UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Experimental Analysis of Algorithms for Extending Causal Graphs

Experimentelle Analyse von Algorithmen zum Erweitern kausaler Graphen

Masterarbeit

verfasst am
Institut für Theoretische Informatik

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Malte Luttermann

ausgegeben und betreut von
Prof. Dr. Maciej Liśkiewicz

mit Unterstützung von
Marcel Wienöbst und Prof. Dr. Ralf Möller

Lübeck, den 6. September 2021

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

Malte Luttermann

Zusammenfassung

Die Analyse kausaler Zusammenhänge ist eine wichtige Aufgabe in vielen Forschungsbereichen. Um kausale Zusammenhänge innerhalb von beobachteten Daten zu untersuchen, werden häufig kausale Graphen verwendet. Eine fundamentale Aufgabe bei der Analyse kausaler Zusammenhänge mittels kausaler Graphen ist das Erweitern eines partiell gerichteten azyklischen Graphen zu einer sogenannten konsistenten DAG-Erweiterung.

Diese Arbeit vergleicht Algorithmen zum Erweitern von partiell gerichteten azyklischen Graphen aus praktischer Sicht. Es wird eine neue Heuristik für den wohlbekannten Algorithmus von Dor und Tarsi (1992) vorgeschlagen, welche die Performance des ursprünglichen Algorithmus in der Praxis verbessert. Darüber hinaus geben wir einen Algorithmus für die Aufzählung von konsistenten DAG-Erweiterungen für CPDAGs an, dessen Verzögerung zwischen zwei aufeinanderfolgenden Ausgaben durch $O(nm)$ begrenzt werden kann, wobei n die Anzahl der Knoten und m die Anzahl der Kanten in dem CPDAG angibt.

Abstract

The analysis of causal relationships is an important task in many research areas. To investigate causal relationships within observed data, causal graphs are widely used. A fundamental task when analyzing causal relationships with the help of causal graphs is extending a partially directed acyclic graph to a so-called consistent DAG extension.

This thesis compares algorithms for extending partially directed acyclic graphs from a practical point of view. A new heuristic for the well-known algorithm given by Dor and Tarsi (1992) is proposed which improves the performance of the original algorithm in practice. Furthermore, we give an algorithm for the enumeration of consistent DAG extensions for CPDAGs whose delay between two consecutive outputs can be bounded by $O(nm)$ where n is the number of vertices and m is the number of edges in the CPDAG.

Contents

1

Introduction

Causal relationships are of great interest in many research areas such as epidemiology, social science, and economics (Greenland, Pearl, and Robins, 1999; Morgan and Winship, 2014; Chalak and White, 2007). For the investigation of causal relationships, randomized controlled experiments are widely used. Conducting a randomized controlled experiment, however, might be impossible, too expensive, or not acceptable for ethical reasons in some scenarios. Thus, it is highly desirable to identify causal relationships from observed data.

In order to model the underlying causal relationships of observed data, causal graphs are widely used (Pearl, 2009). A causal graph is a graph in which the vertices represent random variables of the system being modeled and causal relationships between those random variables are represented by the edges of the causal graph. More precisely, we consider causal graphs to be directed acyclic graphs (DAGs) where each directed edge $X \to Y$ between two vertices (i.e., random variables) X and Y describes that X is a direct cause of Y (and equivalently, that Y is a direct effect of X).

However, obtaining a causal graph that correctly models the underlying causal relationships of the modeled system is a challenging task. As causal graphs may become highly complex, it is often not possible for humans to specify a causal graph by hand and hence the question arises whether causal graphs are learnable from observed data.

Algorithms addressing the problem of learning a causal graph from observed data are already available. Widespread algorithms for learning causal graphs from observed data include the so-called IC algorithm (Verma and Pearl, 1990), the PC algorithm (Spirtes, Glymour, and Scheines, 1993), and the GES algorithm (Chickering, 2003).

When learning a causal graph from observed data, the observed data may provide only incomplete information about the underlying causal relationships of the data. Consequently, there might be several possible causal explanations for the observed data, i.e., there might be different causal graphs explaining the observed data.

Thus, algorithms extracting causal relationships from observed data are not always able to learn an explicit causal DAG, but instead learn a representation for all DAGs that are consistent with the data, i.e., all DAGs that possibly explain the underlying causal relationships of the data. The representation of the set containing all DAGs that are consistent with the observed data is a partially directed acyclic graph (PDAG). A PDAG is a graph that contains both directed and undirected edges but no directed cycle.

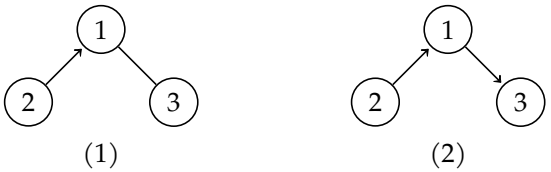


Figure 1.1: Graph (1) shows an exemplary PDAG representing a set of DAGs that contains exactly one DAG. The DAG represented by graph (1) is given by graph (2). Note that the edge $1 - 3$ cannot be oriented as $1 \leftarrow 3$ as this creates the new v-structure $2 \rightarrow 1 \leftarrow 3$.

Many applications in the field of analyzing causal relationships, however, require a specific DAG from the represented set of DAGs instead of the PDAG representation itself (Greenland, Pearl, and Robins, 1999; Williams et al., 2018; Tennant et al., 2020). In order to obtain a specific DAG from the set of DAGs consistent with the observed data, the learned PDAG has to be *extended*. A PDAG G might be extended to obtain a DAG from the set of DAGs represented by G by orienting all undirected edges in G while obeying the following rules.

1. All undirected edges must be oriented in a way such that the resulting directed graph is acyclic.
2. Orienting the undirected edges must not create new v-structures. A v-structure consists of three vertices X, Y, Z with $X \rightarrow Y \leftarrow Z$ and no edge between X and Z.

We call a DAG resulting from orienting the undirected edges of a PDAG G a consistent extension of G if the DAG satisfies the rules given above. In other words, all DAGs in the set of DAGs represented by G are consistent extensions of G. Note that a PDAG G might not admit a consistent extension, i.e., the set of DAGs represented by G can be empty. Figure 1.1 shows an example for a PDAG and the set of DAGs represented by it.

Finding a consistent DAG extension or even all consistent DAG extensions for a PDAG are fundamental tasks in the field of analyzing causal relationships. The computation of consistent DAG extensions is required by algorithms for learning causal graphs from observational data (Meek, 1995; Spirtes, Glymour, and Scheines, 1993; Chickering, 2003) and is frequently applied in software packages for analyzing causal relationships (Scutari, 2010; Kalisch et al., 2012; Textor et al., 2017). This thesis investigates algorithms for computing consistent DAG extensions of PDAGs from a practical point of view.

1.1 Previous Work

About three decades ago, Verma and Pearl (1992) presented an algorithm for deciding whether there is a causal explanation for observed data, i.e., whether a causal DAG consistent with the observed data can be found. The algorithm by Verma and Pearl first constructs a PDAG from the observed data (if possible) and then extends the obtained PDAG to a consistent DAG extension. The worst-case time complexity for the computation of the consistent DAG extension can be bounded by $O(n^4 m)$, where n denotes the number of vertices and m the number of edges in the input PDAG.

In the same year, Dor and Tarsi (1992) proposed an algorithm running in time $O(n^4)$ to solve the problem of finding a consistent DAG extension for a given PDAG[1]. For graphs with a bounded degree Δ, the algorithm runs in time $O(\Delta^2 m)$. The algorithm given by Dor and Tarsi became widely used and was the best known algorithm with respect to the worst-case time complexity for computing consistent DAG extensions of a PDAG until recently. Furthermore, Dor and Tarsi conjectured that there might be an algorithm for finding consistent DAG extensions in linear time (Dor and Tarsi, 1992).

However, nearly 30 years later, Wienöbst, Bannach, and Liśkiewicz (2021a) showed that under certain intractability assumptions no linear-time algorithm can exist. Simultaneously, the authors proposed a new algorithm for computing a consistent DAG extension of a PDAG in time $O(n^3)$ or $O(\Delta m)$ for graphs with a bounded degree Δ. Besides this new algorithm with worst-case time complexity $O(\Delta m)$, an improved version of it for the class of d-degenerate graphs has been presented as well. The improved version runs in time $O(dm)$ for d-degenerate input graphs. As a consequence, the improved algorithm runs in linear time on graphs with a constant degeneracy d such as forests or planar graphs.

Although Wienöbst, Bannach, and Liśkiewicz (2021a) provide a comprehensive theoretical analysis of their proposed algorithms to compute a consistent DAG extension, no experimental evaluation was conducted to verify the practicality of the algorithms. The algorithm presented by Dor and Tarsi has not been evaluated experimentally as well and thus it is highly desirable to create a framework for the comparison of algorithms solving the problem of finding consistent DAG extensions.

Considering the task to enumerate all consistent DAG extensions of a given PDAG, Meek (1995) outlined an algorithm solving this task in time $O(2^m \Delta m^3)$. Furthermore, algorithms counting the number of consistent DAG extensions for a specific subclass of PDAGs, namely completed partially directed acyclic graphs (CPDAGs), are already available (Talvitie and Koivisto, 2019; Wienöbst, Bannach, and Liśkiewicz, 2021b).

1.2 Contributions of this Thesis

The main contributions of this thesis are as follows.

Firstly, a framework for the experimental evaluation of various algorithms in the context of extending PDAGs is provided. This framework contains efficient implementations of five algorithms for computing consistent DAG extensions for PDAGs and more than 1 000 input graph instances. The algorithms for computing consistent DAG extensions include the algorithms by Dor and Tarsi and by Wienöbst, Bannach, and Liśkiewicz as well as two newly proposed modifications of the algorithm by Dor and Tarsi. The framework for the experimental evaluation also provides different graph generators allowing for an enlargement of the input graph instances in the dataset. Those graph generators are able to generate directed graphs, undirected graphs, and partially directed graphs with arbitrary proportions of directed edges. All of the implemented algorithms for computing consistent DAG extensions are evaluated extensively on all available input graph instances from the generated dataset, providing an extensive experimental evaluation that

[1] In the original paper, the runtime is stated as $O(nm)$, i.e., $O(n^3)$ in the worst case. Nevertheless, this claim was shown to be incorrect later on by Chickering (2002).

has been missing in the past.

Secondly, two approaches for optimizing the performance of the algorithm proposed by Dor and Tarsi are presented. Both approaches surpass the performance of the implementation given in the original paper by Dor and Tarsi on many evaluated input graph instances, yielding efficient algorithms with a sound practical performance. One approach applies a heuristic while the other approach uses an additional pre-computation step.

Lastly, the problem of enumerating all consistent DAG extensions for a given PDAG is investigated in-depth. Although Meek (1995) outlined an algorithm to enumerate all consistent DAG extensions for a PDAG, the algorithm was neither fully described nor was its runtime analyzed. In this thesis, the idea for the enumeration algorithm by Meek is described precisely and the runtime of the algorithm is analyzed. Furthermore, we derive an algorithm that significantly reduces the delay between the output of two consecutive consistent DAG extensions when enumerating all consistent DAG extensions of a CPDAG.

1.3 Structure of this Thesis

Besides the introduction, this thesis is structured as follows.

Chapter 2 gives an overview on theoretical backgrounds necessary for the subsequent parts of this thesis. That is, basic graph-related definitions and notations are introduced and important classes of graphs are presented. The problems of computing a consistent DAG extension and enumerating all consistent DAG extensions for a given input graph are defined formally and existing algorithms solving these problems are described.

In Chapter 3, we first discuss the framework for the experimental evaluation, i.e., we examine the applied graph generation approaches used to generate the dataset of input graph instances and investigate the implementations of the algorithms for computing consistent DAG extensions. Afterwards, we solve the problem of enumerating all consistent DAG extensions for a given input graph.

Chapter 4 presents the results of the experimental evaluation. We review the setup of the evaluation, take a look at the obtained results, and give an interpretation of the results.

2

Preliminaries

This chapter gives an overview on theoretical backgrounds necessary for the subsequent parts of this thesis. Basic definitions and notations are introduced, the problems of computing and enumerating consistent DAG extensions are formally defined, and existing algorithms solving these problems are presented.

2.1 Graph Fundamentals

We begin by introducing some basic graphical definitions and notations. A partially directed graph $G = (V, E, A)$ consists of a set of vertices V (also called nodes), a set of undirected edges E, and a set of directed edges A (also called arcs). The number of vertices and edges are denoted as $n = |V|$ and $m = |E| + |A|$, respectively. For any graph G, we say that G is an empty graph if $m = 0$. We demand that there is at most one edge between any two vertices which is either oriented into a single direction or undirected. That is, for each pair of vertices $u, v \in V$, we have either $\{u, v\} \in E$, $(u, v) \in A$, $(v, u) \in A$, or none of those, but never more than one of those at the same time.

Two vertices are called adjacent if there is either an arc or an undirected edge between them. For two vertices $u, v \in V$ that are adjacent in a graph G, we write $u \sim_G v$ or simply $u \sim v$ if the context is clear. All vertices being adjacent to a vertex v are called the neighbors of v. We differentiate between ingoing, outgoing, and undirected neighbors for a vertex $v \in V$: $\delta^-(v) = |\{u \mid (u, v) \in A\}|$ is the number of ingoing neighbors of v, $\delta^+(v) = |\{u \mid (v, u) \in A\}|$ describes the number of outgoing neighbors of v, and $\delta(v) = |\{u \mid \{u, v\} \in E\}|$ represents the number of undirected neighbors of v. The degree of a vertex $v \in V$ is the number of its neighbors and written as $\Delta(v) = \delta(v)^- + \delta(v)^+ + \delta(v)$. We denote the maximum degree of a graph G as $\Delta(G) = \max_{v \in V} \Delta(v)$. If the context is clear, we write Δ instead of $\Delta(G)$.

A connected graph is a graph where each vertex is connected to each other vertex via a sequence of edges or arcs, regardless of the orientations of those arcs. For a graph $G = (V, E, A)$, we define a path as a sequence of distinct vertices $(v_1, v_2, \dots, v_l)$ with $\{v_i, v_{i+1}\} \in E$ or $(v_i, v_{i+1}) \in A$ for all $i \in \{1, 2, \dots, l-1\}$. A cycle is a sequence of distinct vertices $(v_1, v_2, \dots, v_l)$ with $l \geq 3$ and edges between v_i and v_{i+1} for all $i \in \{1, 2, \dots, l-1\}$, as well as an edge between v_l and v_1. An undirected cycle consists of undirected edges only, a

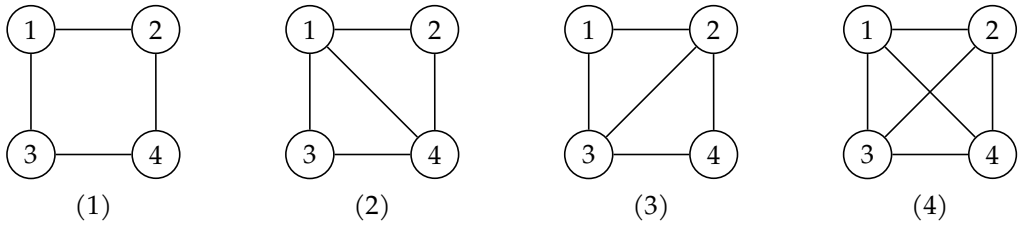


Figure 2.1: Graph (1) is the only non-chordal graph as it contains an induced cycle of length four. The graphs (2), (3), and (4) do not contain an induced cycle of length four or greater because there are additional edges between the four vertices. Consequently, the graphs (2), (3), and (4) are chordal.

semi-directed cycle contains at least one directed edge as well as at least one undirected edge, and a cycle is called directed if all edges belonging to the cycle are directed. We call an undirected graph chordal if no set of four or more vertices induces a cycle. A cycle is induced if there are no additional edges between its vertices in the original graph, as for example in graph (1) of Figure 2.1.

