Practical Algorithms for Orientations of Partially Directed Graphical Models

Malte Luttermann[*] LUTTERMANN@IFIS.UNI-LUEBECK.DE
Institute of Information Systems, University of Lübeck, Germany

Marcel Wienöbst[*] WIENOEBST@TCS.UNI-LUEBECK.DE
Institute for Theoretical Computer Science, University of Lübeck, Germany

Maciej Liśkiewicz LISKIEWI@TCS.UNI-LUEBECK.DE
Institute for Theoretical Computer Science, University of Lübeck, Germany

Editors: Mihaela van der Schaar, Dominik Janzing and Cheng Zhang

Abstract

In observational studies, the true causal model is typically unknown and needs to be estimated from available observational and limited experimental data. In such cases, the learned causal model is commonly represented as a partially directed acyclic graph (PDAG), which contains both directed and undirected edges indicating uncertainty of causal relations between random variables. The main focus of this paper is on the maximal orientation task, which, for a given PDAG, aims to orient the undirected edges maximally such that the resulting graph represents the same Markov equivalent DAGs as the input PDAG. This task is a subroutine used frequently in causal discovery, e. g., as the final step of the celebrated PC algorithm. Utilizing connections to the problem of finding a consistent DAG extension of a PDAG, we derive faster algorithms for computing the maximal orientation by proposing two novel approaches for extending PDAGs, both constructed with an emphasis on simplicity and practical effectiveness.

Keywords: Causal graphical models, Directed acyclic graphs, Markov equivalence, Consistent extension, Maximal orientation, Meek rules

1. Introduction

The development of probabilistic graphical models enables a mathematically sound language to handle uncertainty in a coherent and compact way (Spirtes et al., 2000; Pearl, 2009; Koller and Friedman, 2009; Elwert, 2013). They also provide scientists with intuitive tools for causal analysis and currently receive substantial attention in epidemiology, sociology and other disciplines. Moreover, the graphical modeling approach allows for the use of computational methods that have enabled significant progress towards automated causal inference and causal structure discovery.

Certainly, one of the most prominent graphical models is the *directed acyclic graph* (DAG), whose edges encode direct causal influences between the random variables of interest. In practice, however, the underlying true DAGs are unknown and from observational or limited experimental data they can only be inferred to a certain degree of uncertainty. In such cases, the learned causal model is given, typically, as a *partially directed acyclic graph* (PDAG) which contains both directed and undirected edges. Such a PDAG represents a class of *Markov equivalent* DAGs encoding the same statistical properties and its undirected edges indicate which directed edges may vary across DAGs of the class (Verma and Pearl, 1990; Meek, 1995; Andersson et al., 1997).

[*]Equal contribution.

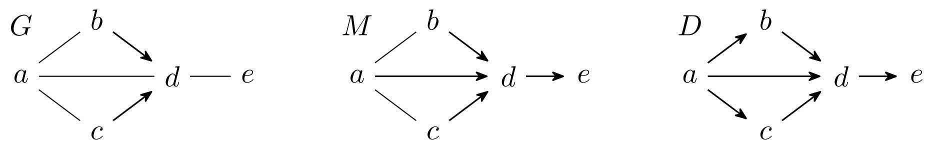


Figure 1: A PDAG G with a maximal orientation M. DAG D shows a consistent extension of G.

In this paper, we investigate *orientations* of PDAG models – primitive tasks used to solve more complex problems of causal analysis. Our goal is to provide simple and effective algorithms that improve performance of downstream tasks, e.g., in causal structure learning, active learning or causal effect estimation. The main focus of our study is on the *maximal orientation* problem, the goal of which is to orient a maximal number of undirected edges in a PDAG G such that the resulting graph, called *maximally oriented PDAG* (MPDAG), represents the same DAGs as G. It is well known that the MPDAG can be obtained from the PDAG by closing it under the celebrated Meek rules (Meek, 1995). For an example PDAG G and its MPDAG M, see Fig. 1.

The primary significance of the MPDAG model, including *completed PDAGs* (CPDAGs, also called *essential graphs*) (Andersson et al., 1997) is that it represents Markov equivalent DAGs in an elegant and convenient way. It is commonly used for solving many important problems as, e.g., counting and sampling Markov equivalent DAGs (He et al., 2015; Wienöbst et al., 2021), estimating causal effects (Maathuis et al., 2009; van der Zander and Liśkiewicz, 2016; Perković et al., 2017), or learning causal models (Chickering, 2002), where CPDAGs represent the states of the search space. Consequently, orienting a given PDAG maximally, is a frequently used primitive in causal inference and discovery, perhaps most prominently arising in constraint-based causal structure learning, as, e. g., the final step in the PC algorithm (Spirtes et al., 2000; Kalisch and Bühlman, 2007) and its modifications. Similarly, several score-based algorithms, e.g., (Pellet and Elisseeff, 2008) based on a generic feature-selection approach, rely on this task as well. Another important example are algorithms for active learning which, beyond observational, use experimental (interventional) data to resolve orientation ambiguities. In searching for an optimal strategy, typical algorithms iteratively construct *interventional essential graphs*, which are obtained by orienting subsequent PDAGs maximally (He and Geng, 2008; Hauser and Bühlmann, 2012; Shanmugam et al., 2015; Squires et al., 2020). Furthermore, methods enumerating all possible total effects or estimating bounds on the effects in a Markov equivalence class need a subroutine to compute the maximal orientations (Maathuis et al., 2009; Guo and Perković, 2021). Often, the maximal orientation task is performed not only once, but repeatedly, necessitating efficient algorithms for this task.

While the algorithms for computing maximal orientations used in practice resort to directly applying the Meek rules, the best theoretical methods are based on two other important primitives of causal analysis: (i) the *consistent extension* of a PDAG to a DAG and (ii) computing the CPDAG representation of the graphs Markov equivalent to a given DAG. This was already mentioned by Chickering (1995) and generalized to instances with background knowledge by Wienöbst et al. (2021). Using the clever algorithm of Chickering (1995), the second task (ii) can be solved in linear time. On the other hand, the computational complexity of the first problem, which is to orient all undirected edges such that no new v-structures arises, is significantly higher. Hence, developing effective practical methods for finding consistent extensions of PDAGs can be a key building block in the efficient computation of maximal orientations.

Previous Work. The study on extendability of PDAGs has been initiated by Verma and Pearl (1992), who provided a method to find a consistent DAG extension in time $O(n^4 m)$, where n denotes the number of nodes and m the number of edges. At the same time, Dor and Tarsi (1992) proposed a faster method of time complexity $O(n^4)$ which is conceptually simple, easy to implement, and widely used in practice so far. In 2021, Wienöbst et al. proposed a new algorithm for the problem which runs in time $O(n^3)$. Simultaneously, the authors showed that, under a computational intractability assumption, the cubic algorithm is optimal.

Despite its importance, relatively little attention has been paid to the algorithmic aspects of the maximal orientation problem so far. A commonly used approach relies on the direct, iterative application of the four Meek rules until none can be applied anymore. This approach is, however, computationally expensive, and its worst-case run time is $O(n^4 m)$, where n and m denote the number of vertices and edges of the graph. In 1995, Chickering proposed a sophisticated algorithm, avoiding the direct application of the Meek rules, which firstly extends a PDAG to a DAG and next computes the corresponding CPDAG directly from the DAG. Since a DAG can be transformed to the corresponding CPDAG in linear time, the total running time of the algorithm is dominated by the time needed for extension. This approach is used in the GES algorithm (Chickering, 2002) and in a modified way in the GIES algorithm (Hauser and Bühlmann, 2012), the latter however having the same worst-case complexity as the direct application of the Meek rules discussed above. More recently, Wienöbst et al. (2021) generalized both approaches, while maintaining the optimal time complexity of $O(n^3)$, to maximally orient a PDAG to an MPDAG.

Finally, in more structured cases, the maximal orientation task may be performed in linear-time $O(n+m)$. This occurs in the setting of active learning using *single-target* interventions (that is, only single variables can be manipulated at a time) and is shown in Wienöbst et al. (2022) (Theorem 5).

Our contributions. In this work, we focus on the general problem of maximally orienting a PDAG. The contributions of this paper are twofold: On the one hand, we give a practical evaluation of methods for the consistent extension problem, contributing two novel algorithms for this task, which are simple yet effective and build upon the algorithm by Dor and Tarsi (1992). On the other hand, we aim to illustrate the strengths of using consistent DAG extensions in the computation of the maximal orientation of a PDAG. While this approach has been proposed theoretically in (Chickering, 1995; Wienöbst et al., 2021), it has only been used in special cases and has not found widespread application in practice, evidenced by the fact that most software packages such as `pcalg` (Kalisch et al., 2012) and `causaldag` (Squires, 2018) rely on algorithms, which apply the Meek rules directly. Utilizing consistent DAG extensions instead, yields a better worst-case complexity and is superior in practice as we demonstrate in our experiments.

