equivalence class contains exactly the DAGs that can be obtained from the partially directed graph G by directing the undirected edges without creating directed cycles or additional immoralities, that is, triplets (a, v, b) of distinct nodes such that there are directed edges $a \to v \leftarrow b$ but no edge between a and b.

It turns out that the problems reduce to the special case where G is an *undirected connected chordal graph* (UCCG). In this case, G does not contain any immoralities to begin with, and thus in the problem we count or sample *moral acyclic orientations* (MAOs) of the UCCG. In the reduction from the general case, we simply remove the directed edges of the essential graph and consider each component separately. The special structure [1] of the essential graph ensures that each component is a UCCG and we may orient it independently from the rest of the graph [28]. This means that the number of DAGs in the equivalence class is obtained as the product of the numbers of MAOs of each UCCG component.

In the remainder of this chapter, we will introduce the contributions of Paper V to the problem of counting and sampling MAOs. We begin by describing the recursive algorithm due to He et al. [35] and the improved algorithm we obtain using dynamic programming. After this, we present a new dynamic programming algorithm based on the clique tree decomposition of the chordal graph. We conclude the chapter by experimentally comparing the performance of the algorithms.



(a) UCCG (b) rooted essential graph (c) subproblems

Figure 5.1: To count the MAOs of UCCG (a), the recursive root picking algorithm [35] counts the MAOs separately for every possible root node. In the case of root node a, the algorithm forms the rooted essential graph (b) by orienting the edges (in red) incident to a away from it, followed by the other orientations (in blue) that follow by the morality condition. After removing all the directed edges, the resulting graph (c) consists of UCCG components. The algorithm recursively counts the MAOs of each of them, multiplying the results to obtain the number of MAOs with root a.

5.1 Recursive root picking

The recursive MAO counting algorithm due to He et al. [35] is based on the observation that each MAO of a UCCG G has exactly one root node, that is, a node with no parents. This holds because any DAG has at least one root node and having two root nodes in a MAO leads to a contradiction as follows. Consider the shortest path $(x_0, x_1, \ldots, x_m)$ in G between two distinct roots in MAO D. Because x_0 and x_m are roots, we have the orientations $x_0 \to x_1$ and $x_{m-1} \leftarrow x_m$. Thus we also have the orientations $x_{i-1} \to x_i \leftarrow x_{i+1}$ for some $1 \leq i < m$. As the path is shortest, there is no edge connecting x_{i-1} and x_{i+1}, and thus (x_{i-1}, x_i, x_{i+1}) is an immorality in D, which is a contradiction with the assumption that D is a MAO.

This observation makes it possible for the algorithm to count the total number of MAOs of G by counting for each $v \in V$ the number of MAOs with root v and adding up the results. To count MAOs with root v, the algorithm counts the MAOs of the *rooted essential graph* $G^{(v)}$, which is obtained from G by incorporating the edge orientations that are fixed by the choice of root. Constructing $G^{(v)}$ from given UCCG G works by first orienting the edges incident to v away from it, and then as long as there is an induced subgraph $a \to b - c$ in the graph, orienting the undirected edge to direction $b \to c$ as implied by the morality condition [34, 35]. Figure 5.1 shows an example of the construction.

If we remove all the directed edges from the rooted essential graph $G^{(v)}$, the components of the resulting graph are UCCGs and each MAO

of $G^{(v)}$ is obtained by independently choosing a MAO for each UCCG component [35]. Similarly to the reduction from general essential graphs to UCCGs, we obtain the number of MAOs of $G^{(v)}$ as the product of the numbers of MAOs of each UCCG component. By recursively solving each subproblem, the algorithm eventually reduces all the subproblems to the base case where $|V| = 1$.

In the worst case where G is a complete graph, all the nodes apart from the chosen root always form a single subproblem, and thus the number of recursive calls is $O(n!)$. Each recursive call considers $O(n)$ roots and for each of them constructs the rooted essential graph in $O(n^3)$ time, which yields a bound $O(n!n^4)$ for the total time complexity. However, for practical instances the bound is very loose, as the UCCG is typically subdivided into multiple small subproblems when removing the directed edges of the rooted essential graph. He et al. [35] further speed up the algorithm by using a formula to count the MAOs in special cases where the UCCG is close to a tree or a complete graph.

