Extendability of Causal Graphical Models: Algorithms and Computational Complexity (Supplementary material)

Marcel Wienöbst[1] **Max Bannach**[1] **Maciej Liśkiewicz**[1]

[1]Institute for Theoretical Computer Science, University of Lübeck, Germany

To simplify the navigation within the Supplementary material, it is organized with the same sections as the main part.

MISSING PROOFS OF SECTION 3: LOWER BOUNDS UNDER THE STRONG TRIANGLE CONJECTURE

Proof of Lemma 3.1. Let $G = (V, E)$ be an instance of TRIANGLE. Construct the graph $G' = (V_1 \cup V_2 \cup V_3, E')$ with $V_i = \{ v_i \mid v \in V \}$ and $E' = \{ \{u_i, v_j\} \mid \{u, v\} \in E$ with $i, j \in \{1, 2, 3\}$ and $i \neq j \}$, i. e., create three copies of the vertex set and insert the edges only between vertices in different copies.

We assume that G does not contain self-loops (as they are not part of any triangle anyway) and show that G contains a triangle if, and only if, G' contains one:

- $\Rightarrow$ Let $a, b, c \in V$ be a triangle in G. Then a_1, b_2, c_3 is a triangle in G'.
- $\Leftarrow$ Since G' is 3-partite, any triangle contains a vertex of each partition. Let a_1, b_2, c_3 be such a triangle; then the corresponding vertices $a, b, c \in V$ are mutually distinct and, hence, form a triangle.

This reduction can be implemented in time $O(n + m)$ and increases the instance only by a constant factor. □

Proof of Theorem 3.3. Apply Lemma 3.1 to reduce TRIANGLE to 3PART-TRIANGLE and let $G = (V, E, \emptyset)$ be the corresponding instance with $V_1 \cup V_2 \cup V_3 = V$. Construct a partially directed graph $G' = (V \cup \{z\}, E', A')$ with

$$\begin{aligned} E' = &\{ \{u, v\} \mid \{u, v\} \in E \text{ with } u \in V_3 \text{ and } v \in V_2 \} \\ &\cup \{ \{u, v\} \mid \{u, v\} \notin E \text{ with } u \in V_3 \text{ and } v \in V_1 \} \\ &\cup \{ \{u, v\} \mid u \neq v \in V_3 \}; \end{aligned}$$

$$\begin{aligned} A' = &\{ (u, v) \mid \{u, v\} \in E \text{ with } u \in V_1 \text{ and } v \in V_2 \} \\ &\cup \{ (z, v) \mid v \in V_3 \}. \end{aligned}$$

We show that G contains a triangle if, and only if, G' is not extendable:

- $\Rightarrow$ Let $a, b, c \in V$ be a triangle in G with $a \in V_1$, $b \in V_2$, and $c \in V_3$. Then G' contains the induced subgraph $a \rightarrow b - c \leftarrow z$ and, hence, G' cannot be extended as either orientation of $b - c$ will create a new v-structure.
- $\Leftarrow$ Assume there is no triangle in G. We construct an extension $\vec{E}'$ of G'. There are three types of undirected edges $x - y$, which we direct as follows:
  1. $x \in V_1, y \in V_3$: Orient as $x \leftarrow y$.
  2. $x \in V_2, y \in V_3$: Orient as $x \leftarrow y$.
  3. $x \in V_3, y \in V_3$: Orient according to any linear ordering τ of V_3.

Table 1: Operations of a data structure to maintain partially directed graphs. Consecutive empty rows mean that the corresponding cell has the same content as the cell above (e. g., the same running time as the method in the row above).