2. Preliminaries

We consider partially directed graphs $G = (V_G, A_G, E_G)$, in which the pairs of vertices $x, y \in V$ are connected by directed edges $u \rightarrow v$ (also called arcs, given in A_G) or undirected edges $u - v$ (given in E_G). We restrict ourselves to graphs, where at most one edge exists between any pair $x, y \in V_G$ and if there is such an edge, we call x and y adjacent, denoted by $x \sim_G y$. The arcs $u \rightarrow v$ and $u \leftarrow v$ have different *orientations* and *orienting* an undirected edge $u - v$ means replacing it with an arc. Vertex x is called a parent of y if $x \rightarrow y \in G$, a child of y if $x \leftarrow y \in G$ and a sibling of y if $x - y \in G$. The sets of parents, children and siblings of vertex v are denoted by $\mathrm{Pa}_G(v)$, $\mathrm{Ch}_G(v)$, $\mathrm{Si}_G(v)$. By $\mathrm{Ne}_G(v) = \mathrm{Pa}_G(v) \cup \mathrm{Ch}_G(v) \cup \mathrm{Si}_G(v)$ we denote the set of neighbors of v

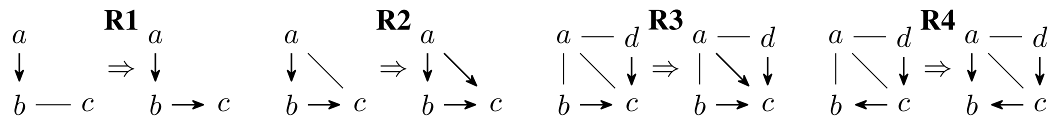


Figure 2: The four Meek rules that are used to characterize MPDAGs (Meek, 1995).

and the degree of v counts the number of its neighbors $|\mathrm{Ne}_G(v)|$. Edges, which have v as endpoint, are called *incident* to v. For a set $S \subseteq V_G$, the induced subgraph $G[S]$ contains all edges from G with both endpoints in S. We skip the subscript if G is clear from the context.

A partially directed graph is called acyclic (or partially directed acyclic graph, PDAG for short) if it does not contain a directed cycle, that is, a sequence of distinct vertices $(c_1, c_2, \ldots, c_k)$, with $k \geq 3$, and edges $c_i \rightarrow c_{i+1}$ for $i \in \{1, \ldots, k-1\}$, and $c_k \rightarrow c_1$. A PDAG without any undirected edges ($E_G = \emptyset$) is called a directed acyclic graph (DAG). There is a linear ordering (called *topological ordering*) of the vertices of every DAG such that $u \rightarrow v$ if u comes before v in the ordering.

Definition 1 *A DAG D is a* consistent extension *of a PDAG G if*

1. *D and G have the same vertex set and $\mathrm{Ne}_G(v) = \mathrm{Ne}_D(v)$ for all vertices v,*
2. *every directed edge in G is also in D, i. e., $A_G \subseteq A_D$,*
3. *for all edges $u - v$ in E_G, there is $u \rightarrow v$ in A_D or $v \leftarrow u$ in A_D, and*
4. *for all $u, v, w \in V$, the induced subgraph $u \rightarrow v \leftarrow w$ ($u \not\sim w$) is in G iff it is in D.*

The set of consistent extensions of a PDAG G is denoted by $[G]$. We remark that an induced subgraph of the form $u \rightarrow v \leftarrow w$ is also called a v-structure. Hence, every $D \in [G]$ has the same v-structures as G. If $[G] \neq \emptyset$, we call G *extendable*. Some undirected edges in a PDAG G might be oriented the same way in all of its consistent extensions. The graph where these undirected edges are replaced by the corresponding invariant arcs is called the *maximal orientation* of G, denoted by $\mathrm{MPDAG}(G)$, which stands for *maximally-oriented PDAG*. It is a well-known fact that $\mathrm{MPDAG}(G)$ can be computed by repeatedly applying the four Meek rules (Meek, 1995), which are shown in Fig. 2, to G until none applies anymore. In this work, we consider the following two fundamental computational problems for PDAGs:

Problem 1 EXT
Input: *PDAG G.*
Output: *$D \in [G]$ or $\perp$ if $[G] = \emptyset$.*

Problem 2 MAX-ORIENT
Input: *PDAG G.*
Output: $\mathrm{MPDAG}(G)$.

For MAX-ORIENT, we assume that G is extendable, else G has no causal interpretation and the definition of $\mathrm{MPDAG}(G)$ is without meaning. In this case, EXT can be utilized for efficiently solving MAX-ORIENT as mentioned above (this is discussed in more detail in Section 5). For EXT, the notion of a *potential-sink* is of central importance. It asserts that all incident edges can be oriented towards a vertex v without introducing a new v-structure or changing the orientation of an arc. Formally, vertex v in G is called a potential-sink if (i) there is no arc $v \rightarrow x$ directed outward from v (i. e., $\mathrm{Ch}(v) = \emptyset$) and (ii) for every sibling $y \in \mathrm{Si}(v)$, vertex y is adjacent to all other neighbors of v. For example, vertex e is the only potential-sink in the graph G shown in Fig. 1.

The following fact states that a consistent extension of a graph G can be obtained by iteratively removing potential-sinks from G. This strategy is known as the Dor-Tarsi algorithm.

Fact 1 (Dor and Tarsi, 1992) *Let G_0 be an extendable PDAG and let G_i, $1 \leq i \leq n$, be obtained by removing potential-sink v_i and its incident edges from G_{i-1}. It holds that $(v_n, v_{n-1}, \dots, v_1)$ is a topological ordering of a consistent extension of G_0.*

In particular, it is true that every G_i has a potential-sink due to the observation that removing a vertex and its incident edges from an extendable graph, again yields an extendable graph (extendability is closed under taking subgraphs).

These notions are exemplified in Fig. 1. The figure shows that the consistent extension D can be obtained by iteratively identifying a potential-sink (at the start, e is the only potential-sink), orienting its incident undirected edges towards it, and continuing on the induced subgraph over the remaining vertices. In this particular case, d would be the potential-sink found afterward, followed by b and c (in arbitrary order), and finally a.

The graph M is the maximal orientation of G. It contains the arc $d \rightarrow e$ as $d \leftarrow e$ cannot occur in any consistent extension of G (it creates a new v-structure that is not in G, see Meek rule R1) and $a \rightarrow d$ follows from Meek rule R3, whereas $a - b$ and $a - c$ are oriented in both directions in different consistent extensions of D and are consequently undirected in M.

3. Two New Simple Algorithms for Extendability

As discussed above, algorithms for extending PDAGs play a key role in the maximal orientation task. While, from a theoretical point of view, previous results suggest that it is likely not possible to further improve the asymptotic run time of extendability algorithms, as Wienöbst et al. (2021) gave a conditional $O(n^3)$ lower bound for combinatorial algorithms, from a practical perspective, the current algorithms are either extremely simple, as Dor-Tarsi's approach, or very sophisticated with considerable practical overhead, as the methods proposed by Wienöbst et al. (2021). In this work, we are searching for a middle ground and give two novel approaches to extend PDAGs, with the main focus lying on simplicity and effectiveness. The first approach is a direct modification of the Dor-Tarsi algorithm, which greatly improves its empirical performance. The second one gives a practical $O(n^3)$ algorithm, which is, however, conceptually simpler than the one presented by Wienöbst et al. (2021) and has significantly less overhead.

3.1. Dor-Tarsi with Degree-Heuristic

The Dor-Tarsi algorithm has an asymptotic run time of $O(n^4)$ and is computationally simplistic. It iteratively identifies a potential-sink and then removes it and its incident edges from the graph. Hence, the whole algorithm consists of n iterations, each amounting to the task of finding a potential-sink. Naively, a potential-sink can be found in time $O(n^3)$ by going through all vertices and checking for each vertex whether it satisfies the potential-sink property. Implemented this way, each vertex is repeatedly tested for being a potential-sink throughout the algorithm (until it eventually becomes a potential-sink and is subsequently removed from the graph). While this brute-force method appears to be severely worse than more clever approaches, which store and update relevant information such as a list of all potential-sinks in sophisticated data structures, there are two notable advantages of this approach: (i) whenever a potential-sink is found, the loop over all vertices can be

Algorithm 1: A heuristic implementation of Dor-Tarsi iterating over the vertices in order of increasing degree.

```
input  : A PDAG $G = (V, A, E)$.
output: $D \in [G]$ or $\perp$ if $[G] = \emptyset$.
1  $D := (V, \emptyset)$
2  repeat $n$ times
3      for $v \in V$ in increasing degree in $G$ do
4          if $v$ is a potential-sink then
5              Remove $v$ and its incident edges from $G$.
6              Add arcs $\{(u, v) \mid u \in \mathrm{Si}_G(v) \cup \mathrm{Pa}_G(v)\}$ to $D$.
7              break
8          end
9      end
10     if no potential-sink has been found then
11         return $\perp$
12     end
13 end
14 return $D$
```

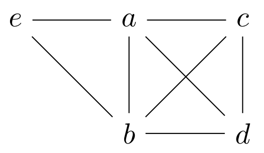
exited immediately, and (ii) the check for potential-sinkness can be cancelled once a single missing edge violates the potential-sink property.