In Paper V, we observe that the recursive algorithm often solves the same subproblem multiple times. We can avoid this repeated computation using memoization, by storing the result of each subproblem in a lookup table after computing it for the first time. We also prove that each subproblem is actually an induced subgraph $G[U]$ of the original UCCG G for some $U \subseteq V$. As there are 2^n such sets U, this dynamic programming algorithm achieves time complexity bound $O(2^n n^4)$. Due to the use of the lookup table, the space complexity increases from $O(n^3)$ to $O(2^n)$. These bounds are often loose in practice, since not all subproblems $U \subseteq V$ are considered.

We also note that by tracing back the recursive steps made by the counting algorithm, we can sample MAOs of G uniformly at random. In this sampling method, we first sample the root $v \in V$ for the MAO weighted by the total number of MAOs with that root. The rooted essential graph $G^{(v)}$ then gives us some of the edge directions, and the subgraph consisting of the remaining undirected edges is oriented by recursively sampling a MAO for each UCCG component. We modify the counting algorithm such that after running it once, the resulting data structure can be used to sample MAOs of G in $O(|V| + |E|)$ time. The modified algorithm saves for each considered subproblem a data structure that enables sampling the root node in $O(1)$ time [89, 90] along with a list of references to the data structures of the resulting subproblems for each possible root. Saving these data structures does not increase the time complexity of the algorithm. However, the space complexity becomes equal to the time complexity.

5.2 Dynamic programming in clique tree

Connected chordal graphs are special in that they can be exactly represented as *clique trees* [7]. To represent the UCCG $G = (V, E)$, we use a clique tree $\mathcal{T} = (\mathcal{X}, L)$ in which the nodes $\mathcal{X} \subseteq 2^V$ are exactly the maximal cliques of G. We refer to nodes of $\mathcal{T}$ as *bags* to differentiate them from nodes of G. The structure of the tree satisfies the *running intersection property*: for each $v \in V$, the bags that contain v induce a connected subtree of $\mathcal{T}$. The UCCG G can be recovered from $\mathcal{T}$ because we have an edge between nodes $a, b \in V$ if and only if there exists a bag $X \in \mathcal{X}$ such that $a, b \in X$. It is known that a clique tree representing UCCG G can be constructed in $O(|V| + |E|)$ time [7].

In this section, we introduce the second main contribution of Paper V: a new algorithm for counting and sampling MAOs based on translating the MAOs of G into assignments $(\preceq_X)_{X \in \mathcal{X}}$ of linear orders to each bag of the clique tree $\mathcal{T}$. The algorithm is based on dynamic programming over subtrees of $\mathcal{T}$ in which we form the assignments by joining together compatible partial assignments $(\preceq_X)_{X \in \mathcal{X}'}$ for subtrees $\mathcal{T}[\mathcal{X}']$. Dynamic programming over the clique tree is a special case of the technique of dynamic programming over a tree decomposition, which is commonly used to solve hard problems on graphs in polynomial time in cases where the treewidth of the graph is bounded [16].

The linear order assignment $(\preceq_X)_{X \in \mathcal{X}}$ corresponding to the acyclic orientation $D = (V, A)$ of G is defined by

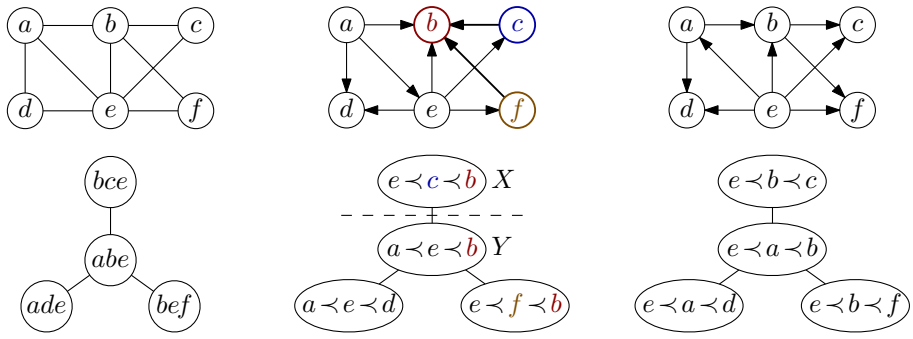
$$\preceq_X = \{(a, b) \in X^2 \,:\, a = b \text{ or } (a, b) \in A\}.$$

Because D is acyclic and each bag is a clique in G, it holds that $\preceq_X$ is a linear order for all $X \in \mathcal{X}$. We characterize the assignments $(\preceq_X)_{X \in \mathcal{X}}$ that correspond to a MAO of G using the following conditions for all pairs of adjacent bags X and Y in $\mathcal{T}$:

(1) The orders $\preceq_X$ and $\preceq_Y$ are compatible, that is, for all $a, b \in X \cap Y$ it holds that $a \preceq_X b$ if and only if $a \preceq_Y b$.