Three vertices $u, v, w \in V$ with $u \to v \leftarrow w$ are said to form a v-structure if u and w are not adjacent. The skeleton of a graph G is the undirected graph resulting from replacing each arc $u \to v$ or $u \leftarrow v$ in G with an undirected edge $u - v$. An induced subgraph of a graph G on a set of vertices $V' \subseteq V$ is written as $G[V']$. $G[V']$ contains all edges $u - v$ and arcs $u \to v$ for which $u \in V'$ and $v \in V'$. For any partially directed graph $G = (V, E, A)$, we call a vertex $v \in V$ a potential sink if the following two properties are satisfied:

1. The vertex v has no outgoing arc in G, i.e., there is no vertex u such that $(v, u) \in A$.
2. All undirected neighbors of v are adjacent to every other neighbor of v.

Then, a potential sink elimination order $\tau = v_1, v_2, \dots, v_n$ for a graph $G = (V, E, A)$ is a vertex ordering such that every vertex v is a potential sink in the induced subgraph $G[V \setminus \{u \mid \tau(u) < \tau(v)\}]$ where $\tau(v)$ gives the position of v in τ (Wienöbst, Bannach, and Liśkiewicz, 2021a). A perfect elimination ordering (PEO) for an undirected graph is a vertex ordering $\tau = v_1, v_2, \dots, v_n$ such that for every vertex v all preceding neighbors of v in τ form a clique. A clique is a subset of vertices where each pair of distinct vertices is adjacent to each other. As observed by Fulkerson and Gross (1965), a graph is chordal if and only if it has a PEO.

We call a graph d-degenerate if there is a vertex ordering $\tau = v_1, v_2, \dots, v_n$ of its vertices such that every vertex v_i has at most d neighbors in $v_1, v_2, \dots, v_{i-1}$ (Wienöbst, Bannach, and Liśkiewicz, 2021a). Such a vertex ordering τ is also called a degeneracy ordering and the degeneracy of a graph G is the smallest value d for which G is d-degenerate. Intuitively speaking, the degeneracy of a graph is a measure for its sparseness. Sparse graphs are graphs that have only a small number of edges compared to the maximum possible number of edges.

2.2 Extensions for Partially Directed Acyclic Graphs

Finding consistent DAG extensions for partially directed acyclic graphs (PDAGs) is a fundamental task in the area of causal analysis with the help of causal graphs. A DAG is a directed acyclic graph, i.e., a graph that has only directed edges and contains no cycle while a PDAG is graph that might contain both directed and undirected edges but no directed cycle. As we have seen before, learning causal graphs from observed data may result in a graph containing both directed and undirected edges because there might be multiple DAGs that possibly explain the underlying causal relationships of the data.

In some applications in the area of causal analysis, however, only DAGs are of practical use (Greenland, Pearl, and Robins, 1999; Williams et al., 2018; Tennant et al., 2020). Thus, computing a consistent DAG extension for a given PDAG is a fundamental task in the context of dealing with causal graphs. Furthermore, deciding whether a causal graph is extendable yields an answer to the question whether observed data have a causal explanation at all (Verma and Pearl, 1992).

A consistent extension of a PDAG G is a DAG G' which consists of the same set of vertices as G and has the same skeleton and v-structures as G. Moreover, for every arc $u \to v$ in G, $u \to v$ is in G' as well. That is, the undirected edges in G are oriented in a way such that neither cycles nor new v-structures arise. We call a PDAG extendable if it admits at least one consistent DAG extension.

With this being said, we define the problem EXTENDABILITY for partially directed graph inputs according to (Wienöbst, Bannach, and Liśkiewicz, 2021a) below. Note that EXTENDABILITY can be defined in the same way for more specific types of input graphs such as completed PDAGs and maximally oriented PDAGs which are introduced in Section 2.3.

Problem 2.1 (EXTENDABILITY).
Input: A partially directed graph G.
Output: A consistent DAG extension of G if G is extendable, else a negative answer.

In the following, we investigate which graphs belong to the class of extendable graphs and give criteria to decide whether a graph is extendable or not. Moreover, algorithms using the presented criteria to compute consistent DAG extensions for partially directed graphs are introduced.

Apparently, every extendable graph has to be acyclic and Dor and Tarsi observed that every partially directed graph that is extendable contains a potential sink (Dor and Tarsi, 1992). Furthermore, in order for a partially directed graph to be extendable, it must admit a potential sink elimination order.

Fact 2.2 (Wienöbst, Bannach, and Liśkiewicz, 2021a). Any partially directed graph is extendable if and only if it admits a potential sink elimination order.

Given a potential sink elimination order $\tau = v_1, v_2, \dots, v_n$ for a partially directed graph G, a consistent DAG extension of G can be computed by iterating over τ and orienting all undirected edges incident to the current v_i towards v_i for $i \in \{1, 2, \dots, n\}$.

Inspired by the observation that a potential sink elimination order yields a consistent DAG extension, Dor and Tarsi proposed an algorithm for computing a consistent DAG

Listing 2.2: Implementation of the algorithm DT according to the pseudocode given by Dor and Tarsi (1992). The algorithm DT searches for a potential sink by iterating over all vertices in the input graph repeatedly.

Input: *A partially directed graph* $G = (V, E, A)$
Output: *A consistent DAG extension of G if G admits such an extension, else a negative answer*

result ← *copy of G*
temp ← *copy of G*
while *there are vertices left in temp* ***do***
  s ← *any vertex in temp that satisfies the following properties*:
    1. *s has no arc directed outward from s*
    2. *all undirected neighbors of s are adjacent to all other neighbors of s*

  if *no vertex s satisfying the properties was found* ***then***
    return *a negative answer*
  end

  orient *all undirected edges incident to s towards s in result*
  remove *s and all incident edges from temp*
end
return *result*

---

extension of a partially directed graph (Dor and Tarsi, 1992). For convenience, we denote the algorithm by Dor and Tarsi as algorithm DT.

The algorithm DT initializes an output graph as a copy of the input graph first and then iterates over the vertices of the input graph. In each iteration, the algorithm DT checks for each vertex s whether s is a potential sink, i.e., whether s has no outgoing arc and all undirected neighbors of s are adjacent to all other neighbors of s. After a potential sink s has been found, all undirected edges incident to s are oriented towards s in the output graph. Subsequently, the vertex s is removed from the input graph and the algorithm repeats the steps above until no more vertices are left in the input graph or until no potential sink can be found.

If no potential sink can be found while there are still vertices left in the input graph, the input graph is not extendable because it does not admit a potential sink elimination order and the algorithm DT returns a negative answer. Listing 2.2 gives the exact steps undertaken by the algorithm DT in order to solve the problem EXTENDABILITY.

Looking at the worst-case time complexity of the algorithm DT, we can make the following observations. Checking whether any vertex v is a potential sink can be done in time $O(n^2)$ because we have to verify that all undirected neighbors of v are adjacent to all other neighbors of v. In the worst case, we have to check for every vertex in the graph whether it is a potential sink in each iteration of the outer loop and hence we need $O(n^2)$ checks in the worst case. Thus, the worst-case time complexity of algorithm DT is $O(n^4)$.

As we have seen, the algorithm DT searches for a potential sink by repeatedly iterating over the neighborhoods of the vertices. The algorithm presented by Wienöbst, Bannach, and Liśkiewicz (2021a) reduces the runtime for checking whether any vertex is a potential sink by storing additional information about the neighborhood of each vertex. We refer to the algorithm given by Wienöbst, Bannach, and Liśkiewicz as algorithm WBL.

A novel data structure is used by the algorithm WBL in order to reduce the cost of searching for a potential sink. The data structure allows to check whether any vertex is a potential sink in time $O(1)$ by using a pre-computation step to store additional information about the neighborhood of each vertex which is then kept up-to-date every time an edge is inserted or removed from the graph.

More precisely, the data structure represents a partially directed graph $G = (V, E, A)$ with the help of a hybrid graph representation that utilizes two directed graphs G_1 and G_2 with the same set of vertices V (Abu-Khzam et al., 2010). Each undirected edge $\{u, v\} \in E$ is stored as two arcs (u, v) and (v, u) in G_1 and each arc $(u, v) \in A$ is stored in G_2. Every graph G_i holds two arrays δ_i^+ and δ_i^- of size n where δ_i^+ stores the out-degree for each vertex and δ_i^- stores the in-degree for each vertex. That is, for any vertex $v \in V$, we have $\delta^+(v) = \delta_2^+[v]$, $\delta^-(v) = \delta_2^-[v]$, and $\delta(v) = \delta_1^+[v]$.

Furthermore, the data structure keeps track of another two arrays α and β of size n, storing information about the neighborhood of each vertex. The arrays α and β store an integer for each vertex:

$$\alpha[v] = |\{\{u, w\} \mid \{v, u\} \in E \wedge \{v, w\} \in E \wedge u \sim_G w\}|$$
$$\beta[v] = |\{\{u, w\} \mid \{v, u\} \in E \wedge (w, v) \in A \wedge u \sim_G w\}|$$

The α- and β-values must be updated whenever an edge is inserted or removed and hence inserting or removing an edge costs time $O(\Delta)$ because we have to iterate over the neighborhood of one endpoint of the edge. Checking whether any vertex $s \in V$ is a potential sink can now be done in time $O(1)$ by utilizing the following observation.

Fact 2.3 (Wienöbst, Bannach, and Liśkiewicz, 2021a). A vertex $s \in V$ is a potential sink in a partially directed graph $G = (V, E, A)$ if and only if

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

The algorithm WBL now begins by inserting all edges from the input graph into the data structure and afterwards computes a list of potential sinks L in a pre-computation step. Whenever an edge is inserted into the data structure or removed from the data structure, the α- and β-values are updated accordingly.

After computing a list of potential sinks L, the algorithm WBL iterates over L and removes a potential sink s from L in each iteration. Every time a potential sink s is removed from L, all undirected edges incident to s are oriented towards s in the output graph. Then, s is removed from the input graph and all neighbors of s that became a potential sink in the input graph are added to L. Listing 2.3 shows the algorithm WBL in detail.

Considering the runtime of the algorithm WBL, we observe that the setup of the data structure can be accomplished in time $O(\Delta m)$. Inserting an edge $\{u, v\}$ or an arc (u, v)

Listing 2.3: Implementation of the algorithm WBL as introduced by Wienöbst, Bannach, and Liśkiewicz (2021a). The algorithm WBL uses a novel data structure that stores additional information about the neighborhood of each vertex in order to reduce the cost for searching a potential sink.

Input: *A partially directed graph $G = (V, E, A)$*
Output: *A consistent DAG extension of G if G admits such an extension, else a negative answer*

result ← *copy of G*
temp ← *empty initialization of the data structure*
foreach *edge e* ***in*** *G* ***do***
  insert *e into temp and update the data structure accordingly*
end

L ← *list of potential sinks in temp*
while *L is not empty* ***do***
  s ← *pop any element from L*
  orient *all undirected edges incident to s towards s in result*
  remove *s and all incident edges from temp and update the data structure accordingly*
  add *all previous neighbors of s to L that became a potential sink in temp*
end
if *there are edges left in temp* ***then***
  return *a negative answer*
end
return *result*

---

into the data structure costs $O(\Delta)$ because we iterate over the neighborhood of either u or v. Thus, inserting all edges costs $O(\Delta m)$, which also gives the runtime for the setup of the data structure as computing the list L of all potential sinks is possible in time $O(n)$ by checking for every vertex whether it is a potential sink in time $O(1)$.

After the setup, potential sinks are repeatedly removed and the costly part is the update of the α- and β-values for the neighborhood of the removed vertex. As any edge removal can be done in time $O(\Delta)$ and all edges of the input graph are removed in the worst case, the loop runs in time $O(\Delta m)$. Hence, the worst-case time complexity of the entire algorithm WBL can be bounded by $O(\Delta m)$.

The worst-case time complexity of $O(\Delta m)$ can be improved even further for the class of d-degenerate graphs, as shown by Wienöbst, Bannach, and Liśkiewicz (2021a). The improved algorithm consists mostly of the same steps as the algorithm WBL but additionally makes use of a degeneracy ordering and hence is referred to as algorithm WBL+DO. Instead of iterating over all edges of the input graph in any order when inserting the edges into the data structure (as in the algorithm WBL), the algorithm WBL+DO computes a degeneracy ordering τ first and then iterates over the vertices in the order given by τ to insert the edges into the data structure.

Listing 2.4: Implementation of the algorithm WBL+DO as presented by Wienöbst, Bannach, and Liśkiewicz (2021a). The algorithm WBL+DO computes a degeneracy ordering to reduce the cost for the setup of the novel data structure used by the algorithm WBL.

```
Input: A partially directed graph G = (V, E, A)
Output: A consistent DAG extension of G if G admits such an extension, else a negative answer

result ← copy of G
temp ← empty initialization of the data structure
vertex_order ← degeneracy ordering of G
foreach vertex v in vertex_order do
  foreach neighbor u of v in G do
    if u precedes v in vertex_order then
      insert the edge between u and v into temp and update the data structure accordingly
    end
  end
end

L ← list of potential sinks in temp
while L is not empty do
  s ← pop any element from L
  orient all undirected edges incident to s towards s in result
  remove s and all incident edges from temp and update the data structure accordingly
  add all previous neighbors of s to L that became a potential sink in temp
end
if there are edges left in temp then
  return a negative answer
end
return result
```

A degeneracy ordering can be computed in time $O(n + m)$ using the algorithm by Matula and Beck (1983). The algorithm by Matula and Beck applies the following steps in order to compute a degeneracy ordering $\tau = v_1, v_2, \dots, v_n$ for a given input graph G.

1. Initialize $j \leftarrow n$ and let G' be a copy of G.
2. Set v_j as a vertex with minimum degree in G'.
3. Remove v_j from G' and set $j \leftarrow j - 1$.
4. If $j \geq 1$, go back to step 2, otherwise return $v_1, v_2, \dots, v_n$.