Hence, the first observation of this work, discussed in more detail throughout the experiments in Section 4, is that in many cases, a naive implementation of the Dor-Tarsi algorithm performs quite well in practice (in particular in cases where many potential-sinks cause frequent early exits of the for-loop and also for graphs in which the test for potential-sinkness usually fails early), being competitive with more subtle approaches. Building on the surprising effectiveness of the Dor-Tarsi method, a heuristic refinement is given in Algorithm 1. The idea is to go through the vertices in order of increasing degree. Using this heuristic leads to fewer cost in case the loop is exited early as the iteration in line 3 always starts with the "cheapest" vertices, i. e., those with minimal degree (checking for potential-sinkness results in worst-case costs quadratic in the number of neighbors $O(|\mathrm{Ne}(v)|^2)$). Fig. 3 exemplifies this advantage.

It demonstrates that checking potential-sinkness for the low-degree vertices first has, on the one hand, lower cost per vertex and, on the other hand, is often more likely to succeed early (for example, vertices with degree one and no outgoing edges are always potential-sinks). We note that some overhead is induced by this approach because the degree of the vertices has to be continuously updated. However, this cost is mostly tolerable as our experiments presented in Section 4 confirm and the heuristic yields a simple and practical improvement over the standard Dor-Tarsi algorithm. Clearly, a heuristic is not always optimal and worst-case examples can be constructed where the algorithm's run time reaches its upper asymptotic bound:

Fact 2 *There are instances on which the Dor-Tarsi heuristic yields a run time of* $\Omega(n^4)$.

PDAG G



#Adj. tests for the first potential-sink (ps)

DT: $b \overset{?}{\sim} c, b \overset{?}{\sim} d, b \overset{?}{\sim} e, c \overset{?}{\sim} d, c \overset{?}{\sim} e$ ps(a) ✗

$a \overset{?}{\sim} c, a \overset{?}{\sim} d, a \overset{?}{\sim} e, c \overset{?}{\sim} d, c \overset{?}{\sim} e$ ps(b) ✗

$a \overset{?}{\sim} b, a \overset{?}{\sim} d, b \overset{?}{\sim} d$ ps(c) ✓

DTH: $a \overset{?}{\sim} b$ ps(e) ✓

Figure 3: A PDAG and the adjacency tests required by Dor-Tarsi (DT) and the heuristic adaption (DTH) for finding the first potential-sink. We assume that DT iterates over the vertices in alphabetical order. DT checks vertices a and b first, which are no potential-sinks. DTH starts with the lowest-degree vertex e. This reduces the cost of testing potential-sinkness and increases the chances of finding a potential-sink early.

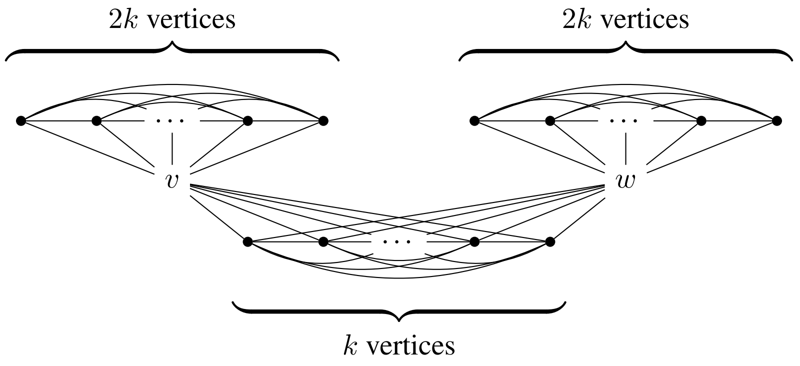


Figure 4: An example instance where the heuristic exhibits run time $\Omega(n^4)$.

Proof Consider graph $G = (V, A = \emptyset, E)$ with $V = \{v, w\} \cup C_v \cup C_w \cup C_{vw}$, where C_{vw} consists of k vertices and C_v and C_w consist of $2k$ vertices each. The edge set E is constructed as

$$E = \{\{x, y\} \mid x, y \in C_L \text{ for } L = v, w, vw\} \cup \{\{v, x\} \mid x \in C_v \cup C_{vw}\} \cup \{\{w, x\} \mid x \in C_w \cup C_{vw}\}.$$

In words, the cliques C_v, C_w are fully connected to v and w, respectively, and C_{vw} is fully connected to both v and w. Moreover, v and w are non-adjacent. The graph is illustrated in Fig. 4.

Dor-Tarsi in combination with the degree-heuristic always iterates over the vertices in C_{vw} with initial degree $k+1$ first. None of these vertices are, however, potential-sinks. Consequently, $\Omega(k)$ vertices are checked before a potential-sink is found. Moreover, testing for potential-sinkness only fails for one pair of neighbors, that is, for v and w, causing the loop over the pairs of neighbors to require $\Omega(k^2)$ iterations before exiting with high probability. Note that for $n = 5k+2$, k is in $\Omega(n)$. Overall, the steps undertaken by the heuristic yield a run time of $\Omega(n^4)$. ∎

3.2. Dor-Tarsi with Improved Worst-Case Complexity

In the previous section, a direct and effective heuristic improvement of the Dor-Tarsi algorithm is presented. In this section, we aim to give another simple adaptation of Dor-Tarsi, but this time with an $O(n^3)$ worst-case run time, thus matching the lower-bounds of the algorithm given in (Wienöbst et al., 2021).

If we take a look at the counterexample from Fig. 4 again, we observe that Algorithm 1 repeatedly iterates over the same neighborhoods when searching for a potential-sink, i. e., there is no information stored between iterations. It is clearly desirable to avoid this repeated computational effort. Previously, improvements were made by constructing elaborate data structures, which maintain the set of potential-sinks throughout the course of the algorithm (Wienöbst et al., 2021). Such data structures, however, induce a significant practical overhead and, in particular, demand an expensive initialization step at the start of the algorithm, even before searching for the first potential-sink.

To address these issues, our proposed approach iterates over the neighbors of every vertex v exactly once, even though potential-sinkness of v might be checked multiple times during the course of the algorithm. After the neighbors of a vertex v have been iterated over, all tuples of neighbors violating the potential-sink property of v are stored in the set $B[v]$, which is afterward continuously updated every time vertices are removed from the graph. The sets $B[v]$ are computed lazily, that is, only at the time the potential-sinkness of v is checked for the first time and thus there is no pre-computation involved. In case the set of violating neighbors $B[v]$ is empty or becomes empty by removing other vertices, v satisfies the potential-sink property, which is due to the observation that throughout the course of the algorithm no *new* potential-sinkness violations are incurred.

The whole approach is described in detail in Algorithm 2. We combine it with the heuristic from the previous section, which is generally favorable to use. Algorithm 2 combines the advantages of both DT and the algorithm from Wienöbst et al. (2021) by, one the one hand, storing information from previous iterations to ensure an $O(n^3)$ worst-case run time while, on the other hand, keeping the overhead to a minimum and, in particular, not relying on an initialization step.

Theorem 1 *Algorithm 2 is implementable with expected worst-case time complexity* $O(n^3)$.

Proof We assume that edge deletion and adjacency tests are supported in constant time and iterating over the neighbors of v has cost $O(|\mathrm{Ne}_G(v)|)$. Using hash tables to store the neighbors of a vertex, these run times are reached *in expectation*. The graph representation is discussed in more detail in the appendix (Section A).

The computation of neighbors violating the potential-sink property in line 7 can clearly be implemented in $O(n^2)$ by looping over all pairs of neighbors of v and checking adjacency. Due to the flag $C[v]$, the neighborhood for each vertex v is visited exactly once and thus line 7 is executed $O(n)$ times, yielding a total run time of $O(n^3)$ for searching potential-sinks. The removal of a potential-sink in lines 10 to 15 runs in time $O(n^2)$ as v has at most n neighbors for which edges are removed from G and added to D and there are at most $O(n^2)$ tuples in the sets in B where v is included. By storing all tuples where v is included separately, we can iterate over them and remove each tuple from the corresponding set in $O(1)$. As every vertex is removed at most once from the graph, the removal is repeated $O(n)$ times, yielding a run time of $O(n^3)$ for the removal of all vertices. Consequently, the run time of the whole algorithm is bounded by $O(n^3)$. ■

The correctness of the algorithm follows immediately from the correctness of the Dor-Tarsi algorithm (Fact 1) as our proposed modification only avoids repeated visits to neighbors but does

Algorithm 2: An adaptation of Dor-Tarsi with worst-case complexity $O(n^3)$.

```
input  : A PDAG $G = (V, A, E)$.
output: $D \in [G]$ or $\perp$ if $[G] = \emptyset$.
1  $D := (V, \emptyset)$
2  $B :=$ array of $n$ initially empty sets
3  $C :=$ bitvector of length $n$ initialized with false
4  while there are vertices left in $G$ do
5      foreach $v \in V$ in increasing degree in $G$ do
6          if $C[v] =$ false and $\mathrm{Ch}_G(v) = \emptyset$ then
7              $B[v] := B[v] \cup \{(u, u') \mid u \in \mathrm{Si}_G(v), u' \in \mathrm{Si}_G(v) \cup \mathrm{Pa}_G(v) \wedge u \neq u' \wedge u \not\sim_G u'\}$
8              $C[v] :=$ true
9          end
10         if $C[v] =$ true and $B[v] = \emptyset$ then
11             Remove $v$ and its incident edges from $G$.
12             Add arcs $\{(u, v) \mid u \in \mathrm{Si}_G(v) \cup \mathrm{Pa}_G(v)\}$ to $D$.
13             Remove from all sets in $B$ tuples including $v$.
14             break
15         end
16     end
17     if no potential-sink has been found then
18         return $\perp$
19     end
20 end
21 return $D$
```

not skip any neighbor check. As a final remark, we note that Algorithm 2 is also implementable using a combination of linked lists and an adjacency matrix to represent the neighbors of a vertex v, as proposed by (Wienöbst et al., 2021). Using such a representation, instead of collecting all neighbors of v violating the potential-sink property in line 7, one could stop as soon as the first pair of neighbors violating the property is found and store a pointer to the violating neighbor, allowing to start the iteration over the neighbors of v the next time at the lastly visited neighbor[1]. However, using linked lists instead of hash sets to represent adjacencies has no impact on the worst-case complexity and is not significantly faster than the hashed data structure. More details about the comparison between these two representations can be found in Appendix A.