(2) Let $\mathcal{X}_X \ni X$ and $\mathcal{X}_Y \ni Y$ be the components of $\mathcal{T}$ remaining after disconnecting the edge between X and Y. Now for all $v \in X \cap Y$, at most one of the following holds:

- There exists $X' \in \mathcal{X}_X$ such that $v \in X'$ and for some $a \in X' \setminus Y$ it holds that $a \prec_{X'} v$.
- There exists $Y' \in \mathcal{X}_Y$ such that $v \in Y'$ and for some $b \in Y' \setminus X$ it holds that $b \prec_{Y'} v$.



(a) UCCG (b) immoral acyclic orientation (c) MAO

Figure 5.2: Consider the UCCG and the corresponding clique tree shown in (a). The assignment of linear orders to the bags shown in (b) corresponds to an acyclic orientation of the UCCG, because it satisfies condition (1): all pairs of adjacent bags agree on the ordering of their common nodes. However, the assignment does not satisfy condition (2): if we choose the adjacent bags X and Y as shown in the figure, then the node $b \in X \cap Y$ has predecessors c and f that only appear in bags on different sides of the edge between X and Y. This means that the corresponding acyclic orientation contains the immorality (c, b, f). The assignment in (c) corresponds to a MAO, because it satisfies both conditions (1) and (2).

Condition (1) ensures that the acyclic orientation $D = (V, A)$ can be recovered from the assignment by setting $A = \bigcup_{X \in \mathcal{X}} \prec_X$. Condition (2) ensures that D does not contain an immorality (a, v, b). The examples of Figure 5.2 illustrate this characterization.

Each subtree of $\mathcal{T}$ considered by the dynamic programming algorithm has a designated root bag, and the algorithm only joins assignments to subtrees that are disjoint and have root bags that are adjacent in $\mathcal{T}$; one of the root bags then becomes the root of the joined subtree. Because of this, the algorithm can lump together different assignments as long as they have the same *essential configuration*, which concerns nodes at the root bag. This kind of local information is sufficient for checking the compatibility of joined assignments, because the running intersection property of clique trees allowed us to formulate the conditions (1) and (2) such that they only consider nodes in two adjacent bags.

For the subtree induced by $\mathcal{X}' \subseteq \mathcal{X}$ with root bag $R \in \mathcal{X}'$, the essential configuration of assignment $(\preceq_X)_{X \in \mathcal{X}'}$ consists of the linear order $\preceq_R$ along with one additional bit of information for every node $v \in R$, which signifies whether there exists $X \in \mathcal{X}'$ and $u \in X \setminus R$ such that $v \in X$ and $u \prec_X v$.

The algorithm works by constructing for each subtree a table which gives the number of assignments for each possible essential configuration. If k is the size of the largest clique in G, then $|X| \leq k$ for all $X \in \mathcal{X}$ and thus each table has at most $2^k k!$ entries. We devised an algorithm for joining two tables in $O(k!2^k k^2)$ time using the fast subset convolution [6]. As there are $O(n)$ bags in total, the counting algorithm runs in $O(k!2^k k^2 n)$ time. The fact that the bound has the form $O(f(k)n^c)$ proves that the problem is *fixed parameter tractable* [16] when parameterized by k.

Similarly to the recursive root picking algorithm, this counting algorithm can be modified to produce a data structure for sampling MAOs in $O(|V|+|E|)$ time. However, the time complexity of the algorithm increases to $O(3^k k!k^2 n)$, because the way the fast subset convolution uses subtraction renders it unusable for sampling. When implementing the algorithm, we noticed that using a straightforward algorithm for the subset convolution instead of the fast subset convolution is in practice faster also in the case of counting MAOs, because the tables are typically sparse and the fast subset convolution does not benefit from sparsity.

