Counting and Sampling Markov Equivalent Directed Acyclic Graphs

Topi Talvitie
Department of Computer Science
University of Helsinki
topi.talvitie@helsinki.fi

Mikko Koivisto
Department of Computer Science
University of Helsinki
mikko.koivisto@helsinki.fi

Abstract

Exploring directed acyclic graphs (DAGs) in a Markov equivalence class is pivotal to infer causal effects or to discover the causal DAG via appropriate interventional data. We consider counting and uniform sampling of DAGs that are Markov equivalent to a given DAG. These problems efficiently reduce to counting the moral acyclic orientations of a given undirected connected chordal graph on n vertices, for which we give two algorithms. Our first algorithm requires $O(2^n n^4)$ arithmetic operations, improving a previous super-exponential upper bound. The second requires $O\big(k!\,2^k k^2 n\big)$ operations, where k is the size of the largest clique in the graph; for bounded-degree graphs this bound is linear in n. After a single run, both algorithms enable uniform sampling from the equivalence class at a computational cost linear in the graph size. Empirical results indicate that our algorithms are superior to previously presented algorithms over a range of inputs; graphs with hundreds of vertices and thousands of edges are processed in a second on a desktop computer.

1 Introduction

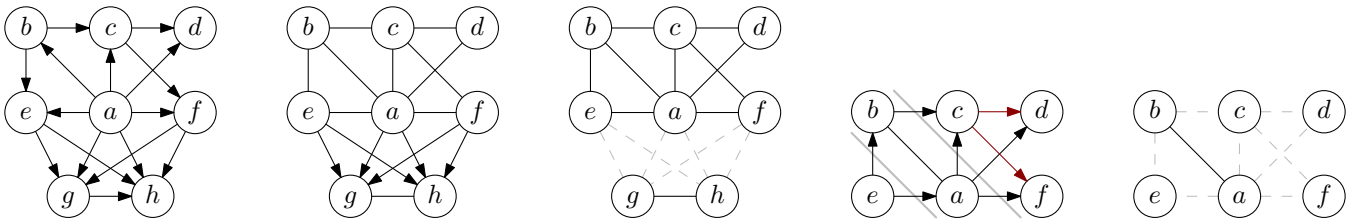
In causal discovery a key task is to learn a *directed acyclic graph (DAG)* on the variables of interest. What makes learning particularly challenging is that different DAGs can represent the same conditional independence relations among the variables; such DAGs are *Markov equivalent*. Every Markov equivalence class can be represented uniquely by an *essential graph* (a.k.a. *completed partial DAG*), in which an edge is undirected unless the direction is unique in the class (Andersson, Madigan, and Perlman 1997). Purely observational data are generally sufficient for identifying a unique essential graph, but not singling out a DAG.

That said, the DAGs within a given equivalence class are well worth exploring, for instance, to estimate causal effects between pairs of variables (Maathuis, Kalisch, and Bühlmann 2009) or to direct some of the undirected edges in the essential graph based on interventional data (Ghassami et al. 2018). Such tasks call for efficient algorithms for generating random DAGs from a given equivalence class, as well as, for studying the related combinatorial problem of computing the size of the equivalence class. It is well known that these problems immediately (and efficiently) reduce to their restricted variants where the essential graph is an *undirected connected chordal graph (UCCG)*; see Gillispie and Perlman (2002) and Section 2. Then the DAGs in the equivalence class correspond to what we, in this paper, call *moral acyclic orientations (MAOs)*.

Two approaches to count and sample MAOs have been investigated recently. For counting, He, Jia, and Yu (2015) discovered a recurrence: by guessing the unique source vertex of a MAO (i.e., branching on the options), the orientations of some edges get fixed, leaving some number of smaller disjoint UCCGs; if a so-encountered UCCG is an "almost clique," then a fast special treatment is given. He and Yu (2016) enhanced the algorithm by extracting at every level of the recurrence a so-called core graph of the UCCG in question. Ghassami, Salehkaleybar, and Kiyavash (2018) observed that the basic recurrence admits a complexity bound $n^{\Delta+O(1)}$ on graphs of maximum degree Δ, and readily allows for uniform sampling.[1] While these algorithms can handle sparse graphs with hundreds of vertices, their best known worst-case complexity bound is $O(n!)$. Bernstein and Tetali (2017) took a different approach and considered sampling MAOs from an almost uniform distribution. They showed that a simple edge-flip Markov chain mixes in time that is exponential in the worst case but polynomial under certain restrictions on the input UCCG.

In this paper we give new exact algorithms. Our algorithms (i) yield improved worst-case complexity bounds and (ii) also run faster in practice, by several orders of magnitude in some cases. We begin in Section 2 by introducing more formally the problem and the basic recurrence. Then, in Section 3, we make a simple observation that gives us a *dynamic programming (DP)* variant of the recursive algorithm with a complexity bound of $O\big(2^n n^4\big)$. In Section 4 we derive a better bound for sparse graphs. Specifically, we assume that the largest clique has size k, potentially much smaller than n, and give another DP algorithm that requires $O\big(k!\,2^k k^2 n\big)$ operations. Using standard routines, both algorithms can be turned into uniform samplers. In Section 5 we report on empirical results and draw conclusions as to which algorithm is the fastest for what type of instances. We end in Section 6 by discussing open questions and directions for future research.

[1] Our complexity bounds assume that two $O(n \log n)$-bit integers can be added and multiplied with one (unit-cost) operation.