4. Evaluation of Extension Algorithms

In this section, we conduct an experimental evaluation of the algorithms discussed in the previous sections to demonstrate their practical effectiveness. We compare the Dor-Tarsi algorithm (DT), its heuristic refinement (DTH), its adaptation with improved worst-case complexity (DTIC), and the algorithm given by Wienöbst et al. (2021) (WBL). All algorithms are implemented in Julia (Bezanson et al., 2017). We present the results for random PDAGs, which are generated by (i) creating a

1. Such a strategy is not possible for hashed data structures due to potential rehashing.

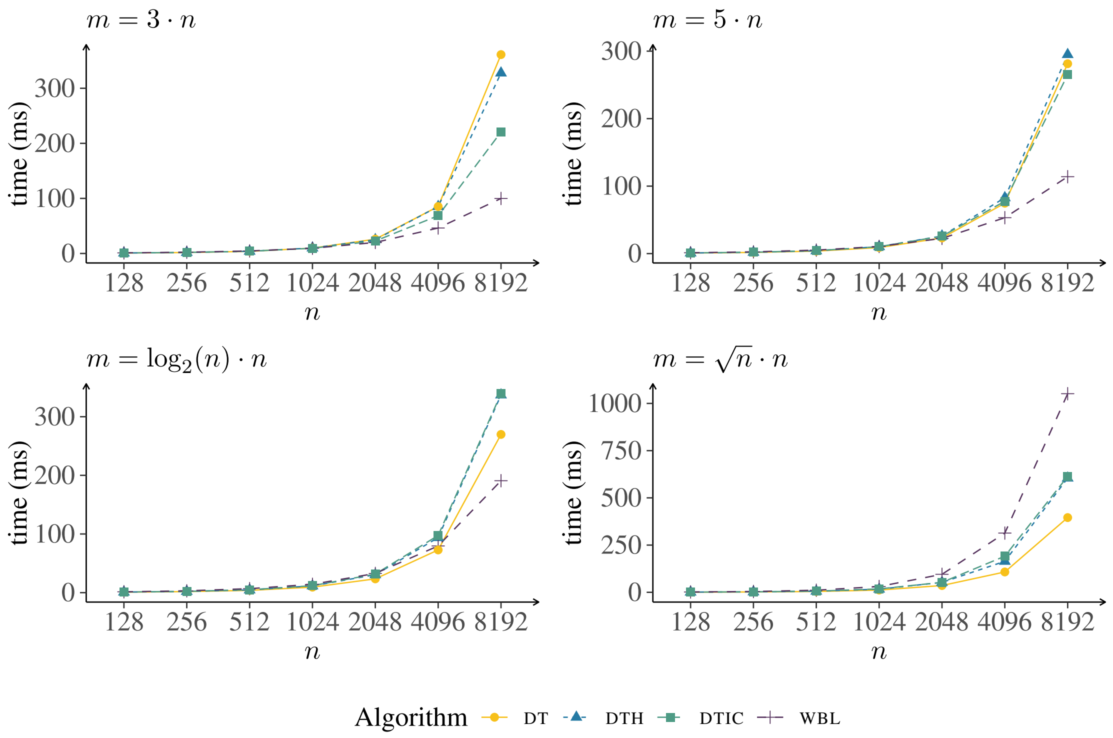


Figure 5: Run times of the algorithms DT, DTH, DTIC, and WBL on randomly generated PDAGs of n vertices and m edges, with $m = 3 \cdot n$ (top left), $m = 5 \cdot n$ (top right), $m = \log_2(n) \cdot n$ (bottom left), and $m = \sqrt{n} \cdot n$ (bottom right).

random DAG D, (ii) replacing all directed edges not participating in a v-structure by an undirected edge, and (iii) orienting between two and five (randomly chosen) undirected edges according to D, resulting in an extendable PDAG. The initial DAG D is generated by creating a random undirected graph and afterward using a random permutation of its vertices as a topological ordering according to which the edges are then oriented. We vary the number of vertices in the input graphs by setting $n = 128, 256, \dots, 8192$ and the number of edges is set to $m = 3 \cdot n, 5 \cdot n, \log_2(n) \cdot n, \sqrt{n} \cdot n$. For each choice of parameters, we generate ten instances and then run every algorithm ten times on each instance. The run times reported in this section and in upcoming sections are averages over all runs on those ten instances. Further experimental results for scale-free PDAGs and chordal graphs are given in Appendix B.

The results are shown in Fig. 5. In the top left plot, the run times on sparse graphs with $m = 3 \cdot n$ edges are given. For small graphs, there are no visible differences among the run times of the algorithms. However, with an increasing number of vertices, it becomes clear that WBL performs best, which is quite expected, as it was shown to have linear-time $O(m)$ for sparse graphs (formalized as constant-degeneracy graphs in (Wienöbst et al., 2021)). The run time of DT increases rapidly as the number of vertices grows, while both DTH and DTIC exhibit a slightly smaller increase. Although DTIC does not outperform WBL, it shows a significant improvement compared to DT.

In both the top right plot and the bottom left plot, WBL yields still the best performance among the algorithms. For denser graphs, with $m = \sqrt{n} \cdot n$ (bottom right), WBL yields a weaker performance than the other algorithms. This can be explained by the fact that WBL relies on an initialization step, which requires an iteration of every pair of neighbors for each vertex in the graph and thereby induces costs of $O(n^3)$ for dense graphs. Those costs emerge even before searching for the first potential-sink and they are particularly high as the initialization loops cannot be exited early. All of the three other approaches do allow for early exits, which turns out to be a non-negligible advantage because the graph is getting smaller and smaller during the course of the algorithm, meaning that the high initialization effort of WBL dominates the run time. Particularly for denser graphs, DT is even slightly faster than DTH and DTIC, which indicates that both DTH and DTIC induce some overhead by continuously maintaining the vertices in sorted order by their degree.

Overall, DTIC provides a stable performance across different graph sizes and graph densities, verifying that it combines the advantages of DT and WBL. DT is often a solid choice in practice but its performance depends heavily on the order of checking vertices for potential-sinkness and thus combining its advantages with a guaranteed worst-case run time of $O(n^3)$ yields a promising algorithm for practical applications.

5. Application to Maximal Orientations

Building on the results in the previous sections, we are able to demonstrate an immediate application of extension algorithms. As shown by Chickering (1995) and Wienöbst et al. (2021), it is possible to utilize extension algorithms when computing the maximal orientation of a PDAG G. In practice, it is common to implement the step from PDAG to MPDAG the "direct way", that is, by repeatedly applying the Meek rules (shown in Fig. 2) in a while-loop. We argue that using consistent extensions not only gives desirable worst-case guarantees, but also performs remarkably well in practice.

5.1. How to Use Extendability for the Computation of Maximal Orientations

Applying the Meek rules repeatedly to a PDAG G yields its maximal orientation. A direct implementation, which loops over the given graph repeatedly is computationally expensive, leading to worst-case run times such as $O(n^4 \cdot m)$[2]. Clearly, it is preferable to traverse the graph only a single time, while deciding which edges should be oriented. The crucial observation is that this can be achieved by utilizing a topological ordering of a consistent extension of G.

More precisely, when computing the maximal orientation of a PDAG, one can distinguish between two situations: (i) the result of applying the Meek rules yields a CPDAG, such as in the final phase of the PC algorithm, and (ii) in case of additional background knowledge, the resulting graph is not necessarily a CPDAG, but a maximally oriented PDAG (i. e., an MPDAG).

In case (i), the CPDAG can be obtained without applying the Meek rules by extending the PDAG into a DAG and afterward computing the corresponding CPDAG directly from the DAG. Indeed, for this second step from DAG to CPDAG, Chickering (1995) gave a linear-time algorithm[3]. In case (ii), starting with the CPDAG obtained as in (i), that is, by performing Chickering's DAG-to-CPDAG algorithm on a consistent extension D, further orienting all edges which are directed

2. $O(m)$ edges might be oriented successively and naively checking the applicability of the Meek rules results in costs of $O(n^4)$.
3. The algorithm proceeds by checking for each edge whether it should be undirected. Although not utilizing the Meek rules, the algorithm makes heavy use of the topological ordering provided by the consistent extension.