After computing the degeneracy ordering $\tau = v_1, v_2, \dots, v_n$, the algorithm WBL+DO iterates over the vertices in the order given by τ and inserts edges to preceding neighbors only. That is, whenever processing a vertex v_i, undirected edges $\{v_i, v_j\}$ and arcs (v_i, v_j) or (v_j, v_i) are only inserted into the data structure if $\tau(v_i) > \tau(v_j)$. Listing 2.4 shows the algorithm WBL+DO in its entirety.

Recall that in a degeneracy ordering $\tau = v_1, v_2, \dots, v_n$, every vertex v_i has at most d neighbors in $v_1, v_2, \dots, v_{i-1}$ where $d \leq \Delta \leq n$ is the degeneracy of the skeleton of the input graph. Therefore, we have to pay only $O(d)$ for inserting an edge from the input graph into the data structure because we insert edges to preceding neighbors only. Inserting all edges from the input graph into the data structure thus costs $O(dm)$.

In the following, we examine how the runtime for the loop can be bounded by $O(dm)$ as well, yielding a total runtime of $O(dm)$ for the algorithm WBL+DO. As before, the costly part is the update of the α- and β-values for the neighborhood of a removed vertex. Wienöbst, Bannach, and Liśkiewicz (2021a) showed that removing a potential sink s and all of its incident edges can be done in time $O(\delta(s)^2 + \delta(s) \cdot \delta^-(s))$ as follows.

We first iterate over all neighbors of s that are connected to s via an arc, i.e., all vertices v for which an arc (v, s) exists in the graph because s has no outgoing arcs. Any arc (v, s) may contribute to the α- or β-value of a vertex u if we have $u \sim v$ and $\{s, u\} \in E$ (having $(u, s) \in A$ does not contribute to the α- or β-value of u and we have $(s, u) \notin A$ because s has no outgoing arcs). It follows that we can update the α- and β-values correctly when removing a single arc (v, s) by iterating over all undirected neighbors u of s. Thus, removing a single arc (v, s) is possible in time $O(\delta(s))$ and the removal of all arcs incident to s can be done in $O(\delta(s) \cdot \delta^-(s))$.

After the removal of all arcs incident to s, we iterate over the undirected neighbors of s and remove all undirected edges incident to s in time $O(\delta(s)^2)$ as s has only undirected neighbors left. Overall, the runtime for the removal of a potential sink can then be bounded by $O(\delta(s)^2 + \delta(s) \cdot \delta^-(s))$. Utilizing the following observation, we can actually bound the runtime for the removal of all potential sinks by $O(dm)$.

Fact 2.4 (Wienöbst, Bannach, and Liśkiewicz, 2021a). A potential sink in a d-degenerate graph has at most $d + 1$ undirected neighbors.

Hence, removing a potential sink s costs $O(d^2 + d \cdot \delta^-(s))$ in a d-degenerate graph. Observe that each removal of a potential sink s deletes the $d + \delta^-(s)$ incident edges of s and thus the total runtime of the algorithm WBL+DO can be bounded by $O(dm)$.

2.3 Extensions for Subclasses of Partially Directed Acyclic Graphs

In this section, we investigate the problem EXTENDABILITY for specific classes of partially directed graphs. We begin with investigating directed input graphs as well as undirected input graphs and afterwards take a look at two important subclasses of PDAGs, namely so-called completed PDAGs (CPDAGs) and maximally oriented PDAGs (MPDAGs).

The linear-time algorithms for directed input graphs and undirected input graphs presented below are included in the experimental evaluation as a baseline but we emphasize that these algorithms are of little practical importance, since in practice we do not know in advance whether an input graph is fully directed or fully undirected. CPDAGs and MPDAGs are introduced as we derive new algorithms operating on CPDAGs and MPDAGs in Chapter 3.

Extensions for Directed Acyclic Graphs

Directed acyclic graphs (DAGs) are of particular interest to us because DAGs are widely used to model causal relationships. DAGs neither contain undirected edges nor cycles and DAGs admit a topological sorting $\tau = v_1, v_2, \ldots, v_n$ of their vertices such that for every arc $v_i \to v_j$ in the DAG we have $\tau(v_i) < \tau(v_j)$, i.e., v_i comes before v_j in τ.

Furthermore, DAGs can be partitioned into so-called Markov equivalence classes. As shown by Verma and Pearl (1990), two DAGs are Markov equivalent if and only if they have the same skeleton and the same set of v-structures. A Markov equivalence class contains all DAGs having the same skeleton and the same set of v-structures (Pearl, 2009). We write $G \approx G'$ if two DAGs G and G' are Markov equivalent. For a DAG G, we denote the Markov equivalence class of G as $[G] = \{G' \mid G \approx G'\}$.

In a causal DAG, the vertices represent random variables and the edges describe causal relationships between those random variables, i.e., a directed edge $X \to Y$ between two random variables X and Y illustrates that X is a direct cause of Y. Considering causal DAGs, Markov equivalence is an important concept as a Markov equivalence class contains all DAGs representing the same set of conditional independencies between the random variables represented by the vertices of the causal DAG (Pearl, 2009).

When solving the problem EXTENDABILITY for directed input graphs, we observe that directed graphs are extendable if and only if they are acyclic. Having the knowledge that an input graph contains only directed edges, checking whether the input graph is acyclic is sufficient in order to obtain a consistent DAG extension for it as every DAG is already a consistent DAG extension of itself. Cycle checking for a directed input graph can be accomplished in linear time, for example by deploying Kahn's algorithm (Kahn, 1962). Kahn's algorithm computes a topological sorting in time $O(n + m)$ if the input graph is acyclic or returns a negative answer if the input graph contains a cycle.

Kahn's algorithm keeps a list L of vertices with no ingoing edges and iterates over L until L becomes empty. In each iteration, some vertex v is removed from L and appended to the output sequence. The selected vertex v and all incident edges are then removed from the input graph. All neighbors of v having no incoming edges after the removal of v are then added to L. If L becomes empty while there are still edges left in the input graph, the input graph must be cyclic and a negative answer is returned. Otherwise, the output sequence forms a topological sorting of the vertices in the input graph and hence the input graph must be acyclic.

An algorithm to compute a consistent DAG extension for a directed input graph G first runs Kahn's algorithm to determine whether G is acyclic. In case G is acyclic, G is returned as a consistent DAG extension of itself, otherwise a negative answer is returned.

Extensions for Undirected Graphs

Undirected graphs are another extreme case for which consistent DAG extensions can be computed in linear time. Each undirected graph is extendable if and only if it is chordal and we have seen that an undirected graph is chordal if and only if it admits a PEO (Fulkerson and Gross, 1965). If we know in advance that an input graph contains only undirected edges, computing a PEO is sufficient to obtain a consistent DAG extension.

Listing 2.5: The Lexicographic Breadth-First Search (LBFS) algorithm to compute a vertex ordering for an undirected graph (Rose, Tarjan, and Lueker, 1976).

Input: *An undirected graph* $G = (V, E)$
Output: *A vertex ordering* τ *of the vertices of* G

$\Sigma \leftarrow (V)$ // initialize a sequence of sets containing a single set with all vertices
$\tau \leftarrow ()$ // initialize an empty output sequence
while $\Sigma = (S_1, S_2, \dots)$ *is not empty* ***do***
  $v \leftarrow$ *pop any vertex from* S_1
  if $S_1 = \emptyset$ ***then***
    remove S_1 *from* Σ
  end
  append v *to* τ
  $N(v) \leftarrow$ *all neighbors of* v *in* G
  $\Sigma \leftarrow (S_1 \cap N(v), S_1 \setminus N(v), S_2 \cap N(v), S_2 \setminus N(v), \dots)$ // only keep non−empty sets
end
return τ

Computing a PEO for a given undirected graph can be done in time $O(n + m)$ by applying algorithms such as Maximum Cardinality Search (MCS) (Tarjan and Yannakakis, 1984) or Lexicographic Breadth-First Search (LBFS) (Rose, Tarjan, and Lueker, 1976).

For an undirected input graph G, the LBFS algorithm keeps an ordered sequence of sets $\Sigma = (S_1, S_2, \dots)$ forming a partition of all unvisited vertices of G. At the beginning, Σ consists of a single set containing all vertices of G. In each iteration, a vertex v is removed from the first set in Σ and appended to the output sequence. Every set S in Σ is then replaced by two sets S' and S'' where S' contains all vertices in S that are adjacent to v and S'' contains the remaining vertices in S. During the replacement of S, S' is being placed before S''. Whenever a set in Σ becomes empty, it is removed from Σ. Listing 2.5 shows the pseudocode of the LBFS algorithm to compute a vertex ordering for an undirected graph G. The MCS algorithm operates in a similar way as the LBFS algorithm.

The vertex ordering computed by the LBFS algorithm is a PEO for G if G is chordal. An algorithm computing a consistent DAG extension for any undirected graph G now proceeds as follows. In the first step, LBFS (or MCS) is deployed to obtain a vertex ordering $\tau = v_1, v_2, \dots, v_n$ in time $O(n + m)$. Afterwards, we check whether τ is a PEO for G in linear time (Rose, Tarjan, and Lueker, 1976). If the check fails, i.e., if τ is not a PEO for G, a negative answer is returned as G does not admit a PEO and hence G cannot be extendable (because every extendable undirected graph has to be chordal and thus admits a PEO). In case τ is indeed a PEO for G, a consistent DAG extension of G can be computed by iterating over the vertices of G in the order given by the reverse of τ (i.e., $v_n, v_{n-1}, \dots, v_1$) and orienting all edges incident to the current v_i towards v_i in G for $i \in \{n, n-1, \dots, 1\}$.

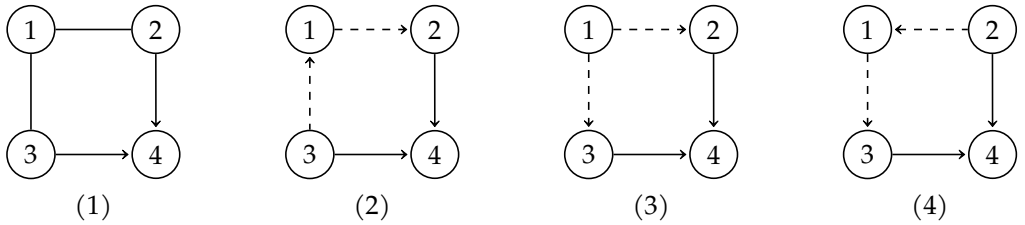


Figure 2.6: Graph (1) shows a CPDAG representing a Markov equivalence class which consists of the three DAGs (2), (3), and (4). Observe that other orientations for the edges $1 - 2$ and $1 - 3$ either cause a cycle or create a new v-structure.

Extensions for Completed Partially Directed Acyclic Graphs

Completed partially directed acyclic graphs (CPDAGs) represent a whole Markov equivalence class $\mathcal{C}$ (Andersson, Madigan, and Perlman, 1997). A CPDAG representing a Markov equivalence class $\mathcal{C}$ contains an undirected edge between two vertices u and v if there are two DAGs $G_1, G_2 \in \mathcal{C}$ with an edge $u \to v$ in G_1 and $u \leftarrow v$ in G_2. The CPDAG contains a directed edge $u \to v$ if there is an edge $u \to v$ in *every* DAG $G \in \mathcal{C}$. For example, graph (1) in Figure 2.6 is a CPDAG representing a Markov equivalence class $\mathcal{C}$ consisting of the DAGs (2), (3), and (4).

CPDAGs are especially relevant in the context of learning causal graphs from observed data. For example, the popular PC algorithm (Spirtes, Glymour, and Scheines, 1993) learns a CPDAG that represents a Markov equivalence class containing all DAGs that possibly explain the observed data used as input for the algorithm.

Each connected component of the undirected graph resulting from removing all directed edges in a CPDAG is an undirected and connected chordal graph (UCCG) (Andersson, Madigan, and Perlman, 1997). We call an orientation of an UCCG U obtained by replacing each edge $u - v$ in U by either $u \to v$ or $v \to u$ an acyclic moral orientation (AMO) of U if the resulting graph is acyclic and contains no v-structure. Note that an AMO can be seen as a consistent DAG extension as the only difference between an AMO and a consistent DAG extension is that an AMO never contains any v-structures.

CPDAGs are extendable by definition due to the fact that every UCCG of a CPDAG is chordal (Andersson, Madigan, and Perlman, 1997). Every DAG in the Markov equivalence class represented by a CPDAG G is a consistent DAG extension of G.

Hauser and Bühlmann (2012) described an algorithm to extend a CPDAG in linear time. A consistent DAG extension for a CPDAG G can be computed by orienting every UCCG in G. An AMO for an UCCG can be computed in linear time with the help of algorithms such as the LBFS algorithm (see Listing 2.5) or the MCS algorithm.

As each UCCG can be oriented in time $O(n + m)$ with n being the number of vertices and m the number of edges in the UCCG, the total runtime for computing a consistent DAG extension of a CPDAG is $O(n + m)$ as well.

Extensions for Maximally Oriented Partially Directed Acyclic Graphs

Maximally oriented partially directed acyclic graphs (MPDAGs) are closed under the four rules of Meek (1995). Meek's rules are illustrated in Figure 2.7 and give necessary ori-

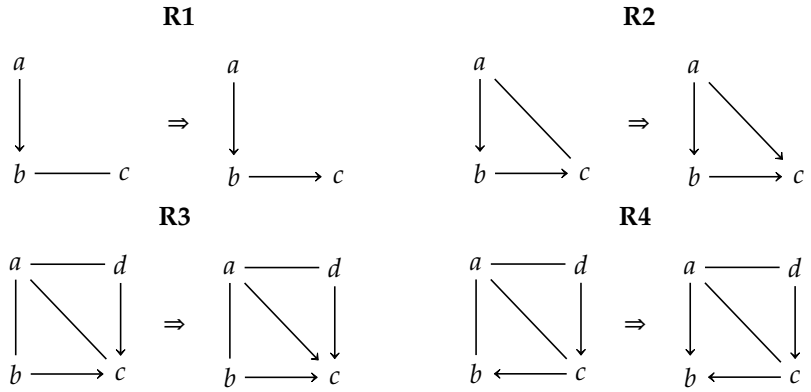


Figure 2.7: The four orientation rules as stated by Meek (1995). Meek's rules give the only possible orientation without causing cycles or new v-structures when orienting all undirected edges of the graphs shown above.