Remark. Independently of our work, Ghassami et al. [27] gave another clique tree-based algorithm for counting and sampling MAOs with time complexity bound $O(n^{\Delta+2})$, where Δ is the maximum degree in G. Our result is stronger in the sense that k is a less restrictive parameter than Δ, because it always holds that $k \leq \Delta + 1$. In addition, their algorithm only proves that the problem is in the class of *slice-wise polynomial* (XP) problems [16] when parameterized by Δ; it does not prove fixed parameter tractability (FPT), because the parameter Δ affects the degree of the polynomial in the time complexity. From our FPT result for parameter k, we obtain as a corollary that the problem is FPT also for parameter Δ.

5.3 Experiments

In Paper V, we experimentally compared the performance of the recursive root picking algorithm, its dynamic programming variant and the clique tree-based dynamic programming algorithm. To generate the UCCGs for the experiments, we used the same method that He et al. [35] used in the experimental evaluation of the recursive root picking algorithm. The method starts from a tree of n nodes and repeatedly chooses a pair of nodes uniformly at random and adds an edge between them if the resulting graph is still chordal. This process is continued until the UCCG has rn edges, where r is a given density parameter.

Figure 5.3 shows the running times of the algorithms for different numbers of nodes n and densities r. In the sparse case $r = 3$, we see that dynamic programming in the clique tree is the fastest algorithm in large instances, which is expected, because sparse instances are unlikely to contain large cliques. The speedup obtained by augmenting the recursive root picking algorithm with dynamic programming is very small in the sparse case, but when moving to the dense case $r = 6$, the speedup factor exceeds an order of magnitude and the algorithm is clearly the fastest.

Remark. Based on an exhaustive enumeration of all the Markov equivalence classes of DAGs with at most 10 nodes, Gillispie and Perlman [28] observed that the average size of an equivalence class appears to converge to a constant that is less than four as the number of nodes increases. A more recent result due to Katz et al. [49] suggests that also in the practically relevant case of sparse DAGs, a vast majority of the equivalence classes are small. Thus from the viewpoint of average case analysis, the problem of counting and sampling DAGs in a Markov equivalence class is easily solved by a simple enumeration of all the DAGs in the equivalence class. Despite this, in the wide class of instances where the essential graph contains large undirected components, the large size of the equivalence class makes the enumeration method unfeasible. More involved methods, such as the ones considered in this chapter, are required to solve these cases.