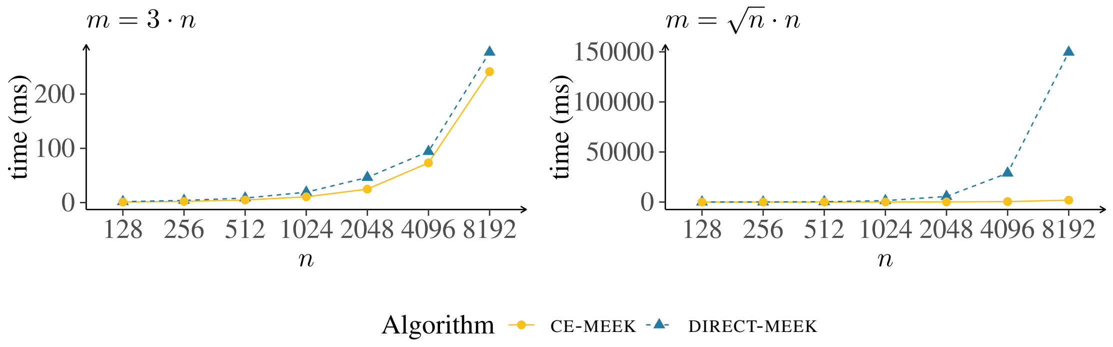


Figure 6: Run times of the algorithms DIRECT-MEEK and CE-MEEK on randomly generated PDAGs with $m = 3 \cdot n$ edges (left) and $m = \sqrt{n} \cdot n$ edges (right).

in G (i. e., the background knowledge edges), and afterward applying the Meek rules in a single iteration over the vertices of the graph in order of a topological ordering of D yields the maximal orientation (Wienöbst et al., 2021). While this procedure appears quite involved, we demonstrate that the extension step is actually the most expensive part and the subsequent steps are comparably cheap. Moreover, we show that this strategy is significantly faster than applying the Meek rules directly. From a theoretical point of view, these approaches guarantee an $O(n^3)$ worst-case run time, which is a significant improvement as well.

5.2. Experimental Comparison with the Direct Application of the Meek Rules

In this section, we compare the two approaches to maximally orient a given PDAG in an experimental evaluation. More specifically, we compare the direct application of Meek's rules (DIRECT-MEEK) to the above mentioned approach originally introduced by Chickering (1995) and generalized to arbitrary PDAGs by Wienöbst et al. (2021), which utilizes the topological ordering of a consistent extension (CE-MEEK). The input PDAGs are the same as in Section 4, and further results for scale-free PDAGs are provided in Appendix C.

The run times of the two approaches on random PDAGs are presented in Fig. 6. Not surprisingly, CE-MEEK outperforms DIRECT-MEEK on every input graph. While the difference in their run times is relatively small on sparse input graphs (left), it increases drastically as n grows on denser input graphs (right)[4]. In addition to the plain comparison of run times, we also analyze the time spent by CE-MEEK on the different phases of the algorithm. More specifically, we measure the time for (i) computing a consistent extension, (ii) finding the corresponding CPDAG, and (iii) applying the Meek rules in a single iteration over the vertices. The results for the same input graphs as before are depicted in Fig. 7. Each bar is divided into the proportions of the total run time for the three phases, i. e., adding together the proportions of the three phases equals 100 percent of the run time. For sparse graphs (left), the majority of the time (more than 75 percent) is spent on phase (i), demonstrating that in order to obtain fast algorithms to compute maximal orientations, it

4. We use DTIC to compute the consistent extension. For sparse graphs, the use of WBL can give a significant speedup, which would in turn translate to CE-MEEK.

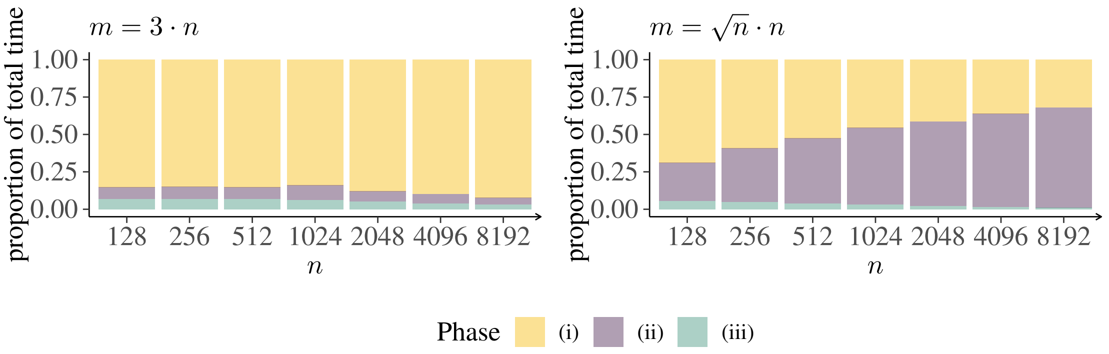


Figure 7: Proportions of the total run time of CE-MEEK for the three phases on randomly generated PDAGs with $m = 3 \cdot n$ edges (left) and $m = \sqrt{n} \cdot n$ edges (right).

is crucial to have algorithms solving the extension problem efficiently. Even though the proportion of phase (ii) dominates for dense graphs (right) with many vertices, phase (i) still accounts for a significant portion (at least 25 percent) of the total run time.

6. Conclusions

In this paper, we demonstrate the effectiveness of utilizing consistent extensions for the task of maximally orienting a PDAG. We started by revisiting the extension problem, presenting two new approaches to efficiently compute consistent DAG extensions in practical applications. The first approach (DTH) refines the widespread Dor-Tarsi algorithm by the employment of a simple heuristic, which reduces the computational effort by prioritizing low-cost vertices throughout its iterations. The second approach (DTIC) stores additional information between iterations to avoid duplicate iterations and thereby matches the worst-case complexity of the WBL algorithm, which achieves the conditional lower bound of $O(n^3)$ for the extension problem. In a practical evaluation, we show that DTIC exhibits the most stable performance overall, combining the advantages of the other approaches. Based on those insights and results, we highlight an important application – the maximal orientation of PDAGs, a procedure ubiquitous in causal discovery. We demonstrate experimentally that utilizing consistent extensions yields highly reliable and effective algorithms for this task, which outperform the direct use of the Meek rules currently used most commonly.

Acknowledgments

The research of Malte Luttermann was partly supported by the Medical Cause and Effects Analysis (MCEA) project.

References

Réka Albert and Albert-László Barabási. Statistical Mechanics of Complex Networks. *Reviews of Modern Physics*, 74:47–97, 2002.

Steen A. Andersson, David Madigan, and Michael D. Perlman. A Characterization of Markov Equivalence Classes for Acyclic Digraphs. *The Annals of Statistics*, 25:505–541, 1997.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59:65–98, 2017.

David Maxwell Chickering. A Transformational Characterization of Equivalent Bayesian Network Structures. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 87–98. Morgan Kaufmann Publishers Inc., 1995.

David Maxwell Chickering. Optimal Structure Identification with Greedy Search. *Journal of Machine Learning Research*, 3:507–554, 2002.

Dorit Dor and Michael Tarsi. A Simple Algorithm to Construct a Consistent Extension of a Partially Oriented Graph. Technicial Report 185, Cognitive Systems Laboratory, UCLA, 1992.

Felix Elwert. Graphical Causal Models. In *Handbook of Causal Analysis for Social Research*, Handbooks of Sociology and Social Research, pages 245–273. Springer, 2013.

Richard Guo and Emilija Perković. Minimal Enumeration of all Possible Total Effects in a Markov Equivalence Class. In *Proceedings of the Twenty-Fourth International Conference on Artificial Intelligence and Statistics (AISTATS-21)*, pages 2395–2403. PMLR, 2021.

Alain Hauser and Peter Bühlmann. Characterization and Greedy Learning of Interventional Markov Equivalence Classes of Directed Acyclic Graphs. *Journal of Machine Learning Research*, 13: 2409–2464, 2012.

Yang-Bo He and Zhi Geng. Active Learning of Causal Networks with Intervention Experiments and Optimal Designs. *Journal of Machine Learning Research*, 9:2523–2547, 2008.

Yangbo He, Jinzhu Jia, and Bin Yu. Counting and Exploring Sizes of Markov Equivalence Classes of Directed Acyclic Graphs. *Journal of Machine Learning Research*, 16:2589–2609, 2015.

Markus Kalisch and Peter Bühlman. Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm. *Journal of Machine Learning Research*, 8:613–636, 2007.

Markus Kalisch, Martin Mächler, Diego Colombo, Marloes H. Maathuis, and Peter Bühlmann. Causal Inference Using Graphical Models with the R Package `pcalg`. *Journal of Statistical Software*, 47:1–26, 2012.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models – Principles and Techniques*. MIT Press, 2009.

Marloes H. Maathuis, Markus Kalisch, and Peter Bühlmann. Estimating High-Dimensional Intervention Effects from Observational Data. *The Annals of Statistics*, 37:3133–3164, 2009.

Christopher Meek. Causal Inference and Causal Explanation with Background Knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 403–410. Morgan Kaufmann Publishers Inc., 1995.

Judea Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition, 2009.

Jean-Philippe Pellet and André Elisseeff. Using Markov Blankets for Causal Structure Learning. *Journal of Machine Learning Research*, 9:1295–1342, 2008.