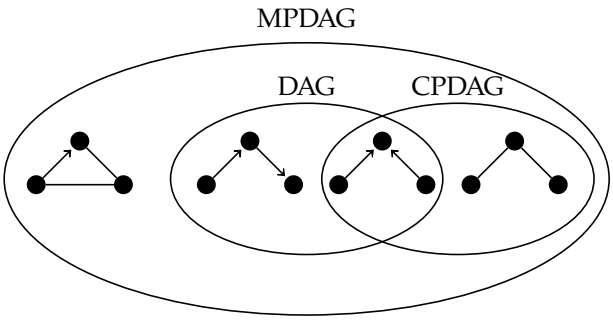


Figure 2.8: The relation between MPDAGs, CPDAGs and DAGs, which are all subclasses of PDAGs. MPDAGs are the most general subclass in this hierarchy as MPDAGs include both DAGs and CPDAGs. Observe that, however, not every MPDAG has to be a DAG or a CPDAG (the leftmost graph is an MPDAG but neither a DAG nor a CPDAG). This illustration is a modification of Figure 1 from (Wienöbst, Bannach, and Liśkiewicz, 2021a).

entations in order to avoid the emergence of cycles and new v-structures when orienting the undirected edges of a PDAG.

The class of MPDAGs includes DAGs, CPDAGs, as well as CPDAGs with additional background knowledge. More specifically, MPDAGs are exactly the graphs that are learnable from observed data and additional background knowledge. CPDAGs thus encode less causal information than MPDAGs in general and MPDAGs generally encode less causal information than DAGs.

In order to gain a better understanding of the relations between MPDAGs, CPDAGs, and DAGs, consider Figure 2.8. Note that every CPDAG is also an MPDAG and every DAG is an MPDAG as well. However, not every MPDAG has to be a CPDAG or a DAG, as the leftmost graph in Figure 2.8 demonstrates.

For an MPDAG G, we call the subgraph induced by a maximal connected undirected component of G a bucket (Perkovic, 2020). A bucket might contain directed edges but no v-structures (Wienöbst, Bannach, and Liśkiewicz, 2021a). Analogous to UCCGs, we call

Graph Class	Time Complexity
PDAG	$O(n^3)$
CPDAG	$O(n + m)$
MPDAG	$O(n + m)$

Table 2.9: The upper bounds on the time complexity of the problem EXTENDABILITY for PDAGs, CPDAGs, and MPDAGs.

an orientation of a bucket B obtained by replacing each edge $u - v$ in B by either $u \to v$ or $v \to u$ an AMO if the resulting graph is acyclic and contains no v-structure.

MPDAGs are not necessarily extendable by definition as the buckets in an MPDAG may contain directed edges and the skeletons of the buckets are not necessarily chordal. An extendable MPDAG must therefore contain only buckets with a chordal skeleton.

Fact 2.5 (Wienöbst, Bannach, and Liśkiewicz, 2021a). An MPDAG is extendable if and only if the skeleton of every bucket is chordal.

In order to orient the buckets of an MPDAG, we cannot apply the LBFS algorithm (and neither the MCS algorithm) because the directed edges in the buckets have to be taken into account. However, Wienöbst, Bannach, and Liśkiewicz (2021a) proposed a modification of the LBFS algorithm that can be applied to extend MPDAGs in linear time.

To extend an MPDAG G in linear time, the LBFS algorithm might be modified to compute an orientation for a bucket of G that is consistent with the already existing directed edges in the bucket. In each step of the LBFS algorithm introduced earlier in Listing 2.5, a vertex is chosen from a set of possible vertices to explore next. The modification of the LBFS algorithm now additionally requires the selected vertex to have no incoming arcs from unvisited vertices in the bucket. As shown by Wienöbst, Bannach, and Liśkiewicz (2021a), such a vertex must always exist.

Subsequently, the vertex ordering τ computed by the modified LBFS algorithm gives an AMO for a bucket B if B is extendable. As buckets are not guaranteed to be extendable, a check whether τ gives indeed an AMO is necessary. Checking whether τ gives an AMO can be done in linear time by verifying that τ is a PEO (Rose, Tarjan, and Lueker, 1976). It follows that a consistent DAG extension for an MPDAG G can be obtained in linear time by computing an AMO for each bucket of G using the modified LBFS algorithm introduced above. If at least one bucket of G is not extendable, a negative answer is returned. The total runtime for the computation of a consistent DAG extension for an MPDAG is then bounded by $O(n + m)$.

Table 2.9 summarizes the time complexities of the problem EXTENDABILITY for PDAGs, CPDAGs, and MPDAGs.

2.4 Enumeration of Extensions for Partially Directed Acyclic Graphs

In this section, we define the problem of enumerating all consistent DAG extensions for a given input graph G. Sometimes, it is not only of interest to find a single consistent DAG

extension for G but instead to compute all consistent DAG extensions of G.

Computing all consistent DAG extensions of a CPDAG G is necessary when all DAGs in a Markov equivalence class need to be considered. For example, the algorithm proposed by Maathuis, Kalisch, and Bühlmann (2009) for the estimation of so-called average causal effects from observed data for a given CPDAG G considers all DAGs in the Markov equivalence class represented by G.

The set of all consistent DAG extensions for a partially directed graph G is denoted as $CE(G)$ and the number of consistent DAG extensions of G is written as $\#CE(G) = |CE(G)|$. We refer to the problem of computing all consistent DAG extensions for an input graph as the enumeration problem and define the problem ENUMERATION for a partially directed input graph below. Clearly, ENUMERATION can be defined for more specific types of input graphs such as MPDAGs or CPDAGs as well.

Problem 2.6 (ENUMERATION).
Input: A partially directed graph G.
Output: $CE(G)$.

Now that we have defined the problems EXTENDABILITY and ENUMERATION, we next investigate how to solve them efficiently in practice.

3

Approaches

In this chapter, we derive practical implementations based on the algorithms from the last chapter to efficiently solve the problems EXTENDABILITY and ENUMERATION in practice. At first, different representations for input graphs that are used in the implementations are discussed. We continue with an investigation of the generated dataset of input graph instances, followed by an overview of the actual implementation details for the algorithms solving the problem EXTENDABILITY efficiently. At the end, we close this chapter with an in-depth analysis of algorithms solving the problem ENUMERATION.

All algorithms have been implemented using the Julia programming language[2] which is well suited for scientific computing. The source code used in the context of this thesis is publicly available on GitHub[3].

3.1 Graph Representations

In this section, the encoding of input graph instances is being examined. All input graphs are read from a text file and then passed into a LightGraphs[4] object. LightGraphs is a popular and simple library for dealing with graphs in the Julia programming language. A special feature of LightGraphs is that the neighbors for each vertex are stored in a sorted list and hence the edges of the graph are not accessed in a random order.

The LightGraphs representation of the input graph is then passed to the algorithms that compute consistent DAG extensions, i.e., all algorithms receive the exact same graph representation as input. Most of the algorithms for computing consistent DAG extensions require a copy of the input graph since vertices and edges have to be removed from the input graph. An exact copy of the LightGraphs representation, i.e., a copy that uses a LightGraphs representation as well, showed a poor performance during the experimental evaluation and is thus not investigated further.

Instead, each algorithm uses its own custom data structure to represent a copy of the input graph which performs better than the LightGraphs representation. In order to keep the comparison between different algorithms for computing consistent DAG extensions

[2] https://julialang.org/ (accessed on 2021-09-05)
[3] https://github.com/Malte311/PdagExtendability (accessed on 2021-09-05)
[4] https://github.com/JuliaGraphs/LightGraphs.jl (accessed on 2021-09-05)

as fair as possible, each algorithm must output its result represented as a LightGraphs object again. When measuring the runtime of an algorithm, we only measure the time needed by the algorithm itself, i.e., we do not measure the time needed for reading the input graph from the text file.

The algorithms WBL and WBL+DO use the data structure presented in Section 2.2 to represent the copy of the input graph. Recall that a partially directed graph G is represented using a hybrid graph representation that utilizes two directed graphs G_1 and G_2 where G_1 stores the undirected edges of G in form of two arcs and G_2 stores the directed edges of G. In theory, the data structure of the algorithms WBL and WBL+DO presented in the original paper by Wienöbst, Bannach, and Liśkiewicz (2021a) uses an adjacency matrix that stores pointers. However, such an adjacency matrix is practically not implementable using the Julia programming language and therefore we use arrays containing hash sets for the implementation as hash sets are generally well-suited in practice. Each directed graph G_i stores two arrays of size n consisting of a hash set for each vertex v containing all ingoing and outgoing neighbors of v, respectively. Those two arrays allow for adjacency tests in expected time $O(1)$.

The other algorithms (i.e., all algorithms except for WBL and WBL+DO) use a graph representation that consists of three arrays containing the ingoing, outgoing, and undirected edges for all vertices, respectively. The array storing the ingoing edges of the graph contains at index v a hash set of all vertices u for which an arc $u \to v$ exists in the graph. Analogously, the array storing the outgoing edges contains at index v a hash set of all vertices u with $v \to u$ and the array storing the undirected edges contains at index v a hash set of all vertices u for which an edge $u - v$ exists in the represented graph.

Before we derive practical implementations of the previously discussed algorithms for efficiently computing consistent DAG extensions in practice, we first examine how a suitable dataset of input graph instances is obtained.

3.2 Dataset Generation

This section explains in detail how a suitable dataset of input graph instances for the experimental evaluation has been generated. Clearly, the generated dataset of input graph instances should cover a wide range of input graphs. On the one hand, extreme cases such as fully directed and fully undirected input graphs should be included. On the other hand, miscellaneous input graphs containing both undirected and directed edges should be considered. As most of the input graphs in the real world are relatively sparse, our main focus is on sparse input graphs.

In total, the generated dataset contains 1 127 different input graph instances. These input graph instances can be separated into 139 fully directed graphs, 193 fully undirected graphs, and 795 partially directed graphs. Among the partially directed graphs, there are graphs with different proportions of directed edges, ranging from roughly 10% of their edges being directed to about 90% of their edges being directed. The generated input graph instances include worst-case instances for the evaluated algorithms as well as randomly generated input graphs. To obtain a great variety in the dataset, different graph generation approaches have been deployed to generate the random graphs.

Listing 3.1: The algorithm to generate a random graph with n vertices and m edges using the Erdős–Rényi model.

Input: *An integer $n \geq 0$ and an integer m with $0 \leq m \leq \frac{n \cdot (n-1)}{2}$*
Output: *A randomly generated graph with n vertices and m edges*

$G \leftarrow$ empty graph with n vertices
while *G contains less than m edges* ***do***
  $u \leftarrow$ uniformly chosen random vertex in G
  $v \leftarrow$ uniformly chosen random vertex in G
  if *u is different from v* ***and*** *the edge $\{u, v\}$ is not already in G* ***then***
    insert *the edge $\{u, v\}$ into G*
  end
end
return *G*

We focus on well-known random graph generation approaches that aim at modeling real-world networks. The applied random graph generation approaches are now described in more detail.

Random Graph Generation Approaches

For simplicity, the random graph generation approaches described below insert undirected edges only. However, all of the approaches can also be modified to insert directed edges instead (or a mixture of undirected edges and directed edges).

A well-known and widely used model to generate random graphs is the Erdős–Rényi model (Erdős and Rényi, 1959; Erdős and Rényi, 1960). In the Erdős–Rényi model, all graphs over the same set of vertices and the same number of edges are equally likely. That is, edges are inserted randomly between two vertices u and v if u and v are not already connected by an edge. The generation procedure to create a random graph with n vertices and m edges using the Erdős–Rényi model is given in Listing 3.1.

Another model that has been applied to construct randomly generated graphs is the Watts–Strogatz model (Watts and Strogatz, 1998). Watts and Strogatz argue that in many real-world networks, edges are not distributed completely random like in the Erdős–Rényi model but instead tend to include local clustering. The Watts–Strogatz model takes three parameters n, k, and β as input and incorporates local clustering.

In order to create a random graph with n vertices, the Watts–Strogatz model starts with an initial ring lattice, i.e., a graph consisting of n vertices where each vertex is connected to the k (or $k - 1$ if k is odd) nearest other vertices. Graph (1) of Figure 3.2 is an example for such a ring lattice with $n = 10$ and $k = 4$. After creating the initial ring lattice, edges are rewired with probability β as described in Listing 3.3. For example, graph (2) of Figure 3.2 has been created with $\beta = 0.1$ and graph (3) has been created with $\beta = 0.9$. Both graph (2) and graph (3) use graph (1) as an initial ring lattice before rewiring edges.

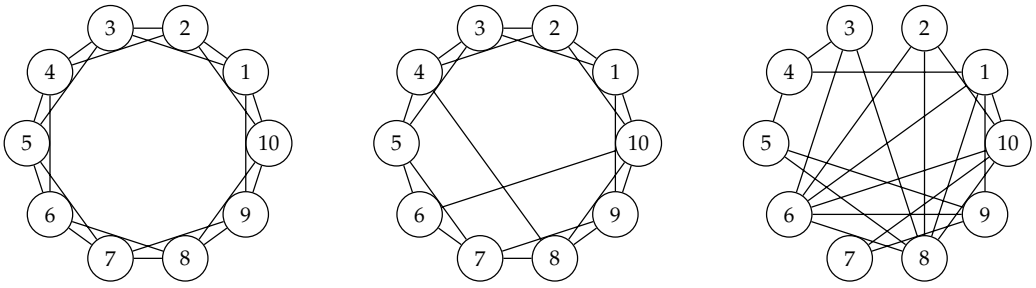


Figure 3.2: Graph (1) shows an example for a ring lattice with $n = 10$ and $k = 4$. Both graph (2) and graph(3) have been generated utilizing the Watts–Strogatz model with graph (1) as an initial ring lattice. Graph (2) has been created with the parameter $\beta = 0.1$ and graph (3) uses $\beta = 0.9$.

Observe that the rewiring of edges results in local clustering. For example, the vertices 6 and 8 in graph (3) of Figure 3.2 each form a local cluster.

As observed by Ravasz et al. (2002), the degree distribution in many real-world networks follows a so-called power law, i.e., there are few vertices with a high degree while most vertices have a small degree. However, in the Erdős–Rényi model and the Watts–Strogatz model, the degree distribution does not follow a power law.

Therefore, we consider the Barabási–Albert model (Albert and Barabási, 2002) as a third model to generate random graphs. Graphs generated with the Barabási–Albert model indeed have degree distributions following a power law. The random graph generation approach using the Barabási–Albert model takes two parameters n and k as input.

In order to create a random graph with n vertices, the approach using the Barabási–Albert model begins with an initial unconnected graph of k vertices and then attaches new vertices one by one until the graph contains n vertices. Each new vertex is attached via k edges to k different vertices already present in the graph by preferential attachment. Preferential attachment means that vertices with many neighbors are more likely to be chosen for the attachment than vertices with less neighbors. Listing 3.4 describes the whole algorithm to create a random graph with the help of the Barabási–Albert model.

The three random graph models introduced above, however, do not guarantee to generate graphs which are actually extendable. In order to assure that there are sufficiently many extendable graphs in the dataset, we apply the techniques described in the following subsection to generate graphs that are guaranteed to be extendable.

Generation of Extendable Input Graph Instances

Recall that directed graphs are extendable if they do not contain any directed cycle. Hence, in order to obtain fully directed graphs that are extendable, it is sufficient to generate random DAGs. We deploy two different approaches for constructing random DAGs.