(a) DAG (b) Essential graph (c) UCCGs (d) e-orientation (e) Subproblem

Figure 1: Illustration of some key concepts. (a) A DAG, (b) its essential graph, (c) the UCCG components of the essential graph. (d) The e-orientation of the larger UCCG. In addition to orienting the edges between vertices at different distances from e, we also orient $c \to d$ and $c \to f$ because otherwise we would get new immoralities $b \to c \leftarrow d$ and $b \to c \leftarrow f$, respectively. (e) The remaining subproblem after removing directed edges.

2 Preliminaries

We will need a number of concepts and terminology, some of which are rather standard in graph theory, and others that are more specific and developed in the literature on Markov equivalence classes (Andersson, Madigan, and Perlman 1997). This section also reviews the basic recurrence discovered by He, Jia, and Yu (2015).

Basic Graph Terminology

A graph is a pair $\mathcal{G} = (V, E)$ with a finite *vertex set* V and an *edge set* $E \subseteq V \times V \setminus \{vv : v \in V\}$. Between two vertices u, v there is an *undirected edge*, called *line* $u - v$, if $uv, vu \in E$, and a *directed edge*, called *arrow* $u \to v$, if $uv \in E$ but $vu \notin E$. The graph is *undirected* (*directed*) if all its edges are undirected (resp. directed).

A graph (S, F) is a *subgraph* of $\mathcal{G}$ and *contained* in $\mathcal{G}$ if $S \subseteq V$ and $F \subseteq E \cap (S \times S)$; it is *spanning* (*induced*) if the former (resp. latter) inclusion is an equality. We write $\mathcal{G}[S]$ for the subgraph of $\mathcal{G}$ induced by S.

A graph is a *directed cycle* (*undirected cycle*) of *length* ℓ if its vertices can be ordered as $v_0, v_1, \ldots, v_{\ell-1}$ such that its edges are $v_{i-1} \to v_i$ (resp. $v_{i-1} - v_i$) for $1 \leq i \leq \ell$, where $v_\ell = v_0$. A graph is an *immorality* (a.k.a. *v-structure*) if its vertices can be labeled as u, v, w such that the edge set is $\{uv, wv\}$, i.e., the graph is $u \to v \leftarrow w$ ("the parents of v are unmarried"). A graph is a *flag* if its vertices can be labeled as u, v, w such that the edge set is $\{uv, vw, wv\}$, i.e., the graph is $u \to v - w$.

A graph is *acyclic* if it contains no directed cycle of length larger than two, *moral* if no induced subgraph is an immorality, and *chordal* if it is undirected and no subset of four or more vertices induces an undirected cycle.

A graph $\mathcal{G}$ is a *chain graph* if it does not contain a semi-directed cycle, that is, a subgraph that is a directed cycle with at least one arrow $u \to v$ such that $u - v$ is a line in $\mathcal{G}$. A *chain component (CC)* of a chain graph is a connected component of the undirected graph obtained by removing all directed edges.

Markov Equivalence Classes

Let $\mathcal{D} = (V, A)$ be a DAG. The *essential graph* of $\mathcal{D}$ is the graph $\mathcal{E} = (V, E)$, where E is the union of the edge sets of the DAGs that are Markov equivalent to $\mathcal{D}$ (Andersson, Madigan, and Perlman 1997). Thus the essential graph is a unique representation of the Markov equivalence class. The essential graph can be constructed efficiently (Meek 1995).

It is known that an essential graph is a chain graph where each CC is an *undirected and connected chordal graph (UCCG)* (Andersson, Madigan, and Perlman 1997). The vertex sets of the CCs, $V_1, V_2, \ldots, V_c$, partition V. Denoting by $\mu(\mathcal{E})$ the size of the Markov equivalence class represented by $\mathcal{E}$, we thus have that (Gillispie and Perlman 2002)

$$\mu(\mathcal{E}) = \prod_{i=1}^{c} \mu(\mathcal{E}[V_i])\,. \tag{1}$$

In other words, the equivalence class represented by $\mathcal{E}$ and a Cartesian product of the equivalence classes represented by each chain component are in one-to-one correspondence.

Figure 1(a–c) shows an example of a DAG, the corresponding essential graph, and its UCCG components.

Moral Acyclic Orientations

The above observations motivate focusing on the problems of counting and sampling DAGs that belong to the equivalence class represented by a given UCCG $\mathcal{C}$. Each such DAG is an *orientation* of $\mathcal{C}$, that is, a spanning subgraph that contains exactly one of the arrows $u \to v$ and $v \to u$ for every line $u - v$ in $\mathcal{C}$; furthermore, the orientation must be moral. It is easy to see that the *moral acyclic orientations (MAOs)* are exactly the members of the equivalence class (He, Jia, and Yu 2015). Our key problem is thus the following:

#MAO
Input: An n-vertex UCCG $\mathcal{C} = (V, E)$.
Output: The number of MAOs of $\mathcal{C}$, i.e., $\mu(\mathcal{C})$.

The Sum–Product Recurrence

It is known that every MAO of a UCCG has a unique source vertex. He, Jia, and Yu (2015) observed that the MAOs with a fixed source vertex s agree on the orientation of some edges, while the remaining edges again leave an undirected chordal subgraph. They call the union of the MAOs an *s-rooted essential graph*; we use the term *s-orientation* for brevity and because the graph may not be an essential graph.

Definition 1. Let $\mathcal{C}$ be a UCCG with vertex s. The *s-orientation* of $\mathcal{C}$, denoted by $\mathcal{C}^s$, is the union of all MAOs of $\mathcal{C}$ whose (unique) source vertex is s.