Emilija Perković, Johannes Textor, Markus Kalisch, and Marloes H. Maathuis. Complete Graphical Characterization and Construction of Adjustment Sets in Markov Equivalence Classes of Ancestral Graphs. *Journal of Machine Learning Research*, 18:220:1–220:62, 2017.

Oylum Şeker, Pinar Heggernes, Tınaz Ekim, and Z. Caner Taşkın. Linear-Time Generation of Random Chordal Graphs. In *Proccedings of the 10th International Conference on Algorithms and Complexity (CIAC-17)*, pages 442–453. Springer International Publishing, 2017.

Karthikeyan Shanmugam, Murat Kocaoglu, Alexandros G. Dimakis, and Sriram Vishwanath. Learning Causal Graphs with Small Interventions. In *Advances in Neural Information Processing Systems 28 (NIPS-15)*, pages 3195–3203. Curran Associates, Inc., 2015.

Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, Prediction, and Search*. MIT Press, 2nd edition, 2000.

Chandler Squires. `causaldag`*: Creation, Manipulation, and Learning of Causal Models*, 2018. URL `https://github.com/uhlerlab/causaldag`.

Chandler Squires, Sara Magliacane, Kristjan Greenewald, Dmitriy Katz, Murat Kocaoglu, and Karthikeyan Shanmugam. Active Structure Learning of Causal DAGs via Directed Clique Trees. In *Advances in Neural Information Processing Systems 33 (NIPS-20)*, pages 21500–21511, 2020.

Benito van der Zander and Maciej Liśkiewicz. Separators and Adjustment Sets in Markov Equivalent DAGs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, (AAAI-16)*, pages 3315–3321. AAAI Press, 2016.

Thomas Verma and Judea Pearl. Equivalence and Synthesis of Causal Models. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI-90)*, pages 255–270. Elsevier Science Inc., 1990.

Thomas Verma and Judea Pearl. An Algorithm for Deciding if a Set of Observed Independencies Has a Causal Explanation. In *Proceedings of the Eighth International Conference on Uncertainty in Artificial Intelligence (UAI-92)*, pages 323–330. Morgan Kaufmann Publishers Inc., 1992.

Marcel Wienöbst, Max Bannach, and Maciej Liśkiewicz. Polynomial-Time Algorithms for Counting and Sampling Markov Equivalent DAGs. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI-21)*, pages 12198–12206. AAAI Press, 2021.

Marcel Wienöbst, Max Bannach, and Maciej Liśkiewicz. Polynomial-Time Algorithms for Counting and Sampling Markov Equivalent DAGs with Applications. *arXiv preprint arXiv:2205.02654*, 2022.

Marcel Wienöbst, Max Bannach, and Maciej Liśkiewicz. Extendability of Causal Graphical Models: Algorithms and Computational Complexity. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI-21)*, pages 1248–1257. PMLR, 2021.

Appendix A. Comparison of Graph Representations

Choosing an appropriate graph representation is crucial to obtain effective extension algorithms. In particular, two important operations that need to be executed fast to solve the extension problem efficiently are adjacency tests and the removal of vertices and edges from the graph. In this section, we compare two different graph representations to support carrying out these operations efficiently. The first representation utilizes a hashed data structure to represent adjacencies in a graph and a the second one employs a combination of linked lists and an adjacency matrix for this purpose.

The graph representation used throughout the course of this paper keeps track of three sets storing the neighbors (ingoing, outgoing, and undirected) for each vertex in the graph. To access elements in these sets in expected constant time, hashing is applied. Representing the neighbors of every vertex with the help of hash sets leads to a simple, yet practical graph representation. However, iterating over hash sets (e. g., during the potential-sink check) is suboptimal, as the entire hash table has to be iterated over. Furthermore, as outlined in Section 3.2, a representation based on linked lists allows DTIC to exit its loops earlier, that is, as soon as a neighbor violating the potential-sink property is found, which is not possible if neighbors are stored in a hashed data structure.

To analyze the impact of these aspects on the performance of DTIC and the other algorithms, we also implemented a graph representation using an adjacency matrix and linked lists to store adjacencies in the graph, which was proposed by (Wienöbst et al., 2021). More precisely, there are three linked lists to store the ingoing, outgoing, and undirected neighbors, respectively, for each vertex. The adjacency matrix contains a pointer to the corresponding linked list entry for every edge in the graph. Adjacency tests run in constant time on the adjacency matrix and the removal of vertices from the graph runs in constant time as well, as the corresponding pointer in the matrix enables an access to any neighbor in the linked list in $O(1)$. The main drawback of this approach is clearly the large memory requirement of the $O(n^2)$ adjacency matrix. Allocating the corresponding memory is also time-consuming.

We found that the usage of linked lists instead of hash sets does not provide significant improvements for the run times of the extension algorithms. In most of the evaluated scenarios, the contrary is the case. A direct comparison of implementations using hash sets and linked lists can be found in Table 1 where the input graphs and settings are the same as in Section 4.

Clearly, the usage of linked lists appears to be at a disadvantage for the instances considered in this work. However, we remark that, in some cases, the usage of linked lists is beneficial. For example, on chordal graphs (see Appendix B for more details), DT is faster on graphs with a linked list implementation compared to graphs using a hashed data structure. In Appendix B, we also introduce scale-free PDAGs, for which the comparison of hash sets and linked lists yields the same relations as in Table 1. In conclusion, our experiments demonstrate that the linked list representation introduces overhead, which does not pay off for the generally sparse graphs considered in this work and usually occurring in practice.

Appendix B. Further Experimental Results for Extendability

To complement the experimental results for randomly generated PDAGs presented in Section 4, we evaluate the algorithms DT, DTH, DTIC, and WBL on scale-free PDAGs and chordal graphs. Scale-free graphs are graphs whose degree distribution is a power law distribution, i. e., there are few vertices with a high degree while most of the vertices have a rather small degree. The scale-free PDAGs are generated in a similar fashion as the graphs from Section 4, that is, the general

n	DT	DT-LL	DTH	DTH-LL	DTIC	DTIC-LL	WBL	WBL-LL
128	0.83	0.14	0.88	0.19	0.94	0.20	1.29	0.26
256	1.80	0.41	1.89	0.54	1.97	0.58	2.60	0.68
512	3.89	1.60	4.07	1.88	4.13	2.03	5.21	2.15
1024	9.96	6.76	10.19	7.62	9.72	8.28	10.88	9.53
2048	27.01	39.55	25.66	40.40	23.32	42.86	23.35	50.90
4096	87.13	149.06	85.38	190.61	68.50	201.69	51.98	236.52
8192	370.79	606.02	323.64	608.67	222.52	654.45	116.92	871.23

n	DT	DT-LL	DTH	DTH-LL	DTIC	DTIC-LL	WBL	WBL-LL
128	0.87	0.19	0.97	0.28	1.04	0.29	2.35	0.46
256	1.86	0.55	2.11	0.81	2.23	0.84	5.61	1.21
512	3.92	1.97	4.71	2.86	4.93	2.98	13.09	4.30
1024	9.62	9.14	11.90	11.68	12.14	12.08	30.98	15.93
2048	23.60	48.02	30.77	54.55	31.38	56.30	75.68	72.03
4096	72.80	175.78	93.31	236.88	96.00	244.46	209.80	323.04
8192	269.53	685.58	333.44	752.79	336.09	785.27	594.51	1159.83

Table 1: Comparison of the algorithms DT, DTH, DTIC, and WBL with a linked lists implementation (-LL) on random PDAGs with $m = 3 \cdot n$ edges (above) and $m = \log_2(n) \cdot n$ edges (below).

procedure of (i) creating a random DAG D, (ii) replacing all directed edges not participating in a v-structure by an undirected edge, and (iii) orienting between two and five (randomly chosen) undirected edges according to D stays the same but the initial DAG D is generated in a different way. More precisely, the initial DAG is generated by first creating a random undirected scale-free graph using the Barabási-Albert model (Albert and Barabási, 2002) and afterward constituting a random permutation of its vertices as a topological ordering according to which the edges are then directed. All parameter choices are identical to those in Section 4.

To generate the chordal graphs, we apply the random subtree intersection method introduced by Şeker et al. (2017). Chordal graphs provide an interesting addition to PDAGs as they are fully undirected and extendable by definition, thus demanding quite some computational effort to extend the graph. We set $n = 128, 256, \ldots, 8192$ again and use $k = 3, 5, \log_2(n), \sqrt{n}$ as a parameter for the random subtree intersection method. The parameter k determines the average size of the random subtrees used to generate the chordal graph and thus controls the number of edges in the graph. However, the exact number of edges slightly differs between instances.

The results for scale-free PDAGs can be found in Fig. 8 where we observe the same patterns as in Fig. 5, i. e., WBL is the fastest for sparser graphs ($m = 3 \cdot n, 5 \cdot n, \log_2(n) \cdot n$) but is the slowest on denser graphs ($m = \sqrt{n} \cdot n$) while the opposite holds for DT (that is, DT is the slowest on sparse graphs with $m = 3 \cdot n$ and the fastest on dense graphs with $m = \sqrt{n} \cdot n$). In total, DTIC yields a stable performance again, showing that DTIC retains the practicality of DT while maintaining the theoretical bounds on the run time complexity of WBL.