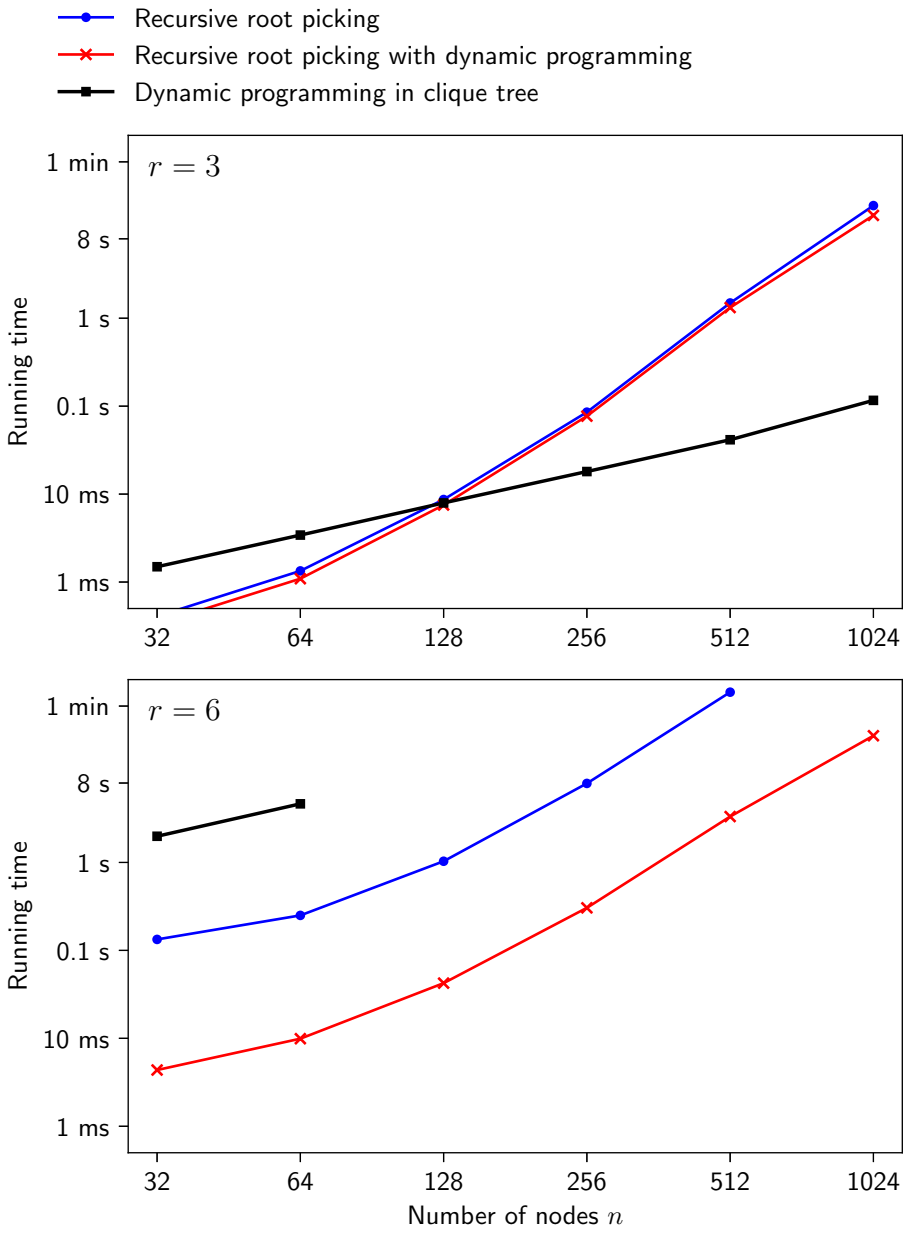


Figure 5.3: The median running times of the algorithms as functions of the number of nodes n for different density parameters r. When $r = 6$ and $n \geq 128$, the running times of the clique tree-based dynamic programming algorithm are missing because the memory consumption of the algorithm exceeds the 4 gigabyte limit.

Chapter 6

Discussion

In this thesis, we studied counting and sampling of directed acyclic structures under a variety of constraints, all of which stem from structure learning in Bayesian networks. Our contributions to these problems are in the form of algorithms whose results are guaranteed to be either exact or within given error bounds with high probability. We did not restrict our attention to algorithms for which we can prove good time complexity bounds, placing emphasis on the practical running time on typical instances instead.

A large part of the thesis was dedicated to sampling and counting linear extensions using methods based on Markov chains. Taking an experimental approach, we discovered in Paper I that alternatives to the Karzanov–Khachiyan chain often perform better in practice despite their lack of convergence bounds. In Paper II, we were able to obtain practical speedups to the state-of-the-art polynomial-time approximate counting scheme by switching to an alternative chain and exploiting structural features in the posets. Then in Paper III we presented a novel Markov chain-based exact sampler, which was successful in speeding up approximate linear extension counting in the relaxation Tootsie Pop algorithm. Bounding the time complexity of the sampler remains an open problem. Our experiments suggest that the time complexity is polynomial. However, the experiments only consider a limited set of benchmark instances—we did not systematically look for worst cases.

In Paper II we also proposed the adaptive relaxation Monte Carlo method (ARMC), which augments the dynamic programming algorithm for counting linear extensions with sampling-based approximation, allowing it to handle larger instances. The method makes the instance easier by relaxing the constraints and then solves it exactly using the dynamic programming algorithm. Sampling is used to estimate the skew in the result introduced by the relaxation. This approach could potentially be used in

other constrained counting problems as well. The ARMC and the relaxation Tootsie Pop algorithm together constitute the new state of the art in approximate linear extension counting. The algorithms are similar to each other in the sense that they are both based on sampling elements of relaxations of the original instance, however, they excel in clearly separated ranges of input size. Thus it is natural to ask whether combining ideas from both algorithms could result in further speedups.