Since an s-orientation is a chain graph whose chain components are UCCGs (He, Jia, and Yu 2015, Thm. 7),

$$\mu(\mathcal{C}) = \sum_{s \in V} \mu(\mathcal{C}^s)\,,$$

where each $\mu(\mathcal{C}^s)$ admits again the product rule (1).

To construct an s-orientation of a UCCG $\mathcal{C}$, He, Jia, and Yu (2015) give the following algorithm (sligthly modified):

Algorithm *Orient*: $(\mathcal{C}, s) \mapsto \mathcal{C}^s$

O1 Create $\mathcal{C}'$ by orienting each line $u - v$ of $\mathcal{C}$ as $u \to v$ if s is closer to u than v in $\mathcal{C}$ (in the shortest-path distance).

O2 As long as $\mathcal{C}'$ has a flag $u \to v - w$ as an induced subgraph, update $\mathcal{C}'$ by orienting $v - w$ as $v \to w$.

O3 Output $\mathcal{C}'$.

Figure 1(d–e) shows an example of an s-orientation and the resulting subproblems.

3 Dynamic Programming

We will make a simple but crucial observation, which shows that the number of distinct subproblems encountered by the sum–product recurrence is at most 2^n, where $n = |V|$ is the number of vertices of the input UCCG. We will do this by showing that every subproblem corresponds to an *induced* subgraph of the input UCCG.

Let us first consider a single step in the recurrence.

Lemma 1. *Let $\mathcal{C}$ be a UCCG with a vertex s. Let $\mathcal{I}$ be a chain component of $\mathcal{C}^s$. Then $\mathcal{I}$ is an induced subgraph of $\mathcal{C}$.*

Proof. We have that $\mathcal{I}$ is an induced subgraph of $\mathcal{C}^s$ and a UCCG (He, Jia, and Yu 2015, Thm. 7). Since $\mathcal{C}^s$ is a subgraph of $\mathcal{C}$ (obtained by removing some edges), it suffices to show that two vertices u and v are non-adjacent in $\mathcal{I}$ only if they are non-adjacent in $\mathcal{C}$. But this is immediate, as in $\mathcal{C}$ the two vertices are either non-adjacent or connected by a line, which is either unchanged or directed in $\mathcal{C}^s$. □

Encouraged by this observation we let $\mu_{\mathcal{C}}(S) := \mu(\mathcal{C}[S])$ for $S \subseteq V$. In particular, $\mu_{\mathcal{C}}(V)$ is the number of MAOs of $\mathcal{C}$. We next show that this function satisfies a sum–product recurrence analogous to the one for UCCGs.

Proposition 2. *Let $\mathcal{C}$ be a UCCG and S its vertex subset that induces an UCCG $\tilde{\mathcal{C}} := \mathcal{C}[S]$. We have that*

$$\mu_{\mathcal{C}}(S) = \sum_{s \in S} \prod_{T} \mu_{\mathcal{C}}(T)\,,$$

where T runs through the vertex sets of the CCs of $\tilde{\mathcal{C}}^s$, the s-orientation of $\mathcal{C}[S]$.

Proof. By the recurrence of He, Jia, and Yu (2015) we have that

$$\mu(\tilde{\mathcal{C}}) = \sum_{s \in S} \prod_{T} \mu(\tilde{\mathcal{C}}^s[T])\,,$$

where T ranges as in the statement. By Lemma 1, we have that $\tilde{\mathcal{C}}^s[T] = \tilde{\mathcal{C}}[T]$. It remains to observe that $\tilde{\mathcal{C}}[T] = \mathcal{C}[T]$, for $T \subseteq S$ (by basic properties of induced subgraphs). □

Proposition 2 suggests a dynamic programming algorithm that computes and stores the value $\mu_{\mathcal{C}}(S)$ using the recurrence at most once for each S. Because potentially only a small fraction of all subsets need be evaluated, we propose a top-down, recursive implementation with memoization:

Algorithm *MemoMao*: $\mathcal{C} \mapsto \mu(\mathcal{C})$

M1 Let $a[\cdot]$ be a storage indexed by $S \subseteq V$; initially $a[S]$ returns 1 if S is a singleton, and NULL otherwise.

M2 Output *Sum-Product*(V).

Function *Sum-Product*(S)

S1 If $a[S] \neq$ NULL, return $a[S]$; else let $a[S] \leftarrow 0$.

S2 For each vertex $s \in S$:

- Run *Orient* to get $\mathcal{C}'$, the s-orientation of $\mathcal{C}[S]$.
- Let $(S_i)_{i=1}^{c}$ be the vertex sets of the CCs of $\mathcal{C}'$.
- Let $a[S] \leftarrow a[S] + \prod_{i=1}^{c}$ *Sum-Product*(S_i).

S3 Return $a[S]$.

The main result of this section now follows.

Theorem 3. *The complexity of* #MAO *is* $O(2^n n^4)$.

Proof. Clearly it suffices to estimate the cost of step S2 of Function *Sum-Product*. Consider an arbitrary subset S of V. By step S1 we have that S2 is run at most once for each S. One S2 needs $O(|S|)$ additions and $O(|S|^2)$ multiplications.