Method	*Time*	*Effect*
init(n)	$O(1)$	Initializes a empty partially directed graph $G = (V = \{v_1, \dots, v_n\}, E = \emptyset, A = \emptyset)$.
is-adjacent(u, v)	$O(1)$	Returns true if $u \sim_G v$.
insert-edge(u, v) insert-arc(u, v) remove-edge(u, v) remove-arc(u, v)	$O(\Delta(u))$	Inserts or removes the edge $\{u, v\}$ or arc (u, v) to E or A.
next-edge(u, v) next-out-arc(u, v) next-in-arc(u, v)	$O(1)$	If $\{u, v\} \in E$ (or $(u, v) \in A$), this returns the "next" neighbor w of u. Next refers to no special order, but calling these methods multiple times will iterate over all neighbors of u. The method returns $\perp$ if all neighbors where traversed and it returns the "first" neighbor of u if called with u and $\perp$.
is-ps(s)	$O(1)$	Tests whether s is a potential-sink.
list-ps$()$	$O(n)$	Returns a list of all potential-sinks in G.
pop-ps(s)	$O(\delta(s)^2 + \delta(s) \cdot \delta^-(s))$	Orients all edges towards s and removes s from the graph. Returns a list of neighbors of s that did become potential-sinks due to this operation.

We show that this extension $\vec{E}'$ is acyclic and contains no new v-structure:

1. For the acyclicity, observe that we have $V_1 \to V_2 \leftarrow V_3 \leftarrow \{z\}$, with an arc $V_i \to V_j$ indicating that all edges between these vertex sets are oriented from $u \in V_i$ towards $v \in V_j$. The sets V_1 and V_2 are not adjacent to z. It is clear that any cycle would have to occur internally in a set V_i as a path $u \in V_i \to v \notin V_i \to \dots \to w \in V_i$ cannot exist. But only the set V_3 contains internal edges, which are topologically ordered. Thus, $\vec{E}'$ is acyclic.
2. In the following case study, we analyze all situations in which $a \to b \leftarrow c$ can occur in $\vec{E}'$, as this is a necessary condition for a new v-structure. We show that either $a \sim_{G'} c$ or the v-structure already existed in G'.
   (a) $a \in V_1, b \in V_2, c \in V_3$: A v-structure would be created if $a - c$ is not in G', but this implies a triangle in G.
   (b) $a \in V_1, b \in V_2, c \in V_1$: These v-structures are present in G'.
   (c) $a \in V_3, b \in V_1, c \in V_3$: We have $a \sim_{G'} c$ as V_3 is fully connected.
   (d) $a \in V_3, b \in V_2, c \in V_3$: Same as (c).
   (e) $a \in V_3, b \in V_3, c \in V_3$: Same as (c).
   (f) $a \in V_3, b \in V_3, c = z$: V_3 and z are fully connected, hence $a \sim_{G'} z$.
   (g) $a = z, b \in V_3, c \in V_3$: Symmetrical to (f). □

We remark that the same reduction also applies for EXT.

MISSING PROOFS OF SECTION 4: EXTENDING PARTIALLY DIRECTED GRAPHS

Our working horse for proving the theorems in this section is a novel data structure that maintains partially directed graphs and potential-sinks therein.

Lemma. *There is data structure to maintain partially directed graphs $G = (V, E, A)$ that uses $O(n^2)$ space and supports the operations listed in Table 1.*

Proof. Our data structure is based on the hybrid graph representation by Abu-Khzam et al. [2010]. To store a partially directed graph $G = (V, E, A)$, we maintain two directed graphs $G_1 = (V, E)$ and $G_2 = (V, A)$, where we insert undirected

edges as a double arc in G_1. Each G_i is represented by an $n \times n$ adjacency matrix M_i, two size-n arrays δ_i^- and δ_i^+ that store the in- and out-degrees of the vertices, and has for every vertex two size-n arrays N_i^v and I_i^v that contain the vertices to which v points and which point to v, respectively. Both arrays store these vertices as list, i. e., the first $\delta_i^+[v]$ elements of N_i^v are the neighbors of v (we use *brackets* to indicate array access, while we use *parentheses* for method calls and mathematical functions). The idea that allows to support all operations efficiently at the same time is that the adjacency matrix is not a binary matrix but rather a matrix that contains at index $M_i[u, v]$ either $\perp$ (if the arc (u, v) is not in the graph) or pointers to the positions in N_i^u and I_i^v at which u and v are stored, respectively.