The first approach to generate a random DAG with n vertices and m arcs starts with an empty graph containing n vertices and proceeds similar to the Erdős–Rényi random graph model. That is, the random DAG is generated by repeating the following two steps until there are m arcs in the graph.

Listing 3.3: The procedure to generate a random graph with n vertices and mean degree k using the Watts–Strogatz model. The parameter k is assumed to be an even number (otherwise, the mean degree is $k - 1$). The model randomizes edges with probability β after creating an initial ring lattice as a starting point.

Input: *An integer $n \geq 0$, an integer k with $0 \leq k < n$, and a real number $0 \leq \beta \leq 1$*
Output: *A randomly generated graph with n vertices and $n \cdot \lfloor k/2 \rfloor$ edges*

if k is odd then
  $k \leftarrow k-1$
end
$G \leftarrow$ *regular ring lattice with n vertices where each vertex has $k/2$ neighbors on each side*
foreach vertex u in G do
  foreach neighbor v of u in G do
    rewire the edge $\{u, v\}$ with probability β:
      1. ***delete*** *the edge $\{u, v\}$ from G*
      2. ***insert*** *the edge $\{u, w\}$ for some uniformly random chosen vertex w that is not already connected to u into G*
  end
end
return *G*

---

1. Insert an uniformly chosen random arc into the graph.
2. Check whether the graph became cyclic. If so, remove the arc again.

The second approach to generate a random DAG starts with a graph consisting of a single vertex and inserts new vertices in each iteration until there are n vertices in the graph. Every time a new vertex v is inserted, an arc between each previously existing vertex u and v is inserted into the graph with a specified probability p. Since arcs are only inserted from already existing vertices to new vertices, no cycle can emerge.

Considering fully undirected graphs, we know that undirected graphs are extendable if they are chordal. Again, two different approaches for generating chordal graphs have been put to use. One approach is to compute the intersection of subtrees of a tree (Şeker et al., 2017). The first step in this approach is to generate a tree T with n vertices uniformly at random. A tree is a graph where any two vertices are connected by exactly one path. The random chordal graph can then be constructed by computing the intersection of n random subtrees of T where a second parameter k gives the average size of those subtrees. For more details about the generation of the random trees, the reader is referred to (Şeker et al., 2017). It has been proven by Şeker et al. (2017) that the approach described above is capable of generating every possible chordal graph.

The second approach being used to generate a random chordal graph is the computation of a random PEO (Andreou et al., 2005). Computing a random PEO is accomplished by inserting a new vertex in each iteration until there are n vertices in the graph. Whenever a new vertex v is inserted into the graph, v is connected via an edge to a random

Listing 3.4: The algorithm to create a random graph with n vertices using the Barabási–Albert model. The graph is constructed beginning with an initial graph of k unconnected vertices. Then, new vertices are attached via k edges to k vertices already present in the graph until there are n vertices contained in the graph.

Input: *An integer $n \geq 0$ and an integer k with $1 \leq k < n$*
Output: *A randomly generated graph with n vertices and $(n-k) \cdot k$ edges*

$G \leftarrow$ empty graph with k vertices
insert *a new vertex s into G and connect s to every existing vertex in G via an edge*
for *$u \leftarrow k{+}2$* ***to*** *n* ***do***
  select *k existing vertices from G; each vertex v is chosen with probability $\Pi(v) = \frac{\Delta(v)}{\sum_{w \in V} \Delta(w)}$*
  insert *the new vertex u into G*
  insert *edges between u and each of the chosen vertices into G*
end
return *G*

---

selection of already existing vertices. The number of randomly selected vertices depends on the parameter k and is chosen uniformly at random from the interval between $k/2$ and $2k$. Furthermore, additional edges are inserted between the randomly selected vertices such that they form a clique.

Now that we have seen how to generate extendable directed graphs as well as extendable undirected graphs, we next examine how to generate extendable partially directed graphs. In order to generate a partially directed graph that is guaranteed to be extendable, the following procedure is applied.

The starting point is an extendable undirected graph G for which a consistent DAG extension G' is computed. Afterwards, each edge of G is oriented with probability p, where p can be chosen arbitrarily. Every time an edge is being oriented, the orientation of the newly formed arc is given by the orientation of the same arc in G'. Orienting the edges of G in the same direction as in G' guarantees that the resulting partially directed graph is actually extendable. This approach, however, generates only PDAGs with a chordal skeleton (i.e., PDAGs without v-structures) as generating extendable PDAGs with different numbers of v-structures at random is a difficult task.

Observe that starting with a directed graph and then replacing arcs randomly with undirected edges does not guarantee to deliver an extendable graph and thus we apply only the approach described above to generate extendable partially directed graphs.

Before we continue with the implementation details on how to efficiently compute consistent DAG extensions in practice, it should be mentioned that there are also some purposefully selected worst-case input graph instances in the dataset. By worst-case input graph instances we mean input graphs for which an algorithm performs exceptionally bad compared to the other algorithms.

The two most important of these worst-case input graph instances are the path graph and the complete graph. A path graph $G = (V, E, A)$ is a graph consisting of n vertices

and $m = n - 1$ edges where we have either $\{i, i+1\} \in E$ or $(i, i+1) \in A$ for each $i \in \{1, 2, \dots, n-1\}$. Complete graphs are graphs where each vertex is connected to every other vertex in the graph via an edge.

3.3 Efficient Computation of Extensions for Partially Directed Acyclic Graphs

This section describes the approaches being used to efficiently compute consistent DAG extensions for arbitrary PDAGs in practice. We first propose two modifications of the algorithm DT given by Dor and Tarsi (1992) which are both more efficient in practice than the algorithm DT itself. Afterwards, we investigate a promising alternative approach for computing consistent DAG extensions that converts the input graph into an MPDAG and then computes a consistent DAG extension for the obtained MPDAG.

Approaches for the Efficient Computation of Extensions

Considering the algorithmic idea presented by Dor and Tarsi (1992), three different variants of this algorithmic idea have been implemented. The first variant is a realization of the pseudocode given in the original paper by Dor and Tarsi. We have called this variant algorithm DT and already investigated the algorithm DT in Listing 2.2.

When searching for a potential sink, the algorithm DT repeatedly iterates over all vertices. We highlight that the algorithm DT has the potential for early cancellation when checking whether a vertex v is a potential sink. That is, the algorithm DT first checks whether v has an outgoing arc in time $O(1)$ and if so, the algorithm DT continues with the next vertex and saves the work of iterating over the neighborhood of v. Furthermore, after verifying that a vertex v has no outgoing arc, the algorithm DT may also stop early while iterating over the neighborhood of v if an undirected neighbor of v is found that is not adjacent to another neighbor of v. Note that the algorithm DT also stops the iteration over the vertices of the input graph as soon as a potential sink has been found.

The second variant that has been implemented is a modification of the algorithm DT which we call algorithm DT+P. The algorithm DT+P begins with a pre-computation of a list of potential sinks L. Computing a list of potential sinks L can be done by iterating over all vertices in the input graph once and adding each vertex that is a potential sink to L. Afterwards, the algorithm DT+P removes a potential sink s from L in each iteration and proceeds similar to the algorithm DT, i.e., all undirected edges incident to s are oriented towards s in the output graph and s is removed from the input graph. Every time a potential sink s is removed from the input graph, all previous neighbors of s that became a potential sink after the removal of s are added to L (Chickering, 2002). If the list of potential sinks L becomes empty while there are still vertices left in the input graph, a negative answer is returned. Listing 3.5 describes the steps undertaken by the algorithm DT+P to compute a consistent DAG extension for a given input graph.

Observe that the pre-computation runs in time $O(n^3)$ because we iterate over all vertices in the graph and check whether each vertex is a potential sink. Checking whether a vertex v is a potential sink needs time $O(n^2)$ because we must verify that all undirected

Listing 3.5: Implementation of the algorithm DT+P that computes a list of potential sinks in advance which is then kept up-to-date after each vertex removal.

```
Input: A partially directed graph G = (V, E, A)
Output: A consistent DAG extension of G if G admits such an extension, else a negative answer

result ← copy of G
temp ← copy of G
L ← list of potential sinks in G
while there are vertices left in temp do
  if L is empty then
    return a negative answer
  end
  s ← pop any element from L
  orient all undirected edges incident to s towards s in result
  remove s and all incident edges from temp
  add all previous neighbors of s in temp that became a potential sink to L
end
return result
```

neighbors of v are adjacent to all other neighbors of v. After the pre-computation, there is a loop with $O(n)$ iterations in total. In each iteration, we have to check for all neighbors of the removed vertex whether they became a potential sink. Consequently, a single iteration costs time $O(n^3)$ and thus the runtime for the whole loop can be bounded by $O(n^4)$. The worst-case time complexity of the entire algorithm DT+P hence is $O(n^4)$.

The third variant of the algorithmic idea by Dor and Tarsi is referred to as algorithm DT+H and uses a heuristic to determine which vertex to check next when searching for a potential sink. Instead of iterating over the vertices in an arbitrary order when searching for a potential sink, the heuristic chooses a vertex with the smallest degree first since vertices with a small degree cause less costs than vertices with a higher degree.

Computing the degrees of the vertices and keeping them up-to-date is done as follows. The algorithm DT+H stores the vertices in a list L of size n where each list entry $L[i]$ contains a hash set of vertices with degree i, i.e., $L[i] = \{v \mid \Delta(v) = i\}$. Every time a potential sink s is removed from the graph, all previous neighbors of s are moved to the correct set in L. That is, if a neighbor u of s is stored at $L[j]$ before the removal of s, then u is removed from $L[j]$ and added to $L[j-1]$ after the removal of s. Listing 3.6 depicts the entire algorithm DT+H to compute a consistent DAG extension for an input graph.

Note that the list L can be constructed in time $O(n)$ by iterating over the vertices of the input graph and looking up their degree in time $O(1)$. The degree $\Delta(v)$ of any vertex v can be determined in time $O(1)$ because the ingoing, outgoing, and undirected neighbors of v are stored in a hash set each and the sizes of the hash sets can be obtained in time $O(1)$ and summed up in order to get $\Delta(v)$. The worst-case time complexity of the algorithm DT+H hence can be bounded by $O(n^4)$.

Listing 3.6: Implementation of the algorithm DT+H that uses a heuristic to prefer vertices with low degrees as they produce less costs than vertices with a higher degree.

Input: *A partially directed graph* $G = (V, E, A)$
Output: *A consistent DAG extension of G if G admits such an extension, else a negative answer*

result ← *copy of G*
temp ← *copy of G*
L ← *list of hash sets with* $L[i] = \{v \in V \mid \Delta(v) = i\}$
while *there are vertices left in temp* ***do***
  foreach *set S* ***in*** *L* ***do***
    s ← *any vertex from S that satisfies the following properties*:
      1. *s has no arc directed outward from s*
      2. *all undirected neighbors of s are adjacent to all other neighbors of s*
    if *a vertex s satisfying the properties was found* ***then***
      break
    end
  end

  if *no vertex s satisfying the properties was found* ***then***
    return *a negative answer*
  end

  orient *all undirected edges incident to s towards s in result*
  remove *s and all incident edges from temp*
  update *the position in L for all previous neighbors of s in temp*
end
return *result*

Alternative Approaches for the Efficient Computation of Extensions

Before we turn our attention to the problem ENUMERATION, we briefly discuss an alternative approach to compute consistent DAG extensions for partially directed input graphs.

The previously introduced algorithms operate directly on the input graph. An alternative approach to compute a consistent DAG extension for a given input graph is to convert the input graph into an MPDAG first by applying Meek's rules and then extend the obtained MPDAG. Such a conversion is possible for extendable input graphs while a conversion might fail for non-extendable input graphs.

The reason we consider this approach is that the conversion from PDAG to MPDAG might be fast in practice if only few applications of Meek's rules are necessary for the conversion. As the obtained MPDAG can then be extended in linear time, the whole approach looks promising.

Any extendable PDAG G can be converted into an MPDAG G' by exhaustively applying Meek's rules on G. Note that each consistent DAG extension of G has the same

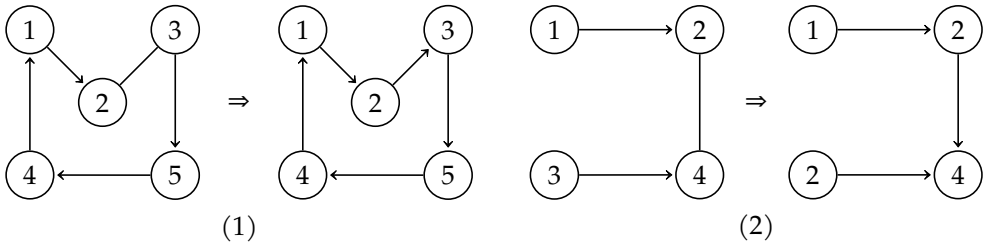


Figure 3.7: Applying Meek's rules on graph (1) causes a cycle while for graph (2) a new v-structure emerges from the application of Meek's rules. In graph (2), the orientation $4 \to 2$ might also arise from applying Meek's rules, yielding a new v-structure $1 \to 2 \leftarrow 4$.

orientations of the arcs contained in G' and thus a consistent DAG extension for G' is a consistent DAG extension for G as well. However, if G is not extendable, the application of Meek's rules may result in new v-structures or even cycles, as illustrated in Figure 3.7.

Considering graph (1) of Figure 3.7, the application of Meek's rules orients $1 \to 2 - 3$ as $1 \to 2 \to 3$ and thus creates a cycle. In graph (2), applying Meek's rules either orients $1 \to 2 - 4$ as $1 \to 2 \to 4$ or $3 \to 4 - 2$ as $3 \to 4 \to 2$ and both orientations result in a new v-structure. Consequently, an algorithm converting the input graph into an MPDAG has to verify that the application of Meek's rules neither creates new v-structures nor cycles.

An algorithm to compute a consistent DAG extension for a given partially directed graph G by converting G into an MPDAG G' and then extending G' proceeds as follows.

1. Apply Meek's rules exhaustively on G to obtain G'.
2. If G' contains a directed cycle or G and G' do not have the same set of v-structures, return a negative answer.
3. Run the algorithm from Section 2.3 to extend an MPDAG in linear time on G' and return its result.

Step 1 can be completed in time $O(\Delta m^2)$ (Wienöbst, Bannach, and Liśkiewicz, 2021a). Cycle checking needs linear time and the v-structures of a graph can be counted in $O(\Delta m)$. As the algorithm to compute a consistent DAG extension for an MPDAG runs in linear time, the worst-case time complexity of the whole algorithm is bounded by $O(\Delta m^2)$.

The experimental evaluation has shown that this algorithm performs significantly worse than the other algorithms for computing consistent DAG extensions and therefore this algorithm is not evaluated further. However, although the worst-case time complexity of $O(\Delta m^2)$ already suggests a worse performance than the other algorithms, it has not been clear before the experimental evaluation that the algorithm indeed yields a poor performance in practice. It would have been conceivable that worst-case scenarios (i.e., scenarios in which the conversion from PDAG to MPDAG is indeed costly) might occur rarely in practice.