To bound the number of other operations, required for constructing each s-orientation and its component UCCGs, consider a fixed $s \in S$. The lengths of the single-source shortest paths and thus step O1 of Algorithm *Orient* can be computed with $O(n^2)$ operations. Step O2 is seen to require $O(n^3)$ operations, by the following argument: for each line $v - w$ oriented as $v \to w$, it suffices to consider once the neighboring $O(n)$ lines of the form $w - x$ and test whether v and x are adjacent (by $v \to x$).

Thus $O(n^4)$ operations suffice for any fixed S, and $O(2^n n^4)$ operations in total. □

Remark 1 (Bit complexity)**.** The proof shows that other operations than additions and multiplications dominate the complexity bound, implying that the bit complexity of #MAO is at most $2^n n^4$ up to a factor logarithmic in n.

Remark 2 (Uniform sampling)**.** In order to support efficient uniform sampling of MAOs, for each encountered set S, we store the constructed s-orientations and associate each with the respective weight (i.e., the product) in the sum–product formula. To enable constant-time sampling of an s-orientation with a probability proportional to its weight, we use the alias method (Walker 1977; Vose 1991). Starting from $S = V$ and proceeding recursively into the component UCCGs, we can draw a random DAG in $O(|V| + |E|)$ time.

4 Using Tree Decomposition

We next present another DP algorithm that is fast if the maximal cliques of the input UCCG are small. The idea is to organize the cliques into a tree structure and tabulate the solutions of subproblems at each clique for configurations whose number is bounded by the clique size—this is the standard paradigm of tree decomposition based DP.

Cliques and Tree Decomposition

A *tree decomposition (TD)* of a graph (V, E) is a tree $\mathcal{T}$ where each *node* x is labelled by a *bag* $B_x \subseteq V$ such that (i) the endpoints of each edge $uv \in E$ occur in one of the bags, and (ii) for each vertex $v \in V$ the nodes whose bag contains v induce a non-empty connected subtree of $\mathcal{T}$. The *width* of the tree decomposition is size of the largest bag minus one.

Chordal graphs are special in that they admit a TD, called *clique tree*, whose bags are exactly the maximal cliques of the graph (Blair and Peyton 1993). Such a decomposition is optimal in the sense that no TD can have a smaller width. Furthermore, one can construct a clique tree in time linear in the graph size (Blair and Peyton 1993). Another useful property of a clique tree is that the underlying graph is obtained as its *intersection graph*, that is, by connecting two vertices by an edge if and only if they both appear in same bag. We will use this property to relate a MAO of $\mathcal{C}$ to a collection of linear orders on the bags of a TD of $\mathcal{C}$.

Definition 2. Let $\mathcal{C}$ be a UCCG and $\mathcal{T}$ its TD where the bag B_x of each node x is a clique of $\mathcal{C}$. Assign each node x of $\mathcal{T}$ a linear order $\prec_x$ on B_x. The assignment is *friendly* to $\mathcal{T}$ if the following holds for all adjacent nodes x, y:

(f1) The orders $\prec_x$ and $\prec_y$ are compatible, i.e., they are equal when restricted to $B_x \cap B_y$.

(f2) Let $X \ni x$ and $Y \ni y$ be the components of $\mathcal{T}$ that remain after disconnecting x and y. For any vertex $v \in B_x \cap B_y$, at most one of the following cases holds:

- For some node $z \in X$ the bag B_z contains v and another vertex u such that $u \prec_z v$ and $u \notin B_y$.
- For some node $z \in Y$ the bag B_z contains v and another vertex u such that $u \prec_z v$ and $u \notin B_x$.

Lemma 4. *The MAOs of $\mathcal{C}$ are in one-to-one correspondence with the linear order assignments friendly to $\mathcal{T}$.*

Proof. Consider first a MAO $\mathcal{A}$ of $\mathcal{C}$. For each node x of $\mathcal{T}$, let $\prec_x$ be the edge set of the induced subgraph $\mathcal{A}[B_x]$. Clearly, the assignment $(\prec_x)_x$ is unique. We need to show that conditions (f1) and (f2) are satisfied. The former is obvious. To check the latter condition, let x, y be adjacent nodes and X, Y as in the statement of the condition. Let $v \in B_x \cap B_y$. Now, if both cases of (f2) hold, then there exist vertices u and u' such $\mathcal{A}$ has the arrows $u \to v$ and $v \leftarrow u'$ even if u are u' are not adjacent in $\mathcal{C}$. Thus $\mathcal{A}$ contains an immorality, which is a contradiction, as desired.

For the other direction, let $(\prec_x)_x$ be a linear order assignment that is friendly to $\mathcal{T}$. Because of the property (ii) of the definition of TD, friendliness condition (f1) also holds for non-adjacent nodes x, y. Thus the linear orders determine a unique orientation $\mathcal{D}$ of $\mathcal{C}$: orient an edge uv as $u \to v$ if $u, v \in B_x$ and $u \prec_x v$ for some x. It remains to show that $\mathcal{D}$ is moral and acyclic. For the former, assume the contrary, i.e., there is an immorality $u \to v \leftarrow u'$. Then there must be adjacent nodes x, y such that $v \in B_x \cap B_y$, $u \in B_x \setminus B_y$, $u' \in B_y \setminus B_x$. This violates (f2), implying $\mathcal{D}$ is moral. To show acyclicity, assume the contrary: $\mathcal{D}$ contains a cycle. We can assume that $u \to v \to w \to u$ is the cycle, because if it is longer, by the chordality of $\mathcal{C}$ we know that there is a shortcut edge, and we can shorten the cycle. Let x, y, z be nodes of $\mathcal{T}$ such that $\{v, w\} \subset B_x$, $\{w, u\} \subset B_y$, $\{u, v\} \subset B_z$. Because $\mathcal{T}$ is a tree, the three unique simple paths from x to y, y to z, and z to x all pass through a common node c. By the definition of TD, we have that $\{u, v, w\} \subset B_c$. Thus it holds that $u \prec_c v \prec_c w \prec_c u$, which is a contradiction because $\prec_c$ is a linear order. Thus $\mathcal{D}$ is acyclic. □

The Algorithm

Lemma 4 allows us to formulate the DP algorithm in the space of (friendly) assignments of linear orders on the bags of a TD. For writing the DP steps, it will be convenient to assume that $\mathcal{T}$ is a *nice TD*, that is, when rooted at some node r, each node x is of one of the following types:

Leaf: no children.