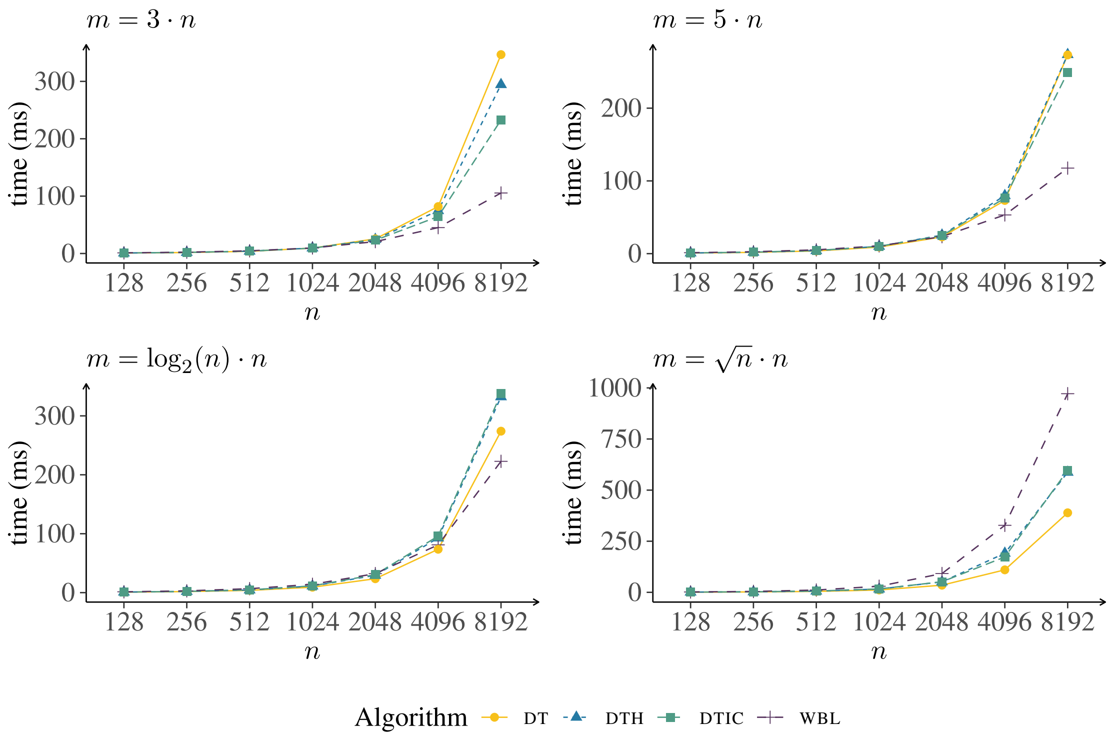


Figure 8: Run times of the algorithms DT, DTH, DTIC, and WBL on randomly generated scale-free PDAGs of n vertices and m edges, with $m = 3 \cdot n$ (top left), $m = 5 \cdot n$ (top right), $m = \log_2(n) \cdot n$ (bottom left), and $m = \sqrt{n} \cdot n$ (bottom right).

Further, Fig. 9 displays the run times of DT, DTH, DTIC, and WBL on chordal graphs. Note that the y-axes are log-scaled in all plots. DT is inferior to all other algorithms sparse graphs (top left and top right) and also becoming the slowest on slightly denser graphs (bottom left). Eventually, the run time of DT becomes almost 100 times slower than the run times of the other algorithms, for example at $n = 8192$ and $k = 3$ (among others). WBL handles sparse graphs (top left and top right) roughly as good as DTH and DTIC but cannot keep pace for the denser graphs shown at the bottom plots. The results indicate that DTH and DTIC yield the best performance on graphs where more effort is necessary to compute a consistent extension.

Appendix C. Further Experimental Results for Maximal Orientations

For the sake of completeness, we also give further experimental results for the algorithms DIRECT-MEEK and CE-MEEK in addition to the results presented in Section 5.2. We report the results for scale-free PDAGs which are identical to those from the previous section and add more edge densities to the evaluated scenarios from Section 5.2[5].

5. Chordal graphs are obviously not interesting in the setting of maximal orientations as they contain only undirected edges and hence no Meek rule is applicable on them.

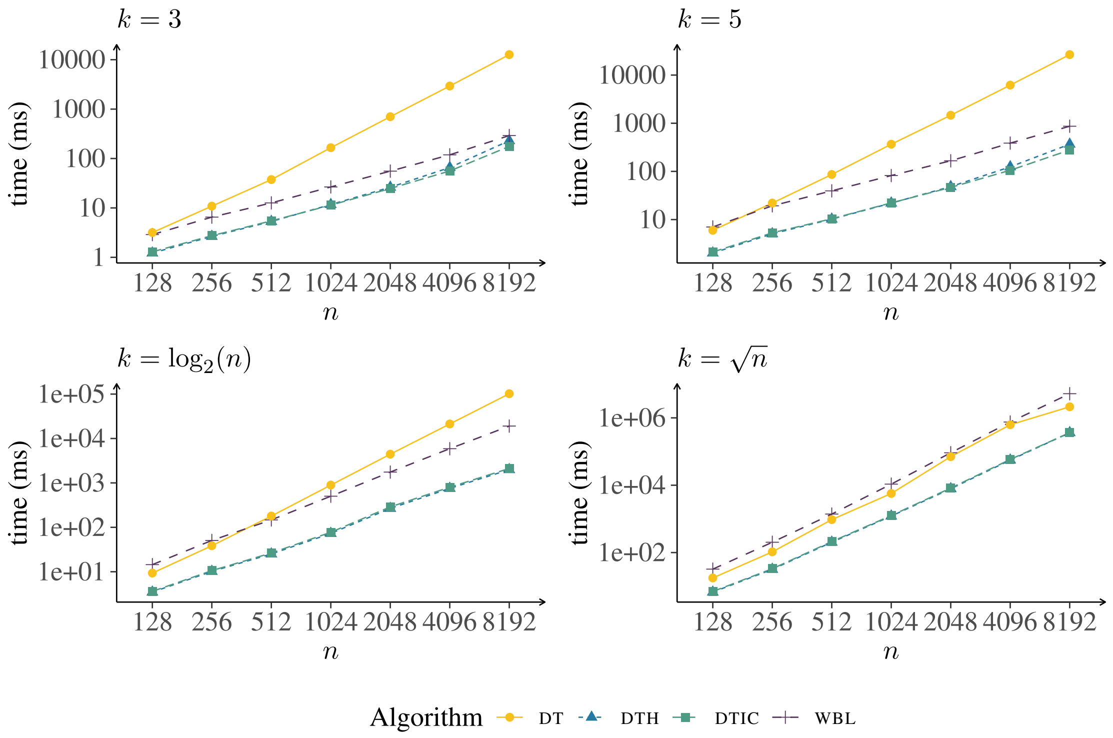


Figure 9: Run times of the algorithms DT, DTH, DTIC, and WBL on randomly generated chordal graphs with $k = 3$ (top left), $k = 5$ (top right), $k = \log_2(n)$ (bottom left), and $k = \sqrt{n}$ (bottom right) with all y-axes being log-scaled.

Fig. 10 depicts the run times of DIRECT-MEEK and CE-MEEK on randomly generated PDAGs containing $m = 5 \cdot n$ and $m = \log_2(n) \cdot n$ edges. These PDAGs are generated the same way as the PDAGs in Fig. 6. The run times pictured in Fig. 10 show the same pattern as in Fig. 6, namely that CE-MEEK is faster than DIRECT-MEEK in every scenario and the advantage of CE-MEEK increases with an increasing number of vertices and edges in the input graph, verifying that the use of consistent extensions for the computation of maximal orientations is highly beneficial.

The time spent by CE-MEEK on the different phases of the algorithm for PDAGs with $m = 5 \cdot n$ and $m = \log_2(n) \cdot n$ edges is plotted in Fig. 11. While the left plot is similar to the left plot from Fig. 7, the right plot in Fig. 11 exhibits a greater proportion of the total run time needed for phase (i) (that is, computing a consistent extension) than the right plot in Fig. 7, showing that phase (i) is dominant for sparser graphs and phase (ii) requires the majority of the total run time for dense graphs ($m = \sqrt{n} \cdot n$, right plot in Fig. 7).

Fig. 12 presents the run times of DIRECT-MEEK and CE-MEEK on scale-free PDAGs. As in Fig. 6 and Fig. 10, CE-MEEK is superior to DIRECT-MEEK on all input graphs. Overall, handling scale-free PDAGs takes more computational effort compared to handling PDAGs having their edges distributed at random.