In order to initialize the data structure in $O(1)$, we allocate the adjacency matrix and the arrays as uninitialized memory. We initialize the values using the standard trick of accessing uninitialized data via a stack of referencing pointers, which eliminates the setup time. Details of this technique are provided by Briggs and Torczon [1993] and can be found in standard textbooks (for instance [Aho et al., 1974, Chapter 2]).

Inserting an arc (u, v) to a G_i is easy: increment $\delta_i^+[u]$ and $\delta_i^-[v]$, set $N_i^v[\delta_i^+[u]] = v$, $I_i^v[\delta_i^-[v]] = u$, and let $M_i[u, v]$ point to these positions. To remove an arc (u, v) we do the opposite, but have to be careful to update the adjacency lists in constant time. To achieve this, we use the pointers of the adjacency matrix to locate v in N_i^v (in $O(1)$) and swap it with the element at position $\delta_i^+[u]$, then we decrement $\delta_i^+[u]$. We do the same in I_i^v and $\delta_i^-[v]$, and we set $M_i[u, v] = \perp$. We can check adjacencies in $O(1)$ and iterate over the outgoing (incoming) arcs of v in time $O(\delta_i^+[v])$ ($O(\delta_i^-[v])$), since we have M_i as well as the adjacency lists N_i^v and I_i^v.

As mentioned before, we represent the partially directed graph $G = (V, E, A)$ with two graphs G_1 and G_2. If we insert an edge $\{u, v\}$ to G, we insert the arcs (u, v) and (v, u) to G_1; if we insert the arc (u, v) to G, we simply insert it to G_2. Hence, we inherit all the insert, remove, and iterate methods claimed in Table 1. Observe that for a vertex $v \in V$ we have $\delta(v) = \delta_1^+[v]$, $\delta^+(v) = \delta_2^+[v]$, and $\delta^-(v) = \delta_2^-[v]$.

To realize list-ps and pop-ps, we mange two additional size-n arrays α and β that store an integer for each vertex:

$$\alpha[v] = |\{\, \{x, y\} \mid \{v, x\} \in E \wedge \{v, y\} \in E \wedge x \sim_G y \,\}|;$$
$$\beta[v] = |\{\, \{x, y\} \mid \{x, v\} \in E \wedge (y, v) \in A \wedge x \sim_G y \,\}|.$$

Initially, we have $\alpha[v] = \beta[v] = 0$ for all $v \in V$, as we do start with an edge-less graph. Whenever we insert an edge $\{u, v\}$ (or an arc (u, v)) to G, we update these values as follows: We iterate over the neighbors of u (that is, all vertices x with $u \sim_G x$) and check whether $x \sim_G v$ as well (i. e., we iterate over the common neighbors of u and v). If u, v, and x indeed constitute a triangle, all three vertices have a new edge in their neighborhood: x has the new edge $u \sim_G v$, u the new edge $x \sim_G v$, and v has $x \sim_G u$. Hence, we may *increase* the α- and β-values of all three vertices (of course, only if the edge directions match the definition of α and β). These updates require $O\big(\delta_1^+[u] + \delta_2^+[u] + \delta_2^-[u]\big)$ operations, which is why the data structure for G does not support edge and arc inserts in $O(1)$. Removing edges or arcs works analogously, we just have to *decrease* the corresponding α- and β-values.

Claim. *A vertex $s \in V$ is a potential-sink iff:*

1. $\alpha[s] = \binom{\delta_1^+[v]}{2}$;
2. $\beta[s] = \delta_1^+[s] \cdot \delta_2^-[s]$;
3. $\delta_2^+[s] = 0$.