For the problem of sampling DAGs from modular distributions, we presented the first exact exponential-time algorithm in Paper IV. The time complexity of the algorithm is $\tilde{O}(3^n)$, where n is the number of nodes. Since any exact algorithm for the general case of the problem has to take into account all of the $\Omega(2^n n)$ input weights, the time complexity will necessarily be at least exponential. One challenge for future work is designing an exact algorithm that improves the time complexity, as there is potentially still room for improvement. However, a more important question is whether we can use special properties of the distribution to bridge the gap between exact methods, which are limited to small instances, and Markov chain-based methods, which can often handle larger instances in practice but offer no guarantees on the quality of the result. The polynomial-time sampling algorithm we presented for the special case of symmetric modular distributions contains one idea towards this direction, as the time complexity of the algorithm adapts to the uniformity of the given weights.

Finally, in Paper V we studied the problem of counting and sampling DAGs from a given Markov equivalence class. We improved the state-of-the-art recursive algorithm by using dynamic programming, which resulted in speedups of multiple orders of magnitude for dense instances. We also presented another dynamic programming algorithm based on the clique tree structure of each chordal undirected component of the essential graph. The algorithm runs in linear time if the size of the maximum clique is bounded, and we also found it to be the fastest algorithm for sparse instances in practice. While we proved that the problem is fixed parameter tractable when parameterized by the clique size, it remains an open question whether the problem is #P-hard when the clique size is unbounded.

In summary, we successfully improved the performance of algorithms for each of the computational problems in terms of practical running time, and in some cases also in the form of improved asymptotic time complexity bounds. Our approach of tailoring each algorithm to the specific problem led to very different algorithms for each of the three considered problems. Despite this, some of the methods we discovered should be useful also more generally in other constrained counting and sampling problems.

References

[1] S. A. Andersson, D. Madigan, and M. D. Perlman. A characterization of Markov equivalence classes for acyclic digraphs. *The Annals of Statistics*, 25(2):505–541, 1997.

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

[3] J. Banks, S. Garrabrant, M. Huber, and A. Perizzolo. Using TPA to count linear extensions. *Journal of Discrete Algorithms*, 51:1–11, 2018.

[4] T. Batu, L. Fortnow, R. Rubinfeld, W. D. Smith, and P. White. Testing closeness of discrete distributions. *Journal of the ACM*, pages 4:1–4:25, 2013.

[5] N. Bhatnagar, A. Bogdanov, and E. Mossel. The computational complexity of estimating MCMC convergence time. In *Proceedings of the 15th International Workshop on Randomization and Computation (RANDOM)*, pages 424–435, 2011.

[6] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: Fast subset convolution. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC)*, pages 67–74, 2007.

[7] J. R. S. Blair and B. Peyton. *Graph Theory and Sparse Matrix Computation*, chapter An Introduction to Chordal Graphs and Clique Trees, pages 1–29. Springer, 1993.

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

[9] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201(1):81–88, 1999.

[10] W. Buntine. Theory refinement on Bayesian networks. In *Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 52–60, 1991.

[11] S. Chakraborty and K. S. Meel. On testing of uniform samplers. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7777–7784, 2019.

[12] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, pages 60:1–60:6, 2014.

[13] D. M. Chickering. *Learning from Data: Artificial Intelligence and Statistics V*, chapter Learning Bayesian Networks is NP-Complete, pages 121–130. Springer, 1996.

[14] G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.

[15] J. Cussens and M. Bartlett. Advances in Bayesian network learning using integer programming. *Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 182–191, 2013.

[16] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[17] P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for Monte Carlo estimation. *SIAM Journal on Computing*, 29(5):1484–1496, 2000.

[18] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[19] 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.

[20] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[21] M. Dyer, A. 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.

[22] F. Eberhardt, C. Glymour, and R. Scheines. On the number of experiments sufficient and in the worst case necessary to identify all causal relations among n variables. In *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 178–184, 2005.

[23] N. Friedman and D. Koller. Being Bayesian about network structure. a Bayesian approach to structure discovery in Bayesian networks. *Machine Learning*, 50(1–2):95–125, 2003.

[24] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[25] D. Geiger and D. Heckerman. Learning Gaussian networks. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 235–243, 1994.

[26] A. Ghassami, S. Salehkaleybar, N. Kiyavash, and E. Bareinboim. Budgeted experiment design for causal structure learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 1724–1733, 2018.

[27] A. Ghassami, S. Salehkaleybar, N. Kiyavash, and K. Zhang. Counting and sampling from Markov equivalent DAGs using clique trees. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 3664–3671, 2019.