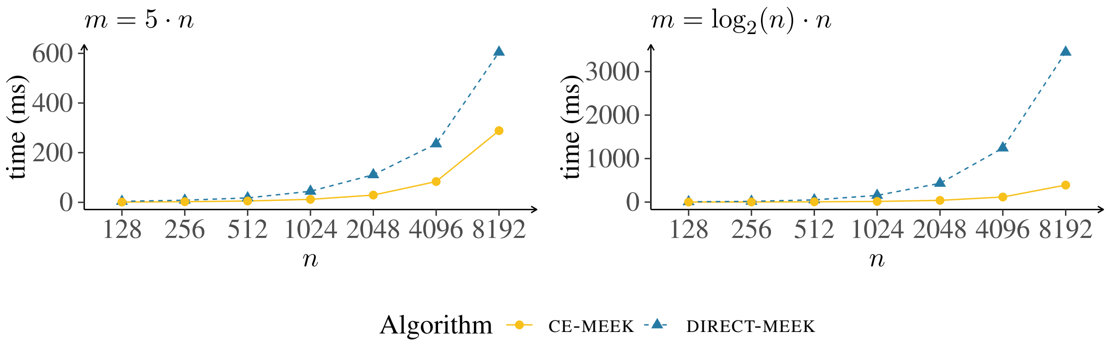


Figure 10: Run times of the algorithms DIRECT-MEEK and CE-MEEK on randomly generated PDAGs with $m = 5 \cdot n$ edges (left) and $m = \log_2(n) \cdot n$ edges (right).

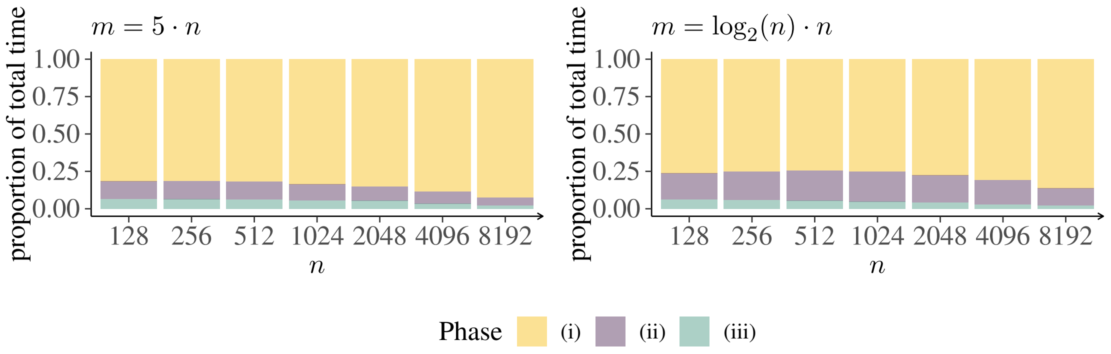


Figure 11: Proportions of the total run time of CE-MEEK for the three phases on randomly generated PDAGs with $m = 5 \cdot n$ edges (left) and $m = \log_2(n) \cdot n$ edges (right).

Furthermore, a visualization of the time spent by CE-MEEK on the different phases of the algorithm for scale-free input PDAGs is given in Fig. 13. The plots for $m = 3 \cdot n$ (top left) and $m = \sqrt{n} \cdot n$ (bottom right) are similar to the plots in Fig. 7 and the plots for $m = 5 \cdot n$ (top right) and $m = \log_2(n) \cdot n$ (bottom left) are similar to the plots in Fig. 11. Despite the similarities, we observe greater proportions of phase (ii) (i. e., finding the corresponding CPDAG to the DAG computed in the first phase) in Fig. 13 than in Fig. 7 and Fig. 11, showing that phase (ii) requires more effort on scale-free PDAGs than on PDAGs having their edges distributed at random. It becomes evident again that higher graph densities increase the proportion of phase (ii) of the total run time.

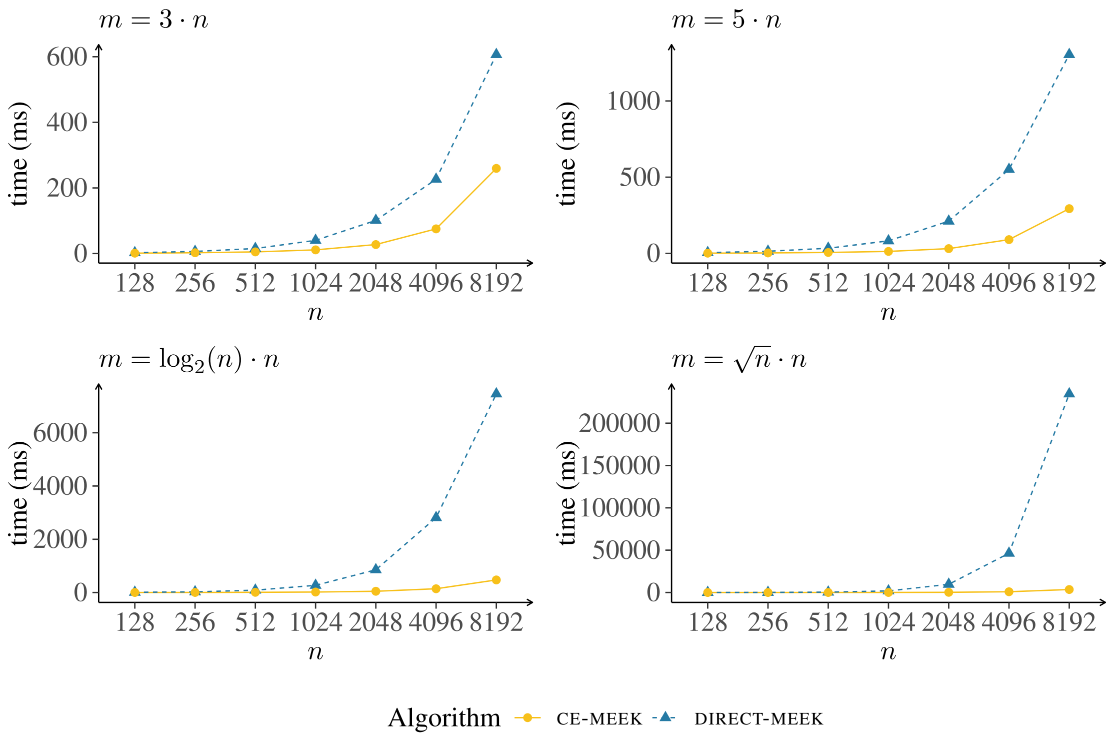


Figure 12: Run times of the algorithms DIRECT-MEEK and CE-MEEK on randomly generated scale-free PDAGs of n vertices and m edges, with $m = 3 \cdot n$ (top left), $m = 5 \cdot n$ (top right), $m = \log_2(n) \cdot n$ (bottom left), and $m = \sqrt{n} \cdot n$ (bottom right).

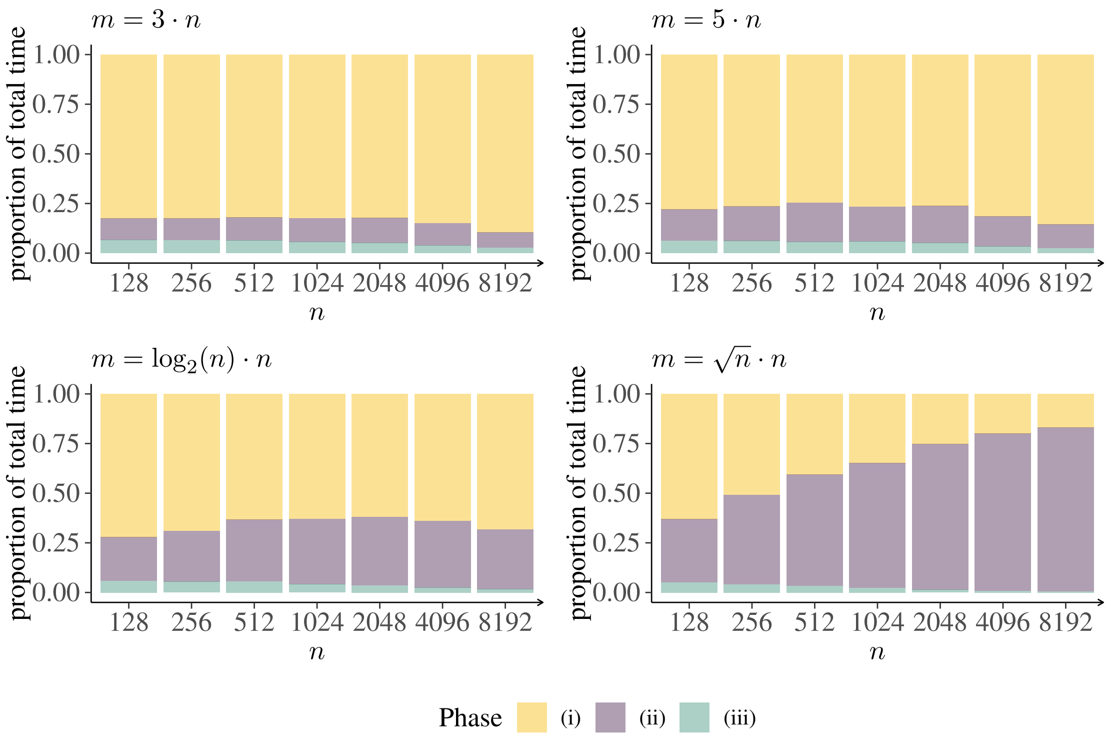


Figure 13: Proportions of the total run time of CE-MEEK for the three phases on randomly generated scale-free PDAGs of n vertices and m edges, with $m = 3 \cdot n$ (top left), $m = 5 \cdot n$ (top right), $m = \log_2(n) \cdot n$ (bottom left), and $m = \sqrt{n} \cdot n$ (bottom right).