Proof. Recall from Definition 4.2 that s is a potential-sink iff (i) $X = \{\, v \mid \{s, v\} \in E \,\}$ is a clique, (ii) $x \sim_G y$ for all $x \in X$ and $y \in Y = \{\, v \mid (v, s) \in A \,\}$, and (iii) $\{\, v \mid (s, v) \in A \,\}$ is empty. Obviously, item (iii) corresponds to the third item in the claim. For (i), we require that X is a clique, i. e., that it contains $\binom{|X|}{2}$ edges. Since X is the undirected neighborhood of s, we have $|X| = \delta_1^+[s]$ and, hence, X is a clique iff $\alpha[s] = \binom{\delta_1^+[v]}{2}$. Finally, for (ii) we require that X (the undirected neighborhood of s) and Y (the set of vertices pointing to s) form a complete bipartite graph, i. e., we need $|X| \cdot |Y|$ edges between these sets. Since $|Y| = \delta_2^-[s]$, this is the case iff item two in the claim holds. □

Since the data structure maintains all values used in the claim, we can check whether a given vertex s is a potential-sink with a constant number of arithmetic operations. To list all potential-sinks, we simply iterate over all vertices and output the ones that satisfy the claim.

The final operation we implement is pop-ps(s), i. e., we wish to direct all edges incident to s towards s and remove s from the graph. This removal is "virtual", i. e., the universe size of our data structure remains n and we just mark s

as "deleted." However, we *actually* delete all edges and arcs incident to s. The crucial part is to archive this in time $\delta(s)^2 + \delta(s) \cdot \delta^-(s) = \delta_1^+[s]^2 + \delta_1^+[s] \cdot \delta_2^-[s]$. (We cannot use our remove-edge method, as this would use $O(\delta_1^+[s] + \delta_2^-[s])$ time *for every* incident edge.)

To obtain the better bound, we observe that updating α and β is significantly simpler if we remove an arc (v, s) instead of removing an arbitrary edge. In fact, we can ask which vertex x may have (v, s) in its neighborhood counted for its $\alpha[x]$- or $\beta[x]$-value. Clearly, we must have $s \sim_G x$ and, since $\delta_2^+[s] = 0$, we have either $\{s, x\} \in E$ or $(x, s) \in A$. In the latter case, the edge (v, s) may be neither in the undirected neighborhood of x nor in the corresponding bipartite graph and, thus, the removal of (v, s) does not effect x. It may, however, effect x if $\{x, s\} \in E$. Therefore, to remove an arc (v, s), we iterate over the undirected neighbors of s (in $\delta(s)$) and for each such x we look-up (in constant time) the type of the edge (or arc) $\{x, v\}$ – note that $x \sim_G v$ as s is a potential-sink. Depending on the type of this edge, we update the α- or β-value of x and then delete the arc from the data structure (in constant time). Using this trick, we can remove all arcs (v, s) in time $O(\delta(s) \cdot \delta^-(s))$ from the graph.

To implement pop-ps(s) in $O(\delta(s)^2 + \delta(s) \cdot \delta^-(s))$, we simply remove the arcs (v, s) as described above *before* we remove the other edges incident to s. Once the arcs are gone, s has only undirected neighbors and, thus, we can remove the remaining edges in time $O(\delta(s)^2)$ using remove-edge.

We iterate one last time over the old neighbors of s and check whether they did become a potential-sink and, if so, output these vertices. □

Proof of Lemma 4.7. Given G in any format, we set up the data structure by simply inserting all edges and arcs to it. This clearly is possible in time $O(\Delta m)$. Then we use list-ps to obtain a list of potential-sinks and, as long as this list is not empty, we remove a potential-sink s with pop-ps(s). If this generates new potential-sinks, we add them to the list. By Fact 4.3, G is not extendable if the list depletes before G becomes edge-less. Observe that we only call list-ps once and that we pay $O(\delta(s)^2 + \delta(s) \cdot \delta^-(s))$ for every potential-sink by calling pop-ps(s). Observe that this operation removes the $\delta(s) + \delta^-(s)$ edges that are incident to s from the graph. Hence, the total running time of the algorithm is bounded by $O(\Delta m)$. □