[28] S. B. Gillispie and M. D. Perlman. The size distribution for Markov equivalence classes of acyclic digraph models. *Artificial Intelligence*, 141(1–2):137–155, 2002.

[29] P. Giudici and R. Castelo. Improving Markov chain Monte Carlo model search for data mining. *Machine Learning*, 50(1–2):127–158, 2003.

[30] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting: A new strategy for obtaining good bounds. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, pages 54–61, 2006.

[31] M. H. Maathuis, M. Kalisch, and P. Bühlmann. Estimating high-dimensional intervention effects from observation data. *The Annals of Statistics*, 37(6A):3133–3164, 2009.

[32] A. Hauser and P. Bühlmann. Two optimal strategies for active learning of causal models from interventional data. *International Journal of Approximate Reasoning*, 55(4):926–939, 2014.

[33] R. He, J. Tian, and H. Wu. Structure learning in Bayesian networks of a moderate size by efficient sampling. *Journal of Machine Learning Research*, 17(101):1–54, 2016.

[34] Y. He and Z. Geng. Active learning of causal networks with intervention experiments and optimal designs. *Journal of Machine Learning Research*, 9:2523–2547, 2008.

[35] Y. He, J. Jia, and B. Yu. Counting and exploring sizes of Markov equivalence classes of directed acyclic graphs. *Journal of Machine Learning Research*, 16(79):2589–2609, 2015.

[36] D. Heckerman. A Bayesian approach to learning causal networks. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 285–295, 1995.

[37] D. Hsu, A. Kontorovich, and C. Szepesvári. Mixing time estimation in reversible Markov chains from a single sample path. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, pages 1459–1467, 2015.

[38] M. Huber. Fast perfect sampling from linear extensions. *Discrete Mathematics*, 306(4):420–428, 2006.

[39] M. Huber. Near-linear time simulation of linear extensions of a height-2 poset with bounded interaction. *Chicago Journal of Theoretical Computer Science*, 2014.

[40] M. Huber. *Perfect Simulation*. CRC Press, 2016.

[41] M. Huber and S. Schott. Using TPA for Bayesian inference. *Bayesian Statistics*, 9:257–282, 2010.

[42] A. Hyttinen, F. Eberhardt, and P. O. Hoyer. Experiment selection for causal discovery. *Journal of Machine Learning Research*, 14:3041–3071, 2013.

[43] M. Järvisalo, D. L. Berre, O. Roussel, and L. Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.

[44] M. Jerrum and A. Sinclair. *Approximation Algorithms for NP-hard Problems*, chapter The Markov Chain Monte Carlo Method: An Approach to Approximate Counting and Integration, pages 482–520. PWS, 1997.

[45] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM*, 51(4):671–697, 2004.

[46] M. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[47] K. Kangas, T. Hankala, T. Niinimäki, and M. Koivisto. Counting linear extensions of sparse posets. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 603–609, 2016.

[48] A. Karzanov and L. Khachiyan. On the conductance of order Markov chains. *Order*, 8(1):7–15, 1991.

[49] D. Katz, K. Shanmugam, C. Squires, and C. Uhler. Size of interventional Markov equivalence classes in random DAG models. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics, AISTATS*, pages 3234–3243, 2019.

[50] B. Kenig and B. Kimelfeld. Approximate inference of outcomes in probabilistic elections. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 2061–2068, 2019.

[51] M. Koivisto and K. Sood. Exact Bayesian structure discovery in Bayesian networks. *Journal of Machine Learning Research*, 5:549–573, 2004.

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

[53] K. B. Korb and A. E. Nicholson. *Bayesian Artificial Intelligence*. CRC Press, 2010.

[54] J. Kuck, T. Dao, S. Zhao, B. Bartan, A. Sabharwal, and S. Ermon. Adaptive hashing for model counting. In *Proceedings of the 35th Conference on Uncertainty in Artificial Intelligence (UAI)*, 2019.

[55] J. Kuipers and G. Moffa. Uniform random generation of large acyclic digraphs. *Statistics and Computing*, 25(2):227–242, 2015.

[56] J. Kuipers and G. Moffa. Partition MCMC for inference on acyclic digraphs. *Journal of the American Statistical Association*, 112(517):282–299, 2017.