3.4 Enumerating Extensions for Partially Directed Acyclic Graphs

The problem ENUMERATION is another important problem in the context of computing consistent DAG extensions. In this section, we are now interested in computing the set of *all*

consistent DAG extensions $CE(G)$ for a given input PDAG G.

Since the number of different orientations for a given input graph G is in $O(2^m)$, the number of consistent DAG extensions $\#CE(G)$ is, in general, in $O(2^m)$ as well. Consequently, algorithms solving the problem ENUMERATION inherently run in exponential time.

Therefore, our intention is to minimize the delay between the output of two consecutive consistent DAG extensions from $CE(G)$. More precisely, instead of computing $CE(G)$ and outputting the result in a single step at the end of the computation, our goal is to compute the members of $CE(G)$ step by step and output them as soon as they are available. The delay between the output of two members of $CE(G)$ should be bounded by a polynomial in the number of vertices and the number of edges in G.

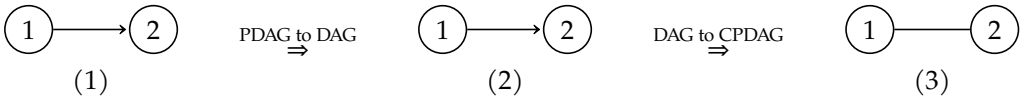
Such a polynomial bound on the delay has several advantages which are briefly highlighted below. Bounding the delay between the output of two consistent DAG extensions from $CE(G)$ allows the algorithm to compute $CE(G)$ in polynomial time for input graphs where the number of consistent DAG extensions $\#CE(G)$ is bounded by a polynomial. Furthermore, generating consistent DAG extensions step by step may reduce the memory needed to compute $CE(G)$. While an approach that fully computes $CE(G)$ and then outputs $CE(G)$ has exponential space requirements, an approach with a delay bounded by a polynomial may have only polynomial space requirements.

Generally, the input graph for the problem ENUMERATION can be any partially directed graph. Nevertheless, we are especially interested in CPDAG and MPDAG inputs because they are practically relevant and their properties simplify the computation of $CE(G)$. Algorithms solving the problem ENUMERATION for MPDAGs can also be used to solve the problem ENUMERATION for arbitrary partially directed graphs G by converting G into an MPDAG before applying the algorithm solving the problem ENUMERATION for MPDAGs.

Recall that every consistent DAG extension of a PDAG G has the same orientations of the arcs contained in the MPDAG resulting from applying Meek's rules on G. Hence, we can convert any extendable graph G into an MPDAG by exhaustively applying Meek's rules on G. As such a conversion might fail for non-extendable input graphs, we check whether a cycle has emerged or new v-structures have been created after applying Meek's rules. Remember that the conversion from PDAG to MPDAG, including the cycle check and the validation of the v-structures afterwards, can be done in time $O(\Delta m^2)$. Consequently, it is always advisable to convert an arbitrary partially directed graph G into an MPDAG before enumerating all consistent DAG extensions of G.

Although the problem ENUMERATION might be easier to solve for a CPDAG input than for an MPDAG input, we cannot convert an input graph G into a CPDAG using the approach given by Chickering (2002) because the orientation of some edges might be lost. The conversion from PDAG to CPDAG presented by Chickering proceeds in two steps. In the first step, a consistent DAG extension G' of the input graph G is computed. In the second step, the DAG G' is converted into the CPDAG representing $[G']$ by replacing all arcs in G' that do not participate in a v-structure by an undirected edge and afterwards applying Meek's rules exhaustively. An example for such a conversion from PDAG to CPDAG where the orientation of an edge is lost can be found in Figure 3.8.

Consider graph (1) of Figure 3.8 as the partially directed graph G for which we want to compute $CE(G)$. Graph (2) is a consistent DAG extension G' of G (in this example, G' is the same graph as G because G is already a DAG). The CPDAG representing $[G']$ is then



(1) (2) (3)

Figure 3.8: Graph (1) shows a partially directed graph G that has exactly one consistent DAG extension, namely G itself. When computing any consistent DAG extension G' of G, we obtain graph (2). Graph (3) is then the result of converting G' into the CPDAG representing $[G']$. Observe that in order to compute $CE(G)$, we cannot apply the transformations PDAG → DAG → CPDAG and then enumerate the consistent DAG extensions for the obtained CPDAG because graph (1) has only one consistent DAG extension while graph (3) has two consistent DAG extensions.

given by graph (3). However, that CPDAG has two consistent DAG extensions (i.e., the orientations $1 \to 2$ and $1 \leftarrow 2$) while G itself admits only one consistent DAG extension.

Meek (1995) already hinted at an algorithm to enumerate all consistent DAG extensions for an arbitrary partially directed graph. In the original paper, however, the algorithm was only outlined and not described in detail. The algorithm chooses any undirected edge in the input graph, tries both orientations for that edge and afterwards applies Meek's rules to propagate changes. Recall that for non-extendable input graphs, this step might create cycles or new v-structures and hence we have to check whether cycles or new v-structures have arisen (or alternatively, we first have to verify that the input graph is extendable by applying an algorithm solving the problem EXTENDABILITY). Subsequently, the algorithm continues with the next undirected edge and recurses until there are no undirected edges left in the graph.

Listing 3.9 describes the algorithm to compute $CE(G)$ for an input graph G in its entirety. The algorithm presented in Listing 3.9 is capable of dealing with arbitrary partially directed input graphs and converts the input graph into an MPDAG first as this might reduce the amount of necessary subsequent computation steps. Although the algorithm from Listing 3.9 works correctly for arbitrary partially directed input graphs, the delay between the output of two consecutive consistent DAG extensions can only be bounded by a polynomial for CPDAG inputs. For input graphs other than CPDAGs, the delay between two outputs might be exponential in the number of edges of the input graph.

We now show that the delay between the output of two consecutive consistent DAG extensions from $CE(G)$ for an input CPDAG G can indeed be bounded by a polynomial and afterwards give an analysis of the worst-case time complexity for the whole algorithm depicted in Listing 3.9.

Theorem 3.1. *The worst-case delay between the output of two consecutive consistent DAG extensions of the algorithm presented in Listing 3.9 can be bounded by $O(\Delta m^3)$ for CPDAG inputs.*

Proof. Let G be a CPDAG. The recursive calls of the algorithm can be visualized in form of a binary tree: The root of that binary tree represents the function call for the graph G itself, the left child represents the function call for the graph where any undirected edge $\{u, v\}$ of G is oriented as $u \to v$ and the right child represents the function call for the graph where the same edge is oriented as $v \to u$. Thus, each node of that binary tree represents an intermediate state computed by the algorithm and leaf nodes always

Listing 3.9: The algorithm outlined by Meek to enumerate all consistent DAG extensions for a partially directed graph (Meek, 1995). The algorithm orients randomly chosen edges, closes the result under the four Meek rules and recurses until there are no more undirected edges left.

Input: *A partially directed graph* $G = (V, E, A)$
Output: $CE(G)$

$\mu \leftarrow$ *number of v−structures in* G
call *enumerate*(G)

procedure *enumerate*(G)
  if G *contains only directed edges* ***then***
    if G *is acyclic* ***and*** *number of v−structures in* $G = \mu$ ***then***
      output G
    end
    exit
  end

  $G' \leftarrow$ *graph obtained by applying Meek's rules exhaustively on* G
  if G' *is acyclic* ***and*** *number of v−structures in* $G' = \mu$ ***then***
    $\{u, v\} \leftarrow$ *any undirected edge in* G'
    $G' \leftarrow$ *graph obtained by orienting* $\{u, v\}$ *as* $u \rightarrow v$
    $G'' \leftarrow$ *graph obtained by orienting* $\{u, v\}$ *as* $v \rightarrow u$
    call *enumerate*(G')
    call *enumerate*(G'')
  end
end

---

represent directed graphs. An exemplary visualization of such a binary tree representing the intermediate states computed by the algorithm can be found in Figure 3.10.

Both orientations for each undirected edge in G must occur in a consistent DAG extension of G since G is a CPDAG. Due to the application of Meek's rules after orienting an undirected edge, a new branch in the binary tree is only started for a valid orientation of the undirected edges. Consequently, every path from the root to any leaf node of the binary tree always ends up with a consistent DAG extension of G. In other words, each recursive call of the algorithm receives an extendable graph as input. The algorithm calls itself recursively until there are no more undirected edges left in G, i.e., a sequence of recursive calls follows some path from the root to a leaf node of the binary tree. Each path from the root to a leaf node in the binary tree has at most length m because every edge can be oriented at most once. Thus, there are at most m recursive calls before reaching a leaf node and outputting a consistent DAG extension.

It follows that the worst-case delay between two outputs is given by the time for

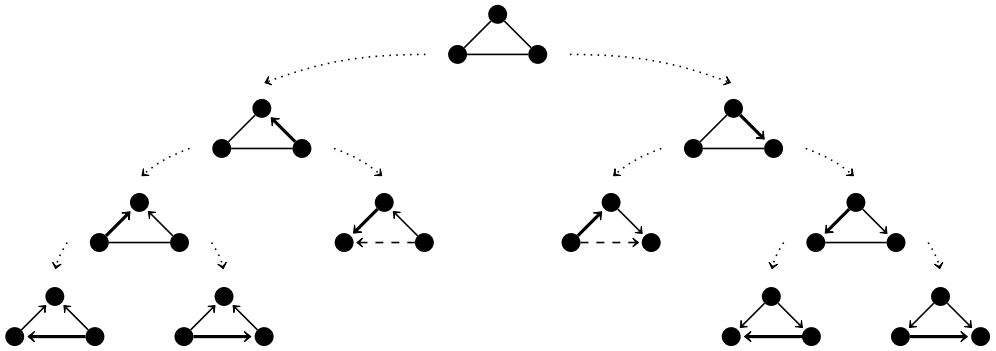


Figure 3.10: A visualization of the intermediate states computed by the algorithm shown in Listing 3.9 for a possible run on an exemplary input CPDAG. The uppermost graph illustrates the input CPDAG with three vertices and undirected edges between all of those vertices. In each step, the algorithm tries both orientations for an undirected edge and then applies Meek's rules to propagate changes. The first orientation of an edge is shown on the left and the second orientation on the right. The procedure is then repeated until no more edges can be oriented. Edges that have been oriented due to the application of Meek's rules are dashed.

traversing two consecutive paths from the root to a leaf node in the binary tree. Traversing a path of length m in the binary tree is possible in time $O(\Delta m^3)$ because in each node of that path, Meek's rules are applied, cycle checking is done, and v-structures are counted which costs time $O(\Delta m^2)$ in total. In conclusion, traversing two consecutive paths needs twice the time $O(\Delta m^3)$, so the delay between two outputs is in $O(\Delta m^3)$. □

Observe that the binary tree that represents the recursive function calls of the algorithm given in Listing 3.9 has at most 2^m leaf nodes because each leaf node represents a consistent DAG extension and there cannot be more than 2^m consistent DAG extensions for any graph with m edges. Thus, there are at most 2^m paths from the root to a leaf node of the binary tree. As we have seen above, traversing a path from the root to any leaf node of the binary tree costs $O(\Delta m^3)$ because every path has at most length m and each function call needs time $O(\Delta m^2)$ in total. The algorithm's total runtime hence is $O(2^m \Delta m^3)$.

Before we continue with the investigation on how to improve the delay when enumerating consistent DAG extensions for CPDAGs, we briefly examine how the algorithm presented in Listing 3.9 might be implemented in practice.

The algorithm shown in Listing 3.9 operates on an MPDAG because Meek's rules are applied before orienting an undirected edge. For an MPDAG, the orientation of one bucket has no effect on the orientations of other buckets and hence the buckets might also be oriented independently of each other. Orienting buckets separately reduces the number of undirected edges to be oriented and in addition to that, Meek's rules are applied on smaller graphs in each step. After orienting the buckets separately, the orientations of the buckets can be combined to obtain a consistent DAG extension for the whole MPDAG.

In the next section, we investigate strategies to improve the worst-case delay between the output of two consecutive DAG extensions for CPDAG inputs.

3.5 Enumerating Extensions for Completed Partially Directed Acyclic Graphs

This section introduces an algorithm solving ENUMERATION for CPDAGs where the delay between the output of two consecutive consistent DAG extensions can be bounded by $O(nm)$. Reducing the delay to $O(nm)$ is a considerable improvement compared to the delay of $O(\Delta m^3)$ yielded by the algorithm presented in the previous section. In order to obtain an algorithm with a delay of $O(nm)$, we adapt the algorithm presented by Talvitie and Koivisto (2019) for counting the number of AMOs in a given UCCG.

Every Markov equivalence class represented by a CPDAG G can be partitioned by fixing source vertices in each UCCG of G (He, Jia, and Yu, 2015). A source vertex s is a vertex where all incident edges are oriented outward from s and each AMO of an UCCG has exactly one source vertex (Hauser and Bühlmann, 2012). Combining the two observations that a Markov equivalence class can be partitioned by fixing source vertices and that each AMO of an UCCG has exactly one source vertex gives a recursive algorithm for enumerating all AMOs of an UCCG.

In order to compute all AMOs for an UCCG U, such a recursive algorithm iterates over the vertices in U and computes the s-orientation for each vertex s in U. For an UCCG U, the s-orientation for a vertex s in U is the graph resulting from applying the following steps on U (He, Jia, and Yu, 2015; Talvitie and Koivisto, 2019):

1. Orient each undirected edge $u - v$ in U as $u \to v$ if u has a smaller shortest path distance to s than v.
2. Orient all occurrences of $u \to v - w$ in U with u and w not being adjacent as $u \to v \to w$.

After computing an s-orientation for a vertex s, the algorithm is called recursively for each remaining undirected component until all edges are oriented. Observe that the computation of an s-orientation for a vertex s includes the application of the first Meek rule in step 2. It has been proven that the application of the first Meek rule is sufficient to prevent the emergence of new cycles or new v-structures in this case (He, Jia, and Yu, 2015).

Note that the AMOs for the UCCGs of a CPDAG G give exactly the members of the Markov equivalence class represented by G (He, Jia, and Yu, 2015). Thus, computing the s-orientation in G for every vertex s that is incident to at least one undirected edge and then recursing on the resulting graph for each s-orientation yields all consistent DAG extensions for G. The s-orientation for a vertex s in G means the graph that is obtained by orienting the UCCG U containing s according to the s-orientation for U in G.

Listing 3.12 contains a description of the entire algorithm to enumerate all consistent DAG extensions for a CPDAG and Figure 3.11 visualizes an exemplary run for an example input CPDAG with three vertices and undirected edges between all of those vertices.

In the following, we first show that the delay between two outputs of the algorithm presented in Listing 3.12 can be bounded by $O(nm)$ and afterwards we examine the worst-case time complexity of the whole algorithm.