We have already proven in the main text how this result can be improved from $O(\Delta m)$ to $O(dm)$ using Claim 4.8:

Proof of Claim 4.8. Since s is a potential-sink, its undirected neighborhood constitutes a clique in the skeleton of the input graph. However, a d-degenerate graph can contain no clique that is larger than $d + 1$, as every induced subgraph has to contain a vertex of degree at most d. □

This completes the proof of Theorem 4.6. It seems unlikely that we can improve the time further due to the lower bounds of Sec. 3. However, the algorithm of Theorem 4.6 as presented here requires space $O(n^2)$ and this space dependency may turn out to be the bottleneck on larger graphs. There are various ways of circumventing this issue in practice. Most naturally, we may implement the underlying adjacency matrices using hash tables, similar to previously presented algorithms [Dor and Tarsi, 1992, Chickering, 2002] that assume constant-time adjacency tests using only linear space, too. Then Theorem 4.6 can be stated as "an consistent extension can be computed in *expected* time $O(dm)$ using $O(n + m)$ space."

If neither expected time nor quadratic space is tolerable, we can still improve on the space requirements while investing only slightly more time. The following time-space trade-off lemma does the job:

Lemma. *There is an algorithm that decides whether a partially directed graph $G = (V, E, A)$ is extendable in time $O(d^2 m)$ and space $O(n + m)$. Here d is the degeneracy of the skeleton of G.*

Proof. We proceed as in the proof of Theorem 4.6 and only update the underlying data structure. The idea is to use the well-known trick of storing a d-degenerate graph in linear space such that we can perform adjacency tests in $O(d)$.

Before we run the algorithm that tests whether the graph is extendable, we compute a degeneracy ordering $\prec$ of the input graph and store for each vertex its at most d preceding neighbors (this is possible in linear time [Matula and Beck, 1983]). Note that we can check whether u and v are adjacent in $O(d)$ by taking the vertex that is larger with respect to $\prec$ and check whether the other vertex is one of its d smaller neighbors.

Given this auxiliary data, we implement the data structure with adjacency lists (in space $O(n + m)$). Whenever we perform an adjacency test in one of the operations, we do so, as described above, in $O(d)$. This clearly increases the total running time by at most a factor of d.

Algorithm 1: Extending a bucket in linear time.

```
INPUT:  bucket B = (V, E, A)
OUTPUT: AMO of B

S ← (V) and τ ← ()
while S is non-empty:
  // assume S = (S_1, S_2, ...)
  v ← any vertex in S_1 such that
    there is no u in a S_i with (u, v) ∈ A
  S ← (S_1 ∩ N(v), S_1 \ N(v), S_2 ∩ N(v), S_2 \ N(v), ...)
  τ ← τ + v
end
return orientation of B given by τ
```

Most operations carry over directly, for instance it is easy to insert an edge into an adjacency list in $O(1)$ if we keep sufficient pointers. However, removing an edge is difficult and would cost $O(\Delta)$ if done naively. We store the adjacency list as doubly-linked list and the edges $\{u, v\}$ as "objects" that are contained in both, the lists of u and v. In this way, we can still remove an edge in $O(1)$ if we are given the "object". This is sufficient, as in the proof of Theorem 4.6 we only have to remove edges (and arcs) whenever we process a potential-sink and *already* iterate over these edges and, thus, have pointers to the corresponding objects "at hand". □

MISSING PROOFS OF SECTION 5: A GRAPHICAL CHARACTERIZATION OF EXTENDABLE MPDAGS

Recall that a *bucket* in an MPDAG is an induced subgraph on the vertex set of an undirected connected component.