Introduce: one child y, introduces a $v \in V \setminus B_x$, i.e., $B_y = B_x \cup \{v\}$.

Forget: one child y, forgets a $v \in B_x$, i.e., $B_y = B_x \setminus \{v\}$.

Join: two children y, z, with $B_x = B_y = B_z$.

On a high level, the algorithm employs DP over the subtrees of a nice TD. For each subtree rooted at node x, it computes a table F_x based on the tables F_y of the children y of x. Finally, $\mu(C)$ is extracted from the table F_r at the root.

Define the table F_x for each node x as follows. Let X be the set of nodes in the subtree rooted at x. For every linear order $\prec$ on B_x and subset $P \subseteq B_x$, define $F_x(\prec, P)$ as the number of assignments of linear orders $\prec_y$ on B_y, for $y \in X$, such that the following conditions hold:

- The assignment $(\prec_y)_{y \in X}$ is friendly to $\mathcal{T}[X]$.
- The linear order $\prec_x$ is equal to $\prec$.
- The set P consists of exactly those vertices in B_x that have a predecessor in some node of the subtree that is not present in B_x (a *lost predecessor*), i.e.,

$$P = \bigcup_{y \in X} \left\{ v \in B_y \cap B_x : u \prec_y v \text{ for some } u \in B_y \setminus B_x \right\}.$$

We will need P to satisfy the friendliness condition (f2).

By Lemma 4, $\mu(C)$ is the sum of the values in the table F_r. It remains to show how each table F_x is computed from the tables of the child nodes. In the description, we include derivations that prove the correctness of the algorithm, i.e., that the computed values, we denote by $F_x[\prec, P]$, equal the corresponding values, $F_x(\prec, P)$. See Figure 2 for an illustration of the algorithm.

Algorithm *TreeMao*: $\mathcal{C} \mapsto \mu(\mathcal{C})$

T1 Let $\mathcal{T}$ be a nice TD of $\mathcal{C}$, rooted at r.

T2 For each node x of $\mathcal{T}$, from the leaves towards the root, initialize the table $F_x[\cdot, \cdot]$ to zero, and then populate it:

- If x is a leaf node: Since the subtree has only one clique, B_x, all linear orders are possible and there are no lost predecessors. Thus we let $F_x[\prec, \emptyset] \leftarrow 1$ for all linear orders $\prec$ on B_x.

- If x introduces vertex v: Let y be the child of x, and consider all the elements $F_y(\prec, P)$. The only way to extend the assignment of linear orders from $X \setminus \{x\}$ to X, satisfying condition (f1), is by the linear order $\prec'$ obtained from $\prec$ by removing vertex v. Condition (f2) cannot be violated by this extension. After the removal of v, we need to add all of its successors in $\prec$ to the set of vertices with lost predecessors, and thus we increase $F_x[\prec', P \cup \{u \in B_x : v \prec u\} \setminus \{v\})]$ by $F_y[\prec, P]$.
- If x forgets vertex v: Let y be the child of x, and consider all the elements $F_y(\prec, P)$. To extend the assignment of linear orders from $X \setminus \{x\}$ to X, by condition (f1), we only need to consider the $|B_x|$ possible linear orders $\prec'$ obtained from $\prec$ by adding vertex v. To satisfy condition (f2), it must hold that there is no predecessor $u \in B_y$ such that $u \prec' v$ that is also in P. Thus we increase $F_x[\prec', P]$ by $F_y[\prec, P]$.
- If x joins nodes y and z: Let Y and Z the nodes sets of the subtrees rooted at y and z, respectively. Because $B_x = B_y = B_z$, every assignment of linear orders to X is obtained by combining assignments $(\prec_w)_{w \in Y}$ and $(\prec_w)_{w \in Z}$ where $\prec_y = \prec_z$, and letting $\prec_x \leftarrow \prec_y$. To satisfy condition (f2), we only need to ensure that the sets of vertices with lost predecessors from Y and Z are disjoint, and in the combined assignment, the set of lost predecessors is obtained as the disjoint union. Thus we let $F_x[\prec, P] \leftarrow \sum_{S \subseteq P} F_y[\prec, S] \cdot F_z[\prec, P \setminus S]$.

T3 Output $\sum_{\prec, P} F_r[\prec, P]$.

We are ready to prove the main result of this section: a parameterized complexity bound for #MAO:

Theorem 5. *TreeMao requires $O(k!\, 2^k k^2 n)$ operations, where k is the size of the largest clique in the input graph.*

Proof. Constructing a clique tree requires $O(nk^2)$ operations, linear in the size of the input. The clique tree has $O(n)$ nodes and width $k-1$. It is transformed in $O(nk^2)$ operations into a nice TD with $O(n)$ nodes and width $k-1$ (Bodlaender, Bonsma, and Lokshtanov 2013). Thus the complexity of step T1 is dominated by the bound being proven.