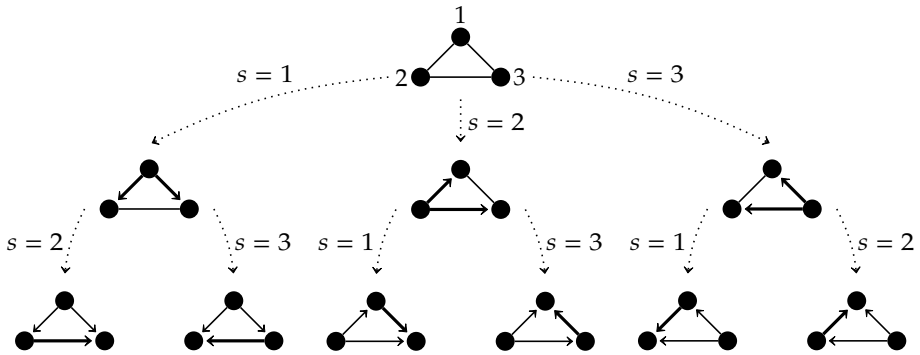


Figure 3.11: A visualization of the intermediate states computed by the algorithm depicted in Listing 3.12 for a possible run on an exemplary input CPDAG. The uppermost graph shows the input CPDAG with three vertices and undirected edges between all of those vertices. On each path from the uppermost graph to a graph at the bottom, the algorithm fixes a vertex s as a source, computes the s-orientation and calls itself recursively until all edges are oriented.

Theorem 3.2. *The worst-case delay between the output of two consecutive consistent DAG extensions of the algorithm presented in Listing 3.12 can be bounded by $O(nm)$.*

Proof. Let G be a CPDAG. The recursive calls of the algorithm can be visualized as a tree such that every node of that tree represents an intermediate state computed by the algorithm. The root of that tree represents the function call for G itself and each child of a node in the tree represents a function call for the graph where the s-orientation for some vertex s has been computed. Consequently, every leaf node of that tree is representing a directed graph. An example for such a tree is visualized in Figure 3.11.

Furthermore, each sequence of recursive function calls (i.e., each path from the root to any leaf node of the tree) actually ends with a consistent DAG extension of G and the leaf nodes represent the whole Markov equivalence class represented by G as a Markov equivalence class can be partitioned by fixing source vertices (He, Jia, and Yu, 2015).

Consequently, the worst-case delay between two outputs is given by the time for traversing two consecutive paths from the root to a leaf node in the tree. Each path from the root node to any leaf node in the tree has at most length n as every vertex can be chosen as a source at most once. In any function call represented by a node in the tree, the s-orientation (including the application of the first Meek rule) can be computed in time $O(n + m)$ by using the algorithm given by Wienöbst, Bannach, and Liśkiewicz (2021b). Traversing a path from the root node to any leaf node in the tree hence is possible in time $O(n \cdot (n + m)) = O(n^2 + nm)$.

However, the algorithm computes the s-orientations in UCCGs and every UCCG is always a connected graph by definition. For connected graphs, we have $m \geq n - 1$ and thus $O(n^2 + nm) \leq O(nm + nm)$. It follows that a path from the root node to any leaf node in the tree can be traversed in time $O(nm)$. As traversing two consecutive paths from the root to a leaf node in the tree can be done in twice the time $O(nm)$, the delay between two outputs is also bounded by $O(nm)$. □

Listing 3.12: The algorithm to efficiently enumerate all consistent DAG extensions for a CPDAG G. The algorithm computes the s-orientation for each vertex s in every UCCG U of G and recurses until all edges in G are oriented.

```
Input: A CPDAG G = (V, E, A)
Output: CE(G)

call enumerate(G)

procedure enumerate(G)
  if G contains only directed edges then
    output G
    exit
  end

  foreach vertex s in G do
    if s is incident to at least one undirected edge in G then
      G' ← graph obtained by computing the s−orientation in G
      call enumerate(G')
    end
  end
end
```

We close this chapter by analyzing the runtime of the algorithm from Listing 3.12. The delay of $O(nm)$ yields an algorithm with worst-case time complexity $O(2^m nm)$ as there are at most 2^m consistent DAG extensions for any input graph and every sequence of recursive function calls actually ends up with a consistent DAG extension of the input CPDAG. In other words, the tree representing the recursive function calls has at most 2^m leaf nodes because every leaf node represents a consistent DAG extension. As each path traversal from the root to a leaf node in the tree costs time $O(nm)$ and there are at most 2^m paths, the total runtime of the algorithm can be bounded by $O(2^m nm)$.

4

Evaluation

In the previous chapter, implementation details have been discussed. The next step is to conduct an experimental evaluation of the presented approaches for computing consistent DAG extensions. Hence, this chapter presents an overview of the experimental setup and afterwards showcases the results from the experimental evaluation. At the end of this chapter, the limitations of the experimental evaluation are briefly summarized.

4.1 Setup of the Evaluation

All experiments have been carried out on an Apple Mac mini (2018) with a 3.2 GHz 6-Core Intel Core i7 processor. The code has been executed in a Docker container with 8 GB RAM available. To obtain a better statistical significance, five measurements have been taken for each run. The final runtime is then given by the median of those five runs.

For each input graph instance, the results computed by all of the evaluated algorithms are compared to each other to verify that all algorithms actually agree on their answer whether the input graph is extendable and if so, whether all resulting consistent DAG extensions have the same set of v-structures as the input graph.

4.2 Evaluated Scenarios

This section gives a brief overview on the evaluated scenarios. Even though the generated dataset contains more than 1 000 input graph instances, we restrict the results presented in Section 4.3 to a small representative portion of the dataset. The input graphs shown in Section 4.3 contain about 10 000 vertices each except for the complete input graphs. The complete input graphs contain 256 vertices to fit the scale.

The presented graphs include both extendable input graphs as well as non-extendable input graphs and are either fully directed, fully undirected, or partially directed. Since sparse graphs are of particular interest from a practical point of view, we mainly consider randomly generated sparse graphs with various amounts of edges. The presented randomly generated sparse graphs yield similar results as other randomly generated sparse graphs with the same number of vertices and edges in the generated dataset, i.e., the

shown input graph instances are representative for the whole dataset.

Apart from the random graph instances, we regard worst-case input graphs as well. These worst-case input graphs consist of complete graphs for the algorithm WBL and path graphs for the algorithm DT. In Section 4.3, we explain in more detail why algorithm WBL performs badly on complete graphs and why algorithm DT is slow on path graphs.

Although a great variety of different types of graphs is exhibited, it should be emphasized again that in practice extendable sparse graphs are most relevant. Before we continue with the results of the experimental evaluation, a short overview of the algorithms being compared is given.

DT An implementation of the algorithm given in Listing 2.2 according to the algorithmic idea introduced by Dor and Tarsi (1992).

DT+H The newly proposed modification of the algorithm DT that uses a heuristic and is described in Listing 3.6.

WBL A realization of the algorithm WBL that has recently been proposed by Wienöbst, Bannach, and Liśkiewicz (2021a) and is shown in Listing 2.3.

WBL+DO The implementation of the algorithm WBL+DO given by Wienöbst, Bannach, and Liśkiewicz (2021a) that is presented in Listing 2.4.

MCS An algorithm to extend undirected graphs using the MCS algorithm (Tarjan and Yannakakis, 1984) as explained in Section 2.3.

TS The algorithm for extending directed graphs presented in Section 2.3 that uses Kahn's algorithm (Kahn, 1962) to compute a topological sorting.

Both algorithms MCS and TS are of little practical relevance and serve only as a baseline for the performance on undirected and directed graphs, respectively. Hence, we are mainly interested in the comparison of the algorithms DT, DT+H, WBL, and WBL+DO. We choose the MCS algorithm over the LBFS algorithm to extend undirected graphs as both algorithms are similarly fast and the MCS algorithm is easier to implement.

The algorithm DT+P (Listing 3.5) is no part of the plots in Section 4.3 to avoid overcrowded plots. Both algorithms DT+P and DT+H yield very similar results overall and as the algorithm DT+H performs slightly better than the algorithm DT+P, we focus on the algorithm DT+H. For directed graphs, the algorithm DT+P is superior to the algorithm DT+H but on all other types of graphs, the algorithm DT+H is faster. It is noteworthy that the algorithm DT+P is comparatively slow on very dense graphs, i.e., on graphs with many edges such as complete graphs. However, the algorithm DT+P still outperforms the algorithm DT in most of the evaluated scenarios.

4.3 Results of the Evaluation

This section covers the results of the experimental evaluation. During the experimental evaluation, three types of input graph instances have been deployed: fully directed graphs, fully undirected graphs, and partially directed graphs with different proportions of directed edges. For each of these graph types, extendable input graphs as well as non-extendable input graphs have been considered. We start with the extreme cases of fully directed and fully undirected input graphs and continue with the more general partially directed input graphs afterwards.

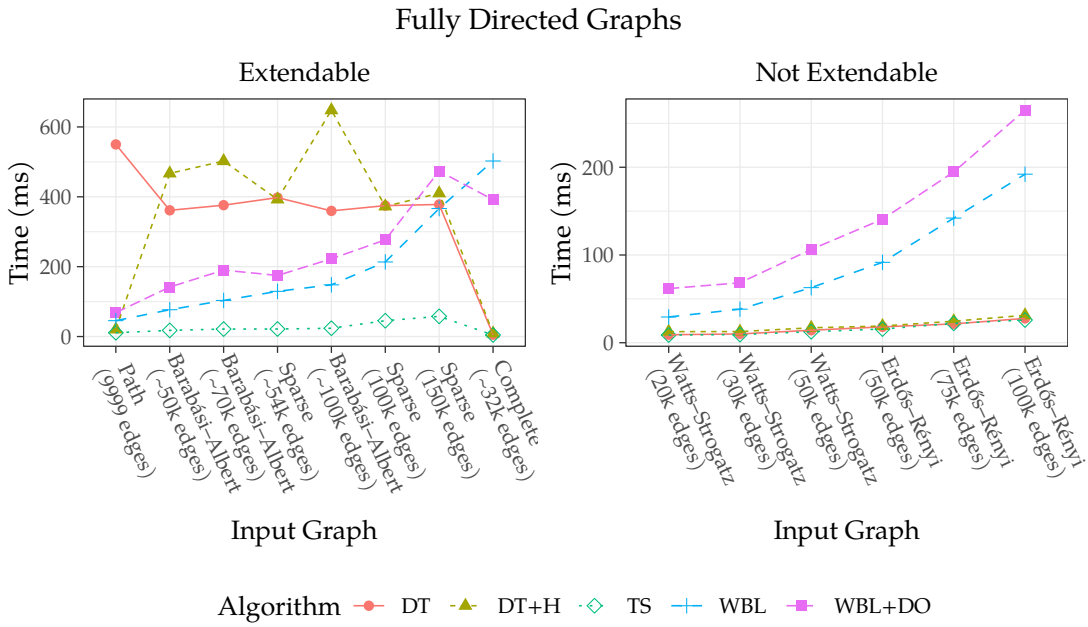


Figure 4.1: The results for fully directed input graphs. The plot on the left contains measurements for extendable input graphs while the right plot contains measurements for non-extendable input graphs only. The complete graph instance consists of 256 vertices while all other graphs are composed of roughly 10 000 vertices each.

Results for Directed Graphs

Directed graphs are an extreme case because they do not contain any undirected edge. Considering fully directed input graphs, each acyclic graph is already a consistent DAG extension of itself. On the other hand, input graphs containing cycles are not extendable.

Figure 4.1 depicts the results for directed input graphs. The plot on the left shows the runtimes for input graphs which are actually extendable while the plot on the right shows results for non-extendable input graphs. For both plots, the x-axis describes the input graph instances ordered by the runtime of algorithm WBL in ascending order and the y-axis denotes the runtime of the algorithms in milliseconds.

We begin by describing the plot on the left displaying the measurements for extendable directed input graphs. The first input graph is a path graph containing 10 000 vertices and 9 999 directed edges. All edges point in the same direction, i.e., the graph contains the arcs $(i, i+1)$ for $i \in \{1, 2, \dots, 9\,999\}$.

Next, there are two input graphs that have been generated using the Barabási–Albert model (Albert and Barabási, 2002). Both graphs consist of 10 000 vertices each. The first graph generated with the Barabási–Albert model has been generated with the parameter $k = 5$ and contains roughly 50 000 edges while the second graph has been generated with $k = 7$ and possesses about 70 000 edges.

Subsequently, there is a randomly generated sparse DAG with roughly 10 000 vertices and about 54 000 edges followed by another graph generated with the Barabási–Albert

model for the parameter $k = 10$ that consists of 10 000 vertices and about 100 000 edges. The following two graphs are again randomly generated sparse DAGs with roughly 10 000 vertices as well as 100 000 edges and 150 000 edges, respectively. Lastly, there is a complete graph consisting of 256 vertices and 32 640 edges.

We notice that the algorithm DT has a worse runtime than the algorithm WBL for every input graph instance except for the complete graph. The difference, however, shrinks the more edges the input graph contains. At some point above 150 000 edges, the algorithm DT starts to get faster than the algorithm WBL. Both algorithms DT+H and WBL+DO do mostly not improve the runtime of the standard implementations DT and WBL, respectively. Looking at the complete graph, the algorithms DT and DT+H are superior to the algorithms WBL and WBL+DO. Furthermore, the runtimes of both algorithms DT and DT+H are very close to the runtime of the linear-time algorithm TS for the complete graph. On the path graph, the algorithm DT has the worst runtime while both algorithms WBL and WBL+DO have similar runtimes as the algorithm TS.

The algorithm WBL outperforms DT on every input graph instance except for the complete graph. As the input graph gets more dense (i.e., contains more edges), the advantage declines and as soon as there are more than 150 000 edges in the input graph, algorithm DT overtakes WBL. We see that the update of the α- and β-values in the data structure used by the algorithms WBL and WBL+DO is expensive on dense graphs, especially on graphs containing large cliques as the results for the complete graph suggest.

Looking at the path graph, we clearly see that the algorithm DT is by far the slowest. In fact, the algorithm DT is even slower on the path graph than on the randomly generated sparse DAG with 150 000 edges. Observe that at any given intermediate step, there is only one potential sink in the path graph. The algorithm DT iterates over the vertices in ascending order, i.e., the only vertex being a potential sink is always found in the last iteration. Algorithm DT+H, however, is not affected by the problem of finding a potential sink in the last iteration and is even slightly faster than the algorithms WBL and WBL+DO. The path graph illustrates that the algorithm DT can become very slow with poorly chosen iteration orders, i.e., the performance of the algorithm DT depends heavily on the chosen iteration order.

Furthermore, the algorithm WBL+DO is mostly slower than the algorithm WBL despite a better worst-case complexity which might be explained by the fact that the input graphs store the neighbors of each vertex in a sorted list. That is, the edges of the input graphs are not inserted into the data structure in a random order by the algorithm WBL. We investigate the impact of the edge order in more detail later on in this chapter.