Proof of Lemma 5.2. Consider a v-structure $a \to b \leftarrow c$. For a, b, c to be in the same bucket, we assume w. l. o. g. that there is an undirected path between a and b not containing c: $b - p_1 - p_2 - \cdots - p_k - a$. Moreover, let a, b, c be chosen such that the path is shortest among all v-structures in the same bucket.

We have an edge between a and p_1 as well as c and p_1 (else G would not be an MPDAG due to R1). These edges cannot be directed $a \leftarrow p_1$ (or $p_1 \to b$, respectively) as otherwise R2 would apply. Moreover, the edges cannot both be undirected, as R3 would apply. Finally, setting one edge as $a \to p_1$ and the other as undirected would again imply R1 if a and c are nonadjacent (which they have to be to form a v-structure). Hence, we have $a \to p_1 \leftarrow c$. But then p_1 would also be in the same bucket as a, c and the path from a to p_1 is shorter than to b. A contradiction. □

Proof of Lemma 5.3. We show that by computing an AMO for every bucket we do neither create a v-structure nor a cycle.

Observe that there is no induced subgraph $a \to b - c$. Furthermore, any induced subgraph $a - b - c$ is in one bucket and, thus, gets oriented morally (i. e. without any v-structure) by assumption. Hence, there is no v-structure.

For the acyclicity, assume there is a cycle and consider the shortest one $c_1 \to c_2 \dots c_p \to c_1$. We assume that G itself does not contain a cycle, so there has to be some $c_i - c_{i+1}$ in G with $c_{i-1} \to c_i$ (if this edge would not exist, the whole cycle would be undirected in G and the vertices would be in the same bucket). We need to have an edge between c_{i-1} and c_{i+1} for $p \geq 4$, as G is an MPDAG. This edge needs to be oriented $c_{i-1} \to c_{i+1}$ or $c_{i-1} \leftarrow c_{i+1}$. Both cases imply a shorter cycle and, thus, a contradiction.

If the cycle has length three, i. e. $p = 3$, we simply distinguish three cases: If only one edge of the cycle is undirected, G is not an MPDAG; if two or three edges are undirected, all three vertices would be in the same bucket. □

Proof of Lemma 5.4. We prove the first direction constructively with a linear-time algorithm that produces an AMO of a bucket B with a chordal skeleton.

The algorithm proceeds as follows: A lexicographic BFS (LBFS) is performed on the skeleton of B. This produces the topological ordering of an AMO in linear time as the skeleton is chordal (for a discussion on this, see e. g. Corollary 40

in [Hauser and Bühlmann, 2012]). Additionally, the LBFS is modified such that in its traversal, the algorithm always chooses a vertex that has no incoming edges in B from unvisited vertices (see Algorithm 1). Hence, the produced ordering also entails the same directed edges as B.

It remains to prove that there is always such a vertex. Note that the LBFS always chooses a vertex from the first set of a set sequence $\mathcal{S} = (S_1, S_2, \dots)$. We prove by induction that the sought vertex exists in S_1.

This is clear at the start (when every vertex is in the first set), as otherwise every vertex would have an incoming edge and, thus, we would have a cycle (every acyclic graph and every induced subgraph of it has at least one source).

Now assume the first i vertices have been visited and there was always a vertex without an incoming edge in the first set. We show that there is again such a vertex in the first set. Note that not all vertices in the first set have an incoming edge from another vertex in this set (as this would again imply a cycle). Hence, there has to be at least one vertex x in the first set with incoming edges only from later sets. Let $x \leftarrow y$ be such an incoming edge. As y is in a later set, there has to be a previously visited neighbor z of x, which is not a neighbor of y (at some point, x was put before y in line 9, because a neighbor of x but not y was visited). As the algorithm has visited z before x we have:

1. either B contains the arc $z \rightarrow x$, which would be a contradiction by Lemma 5.2 as this would be a v-structure in a bucket
2. or B contains the edge $z - x$, which would imply the induced subgraph $z - x \leftarrow y$ contradicting the fact that G is an MPDAG.