Each table F_x contains $O(k!\, 2^k)$ elements. It is easy to see that for leaf, introduce and forget nodes, the computations take $O(k^2)$ operations per element. For a join node x, we get the subtable $F_x[\prec, \cdot]$ from the subtables $F_y[\prec, \cdot]$ and $F_z[\prec, \cdot]$ as the so-called *subset convolution*. Using fast subset convolution it also costs k^2 operations per element (Björklund et al. 2007, Thm. 1). Now, because there are $O(n)$ nodes, we get the claimed complexity bound. □

Practical Tricks

The DP algorithm described above was designed to optimize the upper bound for the worst-case asymptotic complexity. In applications, however, we care more about the actual running time. We next describe some optimizations in our implementation that improve the running time in practice.

While a nice TD simplifies the description of the algorithm, for practical performance it is better to implement introducing, forgetting and joining in a single operation one a clique tree. This way the TD has the minimum number of nodes and the tables can be kept as small as possible by forgetting vertices that do not appear outside the subtree.

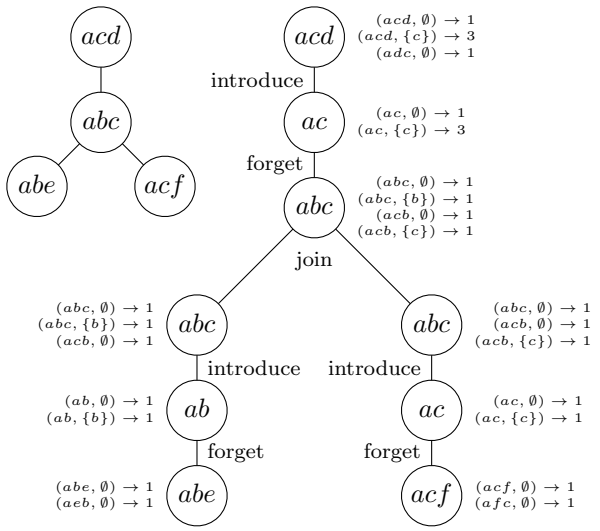


Figure 2: A tree decomposition and a nice tree decomposition for the larger UCCG in Figure 1(c). For each node, the portion of the dynamic programming table where a is the first vertex of the linear order is shown.

To obtain the complexity bound in Theorem 5, fast subset convolution was crucial in join operations. Our preliminary experiments showed that, in practice, the subtables $F_x[\prec, \cdot]$ are often so sparse that a simple enumeration of all pairs of nonzero elements in the child subtables $F_y[\prec, \cdot]$ and $F_z[\prec, \cdot]$ and filtering out the cases where the arguments (i.e., vertex subsets) intersect is faster than fast subset convolution. For this to work, we store the tables in a way that allows for sparsity, e.g., with balanced binary trees or hash tables.

In dense graphs some vertices tend to be symmetric: they appear in the bags of exactly the same nodes, at least in subtrees. Symmetric vertices are interchangeable in the DP tables, and we save space and computation time by storing the symmetric elements in the table only once.

Remark 3 (Sampling)**.** The computed tables enable efficient sampling of MAOs (cf. Remark 2). The only difficulty is that for join nodes, the counting algorithm uses fast subset convolution, which we cannot directly use for sampling the partition of the set P of lost predecessors for the two children. However, by enumerating all partitions, we obtain a sample in $O(2^k n)$ time. We can also sample in $O(|V|+|E|)$ time by a variant of the counting algorithm without fast subset convolution, yielding a complexity of $O(k!3^k k^2 n)$.

5 Experiments

We have evaluated four exact algorithms for #MAO:

- *MemoMao*: The DP algorithm based on the sum–product recurrence (Section 3).
- *TreeMao*: The DP algorithm based on tree decomposition (Section 4), with all the practical optimizations.

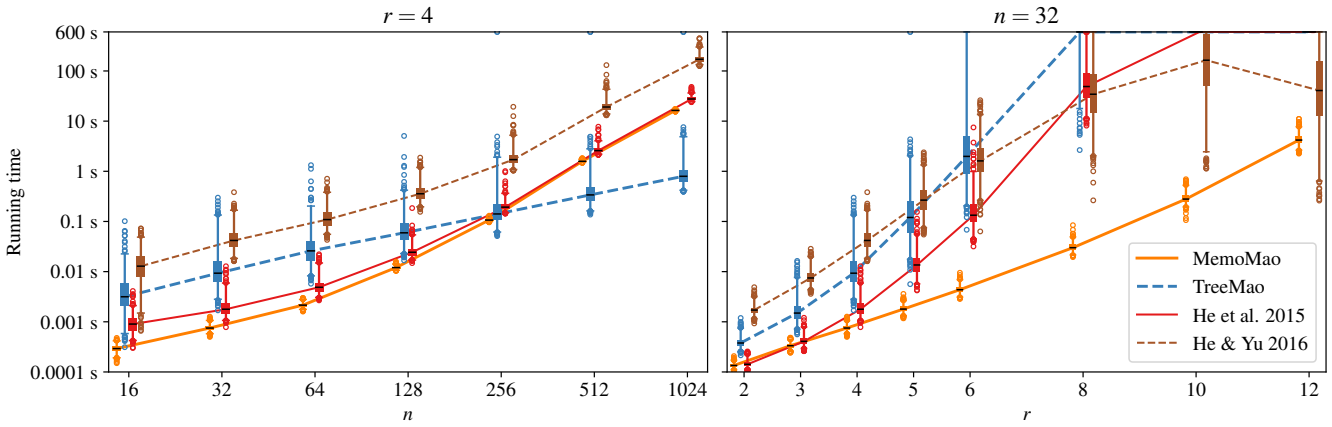


Figure 3: Running times of the four algorithms as functions of n and r on random UCCGs with n vertices and rn edges. The box is between percetiles 25% and 75%; the whiskers are at 1% and 99%.