[57] J. Kuipers, G. Moffa, and D. Heckerman. Addendum on the scoring of Gaussian directed acyclic graphical models. *The Annals of Statistics*, 42(4):1689–1691, 2014.

[58] D. Levin, Y. Peres, and E. Wilmer. *Markov Chains and Mixing Times*. AMS, 2009.

[59] V. Liskovets. On the number of maximal vertices of a random acyclic digraph. *Theory of Probability and Its Applications*, 20(2):401–409, 1975.

[60] T. Lukasiewicz, M. V. Martinez, and G. I. Simari. Probabilistic preference logic networks. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*, pages 561–566, 2014.

[61] D. Madigan and J. York. Bayesian graphical models for discrete data. *International Statistical Review*, 63:215–232, 1995.

[62] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 161–168, 2000.

[63] C. Meek. Causal inference and causal explanation with background knowledge. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 403–410, 1995.

[64] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[65] R. H. Möhring. *Algorithms and Order*, chapter Computationally Tractable Classes of Ordered Sets, pages 105–193. Kluwer Academic Publishers, 1989.

[66] C. Muise, J. C. Beck, and S. A. McIlraith. Optimal partial-order plan relaxation via MaxSAT. *Journal of Artificial Intelligence Research*, 57:113–149, 2016.

[67] C. Muise, S. A. McIlraith, J. C. Beck, and E. I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Advances in Artificial Intelligence*, pages 356–361. Springer, 2012.

[68] J. Naor, M. Naor, and A. Schaffer. Fast parallel algorithms for chordal graphs. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 355–364, 1987.

[69] T. Niinimäki, P. Parviainen, and M. Koivisto. Structure discovery in Bayesian networks by sampling partial orders. *Journal of Machine Learning Research*, 17(57):1–47, 2016.

[70] S. Ott, S. Imoto, and S. Miyano. Finding optimal models for small gene networks. *Pacific Symposium on Biocomputing*, 9:557–567, 2004.

[71] J. D. Park and A. Darwiche. Approximating MAP using local search. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 403–410, 2001.

[72] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2009.

[73] J. Pearl and A. Paz. *GRAPHOIDS: A Graph-based logic for reasoning about relevance relations*. University of California, 1985.

[74] J. G. Propp and D. B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures & Algorithms*, 9(1–2):223–252, 1996.

[75] K. Shanmugam, M. Kocaoglu, A. G. Dimakis, and S. Vishwanath. Learning causal graphs with small interventions. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, pages 3195–3203, 2015.

[76] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82(1):93–133, 1989.

[77] M. Soos and K. S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 1592–1599, 2019.

[78] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. MIT Press, 2001.

[79] R. P. Stanley. Acyclic orientations of graphs. *Discrete Mathematics*, 5(2):171–178, 1973.

[80] B. Steinsky. Asymptotic behaviour of the number of labelled essential acyclic digraphs and labelled chain graphs. *Graphs and Combinatorics*, 20(3):399–411, 2004.

[81] L. Stockmeyer. The complexity of approximate counting. In *Proceedings of the 15th ACM Symposium on Theory of Computing (STOC)*, pages 118–126, 1983.

[82] M. Thurley. sharpSAT – counting models with advanced component caching and implicit BCP. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 424–429, 2006.

[83] J. Tian and R. He. Computing posterior probabilities of structural features in Bayesian networks. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 538–547, 2009.

[84] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[85] L. G. Valiant and V. V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47(3):85–93, 1986.

[86] P. Valiant. Testing symmetric properties of distributions. In *Proceedings of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 383–392, 2008.

[87] P. van Beek and H. Hoffmann. Machine learning of Bayesian networks using constraint programming. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP)*, pages 429–445, 2015.

[88] T. Verma and J. Pearl. Equivalence and synthesis of causal models. In *Proceedings of the 6th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 255–270, 1990.

[89] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17:972–975, 1991.

[90] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3):253–256, 1977.

[91] D. B. Wilson. Mixing times of lozenge tiling and card shuffling Markov chains. *Annals of Applied Probability*, 14(1):274–325, 2004.

[92] C. Yuan and B. M. Malone. Learning optimal Bayesian networks: A shortest path perspective. *Journal of Artificial Intelligence Research*, 48:23–65, 2013.