By induction hypothesis, the arc $z \leftarrow x$ cannot be in B.

Clearly, the algorithm can be implemented in linear time. For the adapted LBFS, note that each set can be split into two parts, the vertices that already have no incoming edge and the rest. These parts can be efficiently maintained, for example by having a counter of incoming edges for each vertex.

For the reverse direction, note that for a non-chordal skeleton there cannot exist an AMO. □

We close this section by recalling the known linear-time algorithms for extending CPDAGs and Chain Graphs. As CPDAGs are a special case of MPDAGs (with the buckets always being undirected and chordal), a simpler version of the algorithm proposed above can be used: Simply orient the edges of the undirected components according to a topological ordering obtained from an LBFS. This is described in more detail in [Hauser and Bühlmann, 2012], see for example Proposition 16.

An algorithm for the extension of Chain Graphs based on a modified Maximum Cardinality Search (MCS) is given in [Andersson et al., 1997]. There is no run time analysis, but one can easily see that an implementation similar to the standard MCS yields linear-time.

MISSING PROOFS OF SECTION 6: RECOGNITION OF CAUSAL GRAPH CLASSES

Proof of Observation 6.2. For PDAG-REC, we only have to check that the input does not contain a directed cycle.

For CPDAG-REC the algorithm proceeds as follows (input is a partially directed graph G):

1. Orient the undirected components of G by performing the LBFS algorithm for each and directing edges according to the traversal order. Let the resulting graph be D.
2. If D contains a cycle, output "No". Otherwise find the CPDAG corresponding to D (let this be C) and output "Yes" if $C = G$ and "No" otherwise.

Step 2 can be performed in $O(n + m)$ as finding the CPDAG representing the Markov equivalence class of a DAG is possible in linear time [Chickering, 1995]. Hence, the whole algorithm runs in time $O(n + m)$.

If G is a CPDAG, it is well-known that D is a consistent extension [Hauser and Bühlmann, 2012]. Hence $C = G$. If G is not a CPDAG, either D contains a cycle or the graph C is computed, which is by definition a CPDAG. Hence, $C \neq G$.

Finally, Chain Graphs can be recognized in linear time as well. We use the following contraction based algorithm: As long as the input graph contains an undirected edge, we contract it to an arbitrary neighbor. Once all undirected edges are gone, we simply check whether the remaining graph is acyclic.

Algorithm 2: Maximal orientation of an extendable PDAG.

```
INPUT:  Extendable PDAG G
OUTPUT: Maximal orientation of G

D ← consistent extension of G
π ← topological ordering of D
// π^{-1}(v) denotes the position of v in π
C ← CPDAG of D

// the following modifications are
// performed in place in C
for each undirected component U of C:
  copy edge directions from G to U
  for v ∈ V_u in order π:
    // R1
    for p − v with π^{-1}(p) < π^{-1}(v):
      if ∃a s.t. a → p − v:
            orient p → v
      end
    end
    // R4
    for p → v with π^{-1}(p) < π^{-1}(v):
      if ∃d, a s.t. d → p → v and a — v ← p
        orient a → v
      end
    end
    // R2
    for p − v with π^{-1}(p) < π^{-1}(v)
        in decr. π^{-1}(p):
      if ∃b s.t. p → b → v :
        orient p → v
      end
    end
  end
end
return C
```

If the input contains a semi-directed cycle, the undirected parts of the cycle get contracted and the resulting graph contains a directed cycle. On the other hand, any directed cycle in the contracted graph corresponds to a cycle in the original graph with at least one directed edge. □

MISSING PROOFS OF SECTION 7: APPLICATION TO MAXIMAL ORIENTATIONS OF PDAGS

Proof of Theorem 7.1. MAXIMALLY-ORIENT can be solved with Algorithm 2. It applies the Meek rules starting with CPDAG C instead of PDAG G. Such an approach is valid, see for example the proof of Theorem 4 in Meek [1995].