- *He et al. 2015*: The algorithm based on the sum–product-recurrence without DP due to He, Jia and Yu (2015).
- *He & Yu 2016*: Improved version of the previous method based on core graphs (He and Yu 2016).

We implemented all four algorithms in C++[2], using only a single thread of execution and exact integer computations.

Extending on the experimental setup from He, Jia and Yu (2015), we run all the algorithms on randomly generated UCCGs with n vertices and rn edges, where $16 \leq n \leq 1024$ and $2 \leq r \leq 12$. The random generation works in two phases: First we generate a tree by starting from a single vertex and repeatedly adding a vertex as a neighbor of a randomly chosen vertex in the current tree until the tree has n vertices. After this, we add edges between pairs of vertices chosen uniformly at random, while on each step maintaining the chordality of the graph, until the graph has rn edges.

For each r and n, we generated 1000 UCCGs and ran each algorithm for every UCCG with time limit of 10 minutes and memory limit of 4 GB. Figure 4 shows the median running times of the algorithm, and Figure 3 shows the ranges of the running times for slices $r = 4$ and $n = 32$.

From the results we see that *MemoMao* is the fastest algorithm in most cases, but *TreeMao* is faster in sparse instances. In the median running times in Figure 4, *MemoMao* is faster than *TreeMao* when $n \leq 2^{r+4}$. The growth of the running time of *MemoMao* as a function of r is remarkably slow, and the improvement over *He et al. 2015* on which it is based is multiple orders of magnitude for large r. The random fluctuations in the running times of *MemoMao* are also very small. *He & Yu 2016* does not do particularly well except for very dense instances such as $n = 32, r = 12$.

6 Concluding Remarks

We have presented two new exact algorithms to count and uniformly sample DAGs that are Markov equivalent to a given DAG. We focused on the core case where the equivalence class is represented by an undirected, connected, and chordal essential graph. Common to both algorithms, *MemoMao* and *TreeMao*, is a systematic exploitation of the overlapping subproblems structure using dynamic programming. The experiments confirmed that the algorithms not only lower the known asymptotic worst-case complexity upper bounds, but also are faster than previously presented algorithms over a range of inputs.

To enhance the DP algorithms, one could seek more efficient ways to exploit symmetries, similar to those by He, Jia, and Yu (2015) and He and Yu (2016). Another direction is to implement hybrid methods, for instance, by letting *TreeMao* solve sparse subproblems encountered by *MemoMao*.

Some fundamental theoretical questions remain open. Can one solve #MAO in polynomial time; is it #**P**-hard? Does the problem admit a fully polynomial randomized approximation scheme (FPRAS)? We find these questions intriguing, since the related problem of counting acyclic orientations is polynomial time for chordal graphs, but hard in general (Vertigan and Welsh 1992), and an FPRAS is only known for some special cases (Bordewich 2004).

We may ask whether, in practice, computing the size of a Markov equivalence class is actually easy. This viewpoint is supported by the typical size of equivalence classes, i.e., the ratio of the number of all DAGs to the number of essential graphs on the same vertices. While the exact number of essential graphs is only known for small n (Steinsky 2013; Radhakrishnan, Solus, and Uhler 2017), we know that about every 14th DAG is in itself an essential graph and thus the only member of the equivalence class (Steinsky 2003; 2004). Even more strikingly, the average equivalence class size appears to be asymptotically less than four (Gillispie and Perlman 2002). As most graphs are dense, these ratios mainly concern properties of dense graphs—we do not know whether the ratios are significantly larger or smaller for sparse graphs, which are more relevant for practical applications (Gillispie and Perlman 2002, p. 152). Fortunately, in a sparse essential graph, we may expect that the chain components typically are either large and sparse, or small and potentially dense. In either case, one or the other of the presented two DP algorithms scales well.

[2] github.com/ttalvitie/count-mao

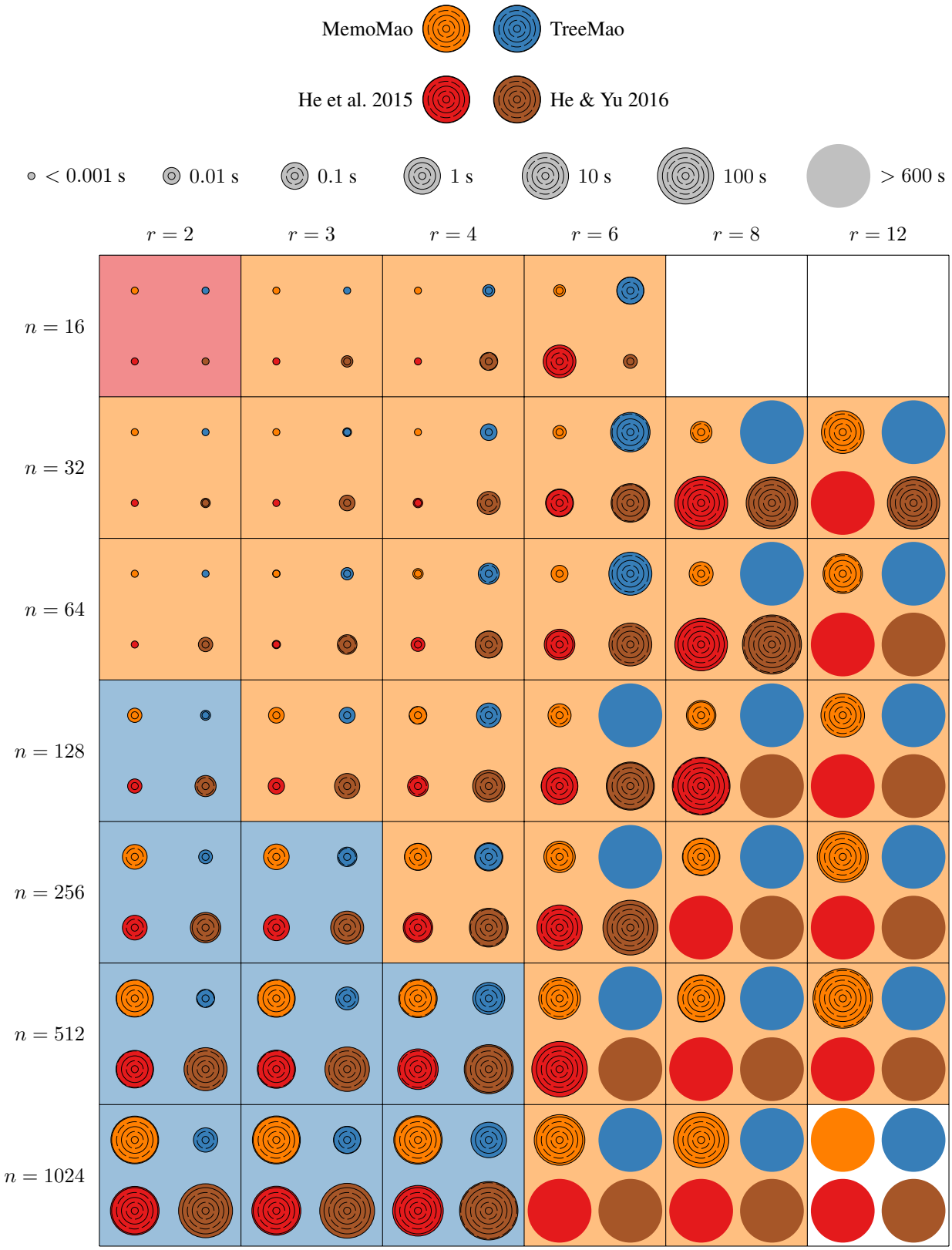


Figure 4: Median running times of the four algorithms for different n and r on random UCCGs with n vertices and rn edges. The radius of the circle indicates the running time. The background color of the cell marks the fastest algorithm in that cell. A missing circle outline indicates that the algorithm ran out time or memory.

Acknowledgments

This work was supported in part by the Academy of Finland, under Grant 276864.

References

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

Bernstein, M., and Tetali, P. 2017. On sampling graphical Markov models. *ArXiv e-prints 1705.09717*.

Björklund, A.; Husfeldt, T.; Kaski, P.; and Koivisto, M. 2007. Fourier meets Möbius: Fast subset convolution. In *Proceedings of the 39th ACM Symposium on Theory of Computing*, 67–74. New York, NY, USA: ACM.

Blair, J. R. S., and Peyton, B. 1993. An introduction to chordal graphs and clique trees. In George, A.; Gilbert, J. R.; and Liu, J. W. H., eds., *Graph Theory and Sparse Matrix Computation*, 1–29. New York, NY: Springer.

Bodlaender, H. L.; Bonsma, P. S.; and Lokshtanov, D. 2013. The fine details of fast dynamic programming over tree decompositions. In *Parameterized and Exact Computation – 8th International Symposium, IPEC 2013*, volume 8246 of *Lecture Notes in Computer Science*, 41–53. Springer.

Bordewich, M. 2004. Approximating the number of acyclic orientations for a class of sparse graphs. *Combinatorics, Probability and Computing* 13(1):1–16.

Ghassami, A.; Salehkaleybar, S.; Kiyavash, N.; and Bareinboim, E. 2018. Budgeted experiment design for causal structure learning. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, 1724–1733. PMLR.

Ghassami, A.; Salehkaleybar, S.; and Kiyavash, N. 2018. Counting and uniform sampling from Markov equivalent DAGs. *ArXiv e-prints 1802.01239*.

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

He, Y., and Yu, B. 2016. Formulas for counting the sizes of Markov equivalence classes of directed acyclic graphs. *ArXiv e-prints 1610.07921*.

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

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

Meek, C. 1995. Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, 403–410. Morgan Kaufmann.

Radhakrishnan, A.; Solus, L.; and Uhler, C. 2017. Counting Markov equivalence classes by number of immoralities. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence*. AUAI Press.

Steinsky, B. 2003. Enumeration of labelled chain graphs and labelled essential directed acyclic graphs. *Discrete Mathematics* 270(1):267–278.

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

Steinsky, B. 2013. Enumeration of labelled essential graphs. *Ars Combinatoria* 111:485–494.

Vertigan, D. L., and Welsh, D. J. A. 1992. The computational complexity of the Tutte plane: the bipartite case. *Combinatorics, Probability and Computing* 1(2):181–187.

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

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