Moreover, it is sufficient to apply the Meek rules inside the undirected components of C (with the directed edges from G copied over). This is because a vertex outside the undirected component cannot take part in the rules R1-R4. The only such vertices could potentially be a in R1 and b in R2, but clearly the rules cannot be applied in C immediately and directing further edges will not create such a situation. Therefore, we obtain the following problem. Given a partially directed graph H with chordal skeleton and no v-structures, and a topological ordering π of a consistent extension, exhaustively apply R1-R4. Note that R3 cannot occur as it contains a v-structure.

The algorithm visits the vertices in topological order. We prove by induction that after handling vertex v, the induced subgraph on all visited vertices coincides with this induced subgraph in M. After visiting the first vertex, the proposition above holds as the induced subgraph contains no edge. Now assume that the statement holds until the vertex u is visited immediately before v. Then, all edges going into any *visited* vertex except v have already been set correctly by induction hypothesis. To complete the proof, we only have to show that every edge into v is correctly oriented or left undirected. Note that in case we direct the edge, only the orientation consistent with π is valid.

It is easy to see that it is sufficient to consider vertices coming before v in π for Meek rule detection. The correctness of the fast Meek rule detection was already argued in the proof of Theorem 6.1. However, we have to be careful due to the fact that applying a rule to some edge $p_1 - v$ may lead to the orientation of $p_2 - v$ at some later point. Hence, we have to make sure that edges $p_2 - v$ do not have to be rechecked for applicability of R1-R4 repeatedly.

We can see that rechecking for R1 is not necessary as the edge incident to v has to be *undirected* (vertex v corresponds to vertex c in R1, as we want to find new directed edges into v). In R4 (here v corresponds to vertex b) there is such a directed edge $c \to b$ incident to v, but this edge always follows from R1 (because of $d \to c$). Hence, by first exhaustively applying R1 and afterward R4, these rules are handled correctly. As R3 does not apply, R2 remains. Here, we consider the edges $p - v$ with decreasing $\pi^{-1}(p)$, i. e., the closest p first. When considering p, we hence know that for all p' with $\pi^{-1}(p) < \pi^{-1}(p') < \pi^{-1}(v)$, R2 has already been applied. Thus, when searching for $p \to p' \to v$ we know that $p' \to v$ has already been directed if it is directed in M.

It is immediately clear that the algorithm runs in time $O(\Delta m)$. To see that it is actually $O(dm)$, observe that we only consider parents of vertices in D. As we are in an undirected component without any v-structures, the parents have to be fully connected, i. e., form a clique together with v. But d-degenerate graphs contain cliques with at most $d + 1$ vertices. □

References

Faisal N. Abu-Khzam, Michael A. Langston, Amer E. Mouawad, and Clinton P. Nolan. A hybrid graph representation for recursive backtracking algorithms. In *FAW*, volume 6213 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2010.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

Steen A Andersson, David Madigan, and Michael D Perlman. On the Markov equivalence of chain graphs, undirected graphs, and acyclic digraphs. *Scandinavian Journal of Statistics*, 24(1):81–102, 1997.

Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

David Maxwell Chickering. A transformational characterization of equivalent Bayesian network structures. In *Proc. of the Conference on Uncertainty in Artificial Intelligence, UAI 1995*, pages 87–98, 1995.

David Maxwell Chickering. Learning equivalence classes of Bayesian-network structures. *Journal of Machine Learning Research*, 2:445–498, 2002.

Dorit Dor and Michael Tarsi. A simple algorithm to construct a consistent extension of a partially oriented graph. *Technicial Report R-185, Cognitive Systems Laboratory, UCLA*, 1992.

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.

David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3): 417–427, 1983.

Christopher Meek. Causal inference and causal explanation with background knowledge. In *Proc. of the Conference on Uncertainty in Artificial Intelligence, UAI 1995*, pages 403–410, 1995.