Although the overhead induced by the heuristic in algorithm DT+H is small, the algorithm DT+H is mostly slower than the algorithm DT. Since there are only directed edges in the input graphs, no neighborhoods have to be checked for being a clique and hence the heuristic does not provide a great benefit. It should be noted, however, that the heuristic seems to help reducing the chance of poorly selected iteration orders making the algorithm DT+H a more stable algorithm than the algorithm DT.

Next, we consider the plot on the right in Figure 4.1 constituted of measurements for non-extendable directed input graphs. The first three input graph instances are generated by the Watts-Strogatz model (Watts and Strogatz, 1998) and contain 10 000 vertices each. For the parameter β, we choose $\beta = 0.9$ to obtain a sufficient amount of randomness.

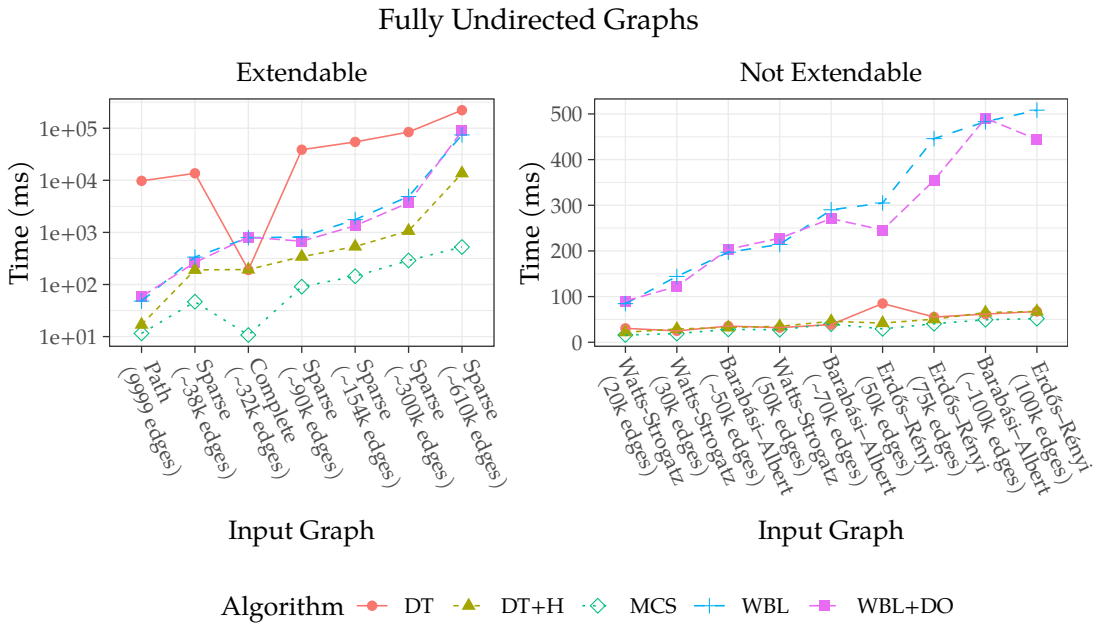


Figure 4.2: The results for fully undirected input graphs. The plot on the left contains measurements for extendable input graphs while the right plot shows measurements for non-extendable input graphs. The complete graph instance consists of 256 vertices while all other graphs are composed of roughly 10 000 vertices each.

However, the results for smaller β-values are similar to the results for $\beta = 0.9$. The first graph generated with the Watts-Strogatz model contains 20 000 edges and is generated with the parameter $k = 5$, the second graph has 30 000 edges and originates from $k = 7$, and there are 50 000 edges in the third graph with $k = 10$.

The remaining three input graphs are generated using the Erdős–Rényi model (Erdős and Rényi, 1959) with a fixed number of edges and contain 10 000 vertices each as well as 50 000 edges, 75 000 edges, and 100 000 edges, respectively.

We observe that both algorithms DT and DT+H are very fast and they outperform the algorithms WBL and WBL+DO. For all input graph instances, both algorithms DT and DT+H have a runtime close to the runtime of the linear-time algorithm TS.

Note that the algorithms DT and DT+H are able to stop immediately when no potential sink is found. The algorithms WBL and WBL+DO, however, always perform an additional pre-computation that is costly and does not provide a great benefit in case no potential sink can be found anymore at an early stage of the algorithm.

Results for Undirected Graphs

Fully undirected graphs are the other extreme case in the spectrum of possible input graph instances. They do not contain any directed edge and thus require the algorithms to orient *every* edge in the graph. As we have seen, fully undirected graphs are extendable if and only if they are chordal.

The results for fully undirected graphs can be found in Figure 4.2. As before, the plot on the left displays extendable input graph instances while the plot on the right shows non-extendable input graphs. The input graph instances depicted on the x-axis are again ordered by the runtime of algorithm WBL in ascending order. On the y-axis, we can see the runtime of the algorithms in milliseconds again. Note that the y-axis of the left plot uses a logarithmic scale.

Beginning with the plot on the left, we are looking at a path graph with 10 000 vertices and 9 999 edges followed by a randomly generated sparse chordal graph with 10 000 vertices and about 38 000 edges first. The randomly generated sparse chordal graph has been created with the subtree approach (Şeker et al., 2017) and the parameter $k = 3$. Subsequently, there is a complete graph consisting of 256 vertices and 32 640 edges.

The last four input graph instances are again randomly generated sparse chordal graphs with 10 000 vertices and between 90 000 and 610 000 edges. Only the graph containing about 610 000 edges has been generated by the perfect elimination order approach, with the parameter $k = 2$. The other three chordal graphs have been generated using the subtree approach with $k = 5$, $k = 7$, and $k = 10$, respectively.

We can see that the algorithm DT performs worst on all input graphs except for the complete graph. However, the more edges the input graph contains, the closer the runtime of the algorithm DT is to the runtime of the algorithms WBL and WBL+DO which have similar runtimes on all input graphs. The algorithm DT+H benefits from the heuristic and outperforms all of the other algorithms except for the linear-time algorithm MCS.

On the complete graph, the algorithms WBL and WBL+DO are the slowest with similar runtimes while the algorithms DT and DT+H have similar runtimes as well. The algorithm DT has the worst runtime on the path graph whereas the algorithm DT+H is even faster on the path graph than the algorithms WBL and WBL+DO.

As before with the directed input graph instances, the algorithms WBL and WBL+DO become slower with more edges in the input graph which may be attributed to the update of the α- and β-values of the data structure. However, the algorithms WBL and WBL+DO are still faster than the algorithm DT on the input graph with roughly 610 000 edges. Recall that the algorithm DT needs to verify that all undirected neighbors of a vertex v are adjacent to all other neighbors of v in order to check whether v is a potential sink. Hence, the more undirected edges the input graph contains, the more work the algorithm DT has to do in order to find a potential sink which explains why the algorithm DT is still slower than the algorithms WBL and WBL+DO on the input graph with roughly 610 000 edges.

The algorithm DT+H is faster than the algorithms DT, WBL, and WBL+DO on almost every input graph instance. On the complete graph, the algorithm DT is a few milliseconds faster than the algorithm DT+H.

Turning our attention to the right plot of Figure 4.2 showing non-extendable undirected input graphs, we are looking at three different types of input graphs. The first two graphs are generated by the Watts-Strogatz model. There are 10 000 vertices in each of those two graphs and they have been generated with the parameter $\beta = 0.9$ and different choices for the parameter k. For $k = 5$, there are 20 000 edges in the graph and for $k = 7$ we have 30 000 edges in the graph.

The subsequent graph has been generated using the Barabási–Albert model with the parameter $k = 5$ and contains 10 000 vertices as well as about 50 000 edges. Afterwards,

there is another graph generated with the Watts-Strogatz model using the parameters $\beta = 0.9$ and $k = 10$. The graph has 10 000 vertices and 50 000 edges.

Next, there is a graph generated with the Barabási–Albert model and the parameter $k = 7$ containing 10 000 vertices and about 70 000 edges. The following two graphs are generated utilizing the Erdős–Rényi model with a fixed number of edges and they consist of 10 000 vertices as well as 50 000 edges and 75 000 edges, respectively.

Subsequently, there is a graph generated with the Barabási–Albert model and the parameter $k = 10$ having 10 000 vertices and about 100 000 edges. The last graph has been generated with the help of the Erdős–Rényi model with a fixed number of edges and possesses 10 000 vertices as well as 100 000 edges.

Both algorithms DT and DT+H perform similarly well. The algorithms DT and DT+H are very fast and their performances are close to the performance of the linear-time algorithm MCS. The algorithms WBL and WBL+DO, however, are significantly slower on all input graph instances. Sometimes, the algorithm WBL+DO is slightly faster than the algorithm WBL but their runtimes are mostly close to each other.

The algorithms WBL and WBL+DO always perform a pre-computation of the α- and β-values in the data structure and update the α- and β-values after every vertex removal. Thus, the overhead for the additional computations necessary to keep the data structure up-to-date seems to dominate the runtime of the algorithms WBL and WBL+DO on non-extendable input graphs.

Results for Partially Directed Graphs

Partially directed graphs might contain both directed and undirected edges and are most relevant in practice. Especially sparse PDAGs that are extendable are important as they are most common in the real world.

Figure 4.3 shows the results for partially directed input graphs. All input graphs shown in Figure 4.3 contain about 50% directed edges and about 50% undirected edges. Once more, the measurements for extendable input graphs are depicted in the plot on the left while the plot on the right contains the results for non-extendable input graphs. The input graph instances depicted on the x-axis are again ordered by the runtime of the algorithm WBL in ascending order and the y-axis shows the runtime of the algorithms in milliseconds. The y-axis of the left plot uses a logarithmic scale.

We begin with a description of the plot on the left. Each input graph has been generated by creating an extendable undirected graph G and then orienting about 50% of the edges in G according to the orientation given by a consistent DAG extension of G.

The first input graph is a path graph with 10 000 vertices and 9 999 edges. Subsequently, there are two randomly generated sparse PDAGs with roughly 10 000 vertices as well as about 38 000 edges and about 90 000 edges, respectively. Both of these randomly generated sparse PDAGs originate from randomly generated undirected chordal graphs with the subtree approach. For the first sparse PDAG, the parameter $k = 3$ has been used and $k = 5$ has been the choice for the second graph.

Next, there is a complete graph with 256 vertices and 32 640 edges followed by another three randomly generated sparse PDAGs. The randomly generated sparse PDAGs are composed of roughly 10 000 vertices as well as about 154 000 edges, about 300 000

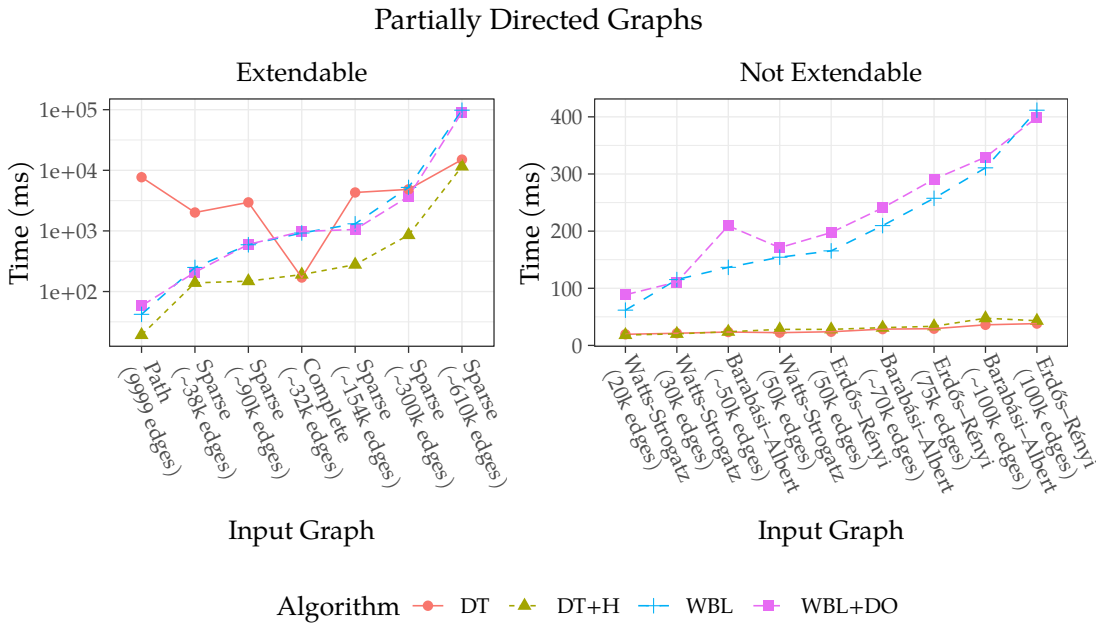


Figure 4.3: The results for partially directed input graphs with about 50% directed edges and about 50% undirected edges. The plot on the left shows measurements for extendable input graphs while the right plot contains measurements for non-extendable input graphs. The complete graph instance consists of 256 vertices while all other graphs are composed of roughly 10 000 vertices each.

edges, and about 610 000 edges, respectively. For the first two of those randomly generated sparse PDAGs, the underlying undirected chordal graph has been generated by the subtree approach with the parameter $k = 7$ and $k = 10$, respectively. The last sparse PDAG containing about 610 000 edges originates from an undirected chordal graph generated with the perfect elimination order approach and the parameter $k = 2$.

Overall, the algorithm DT+H performs quite well and is faster than the other algorithms on all input graph instances except for the complete graph. On the complete graph, the algorithm DT+H is only slightly slower than the algorithm DT. The algorithms WBL and WBL+DO do not differ very much and are always slower than the algorithm DT+H.

In comparison to the algorithm DT, both algorithms WBL and WBL+DO are faster on the randomly generated PDAGs with less than 300 000 edges. Above 300 000 edges, the tide turns and the algorithms WBL as well as WBL+DO become slower than the algorithm DT. Both algorithms WBL and WBL+DO are inferior to the algorithms DT and DT+H on the complete graph. Considering the path graph, the algorithm DT is by far the slowest algorithm while the algorithm DT+H is the fastest algorithm.

We can see that the algorithm DT overtakes the algorithm WBL from roughly 300 000 edges onward. Remember that for undirected graphs, the algorithm DT has been slower than the algorithm WBL on the input graph with about 610 000 edges. The results confirm that the greater the proportion of directed edges in the input graph, the faster the algorithm DT is. As the algorithm DT can skip the nested iterations over the neighbor-

hood of a vertex v if v has at least one outgoing arc when searching for a potential sink, more directed edges increase the chance for the algorithm DT to skip these iterations.

In the following, we examine the right graph of Figure 4.3 containing the results for non-extendable input graphs. Each input graph has been generated by creating an undirected graph and then randomly orienting about 50% of its edges.

The first two input graph instances are generated utilizing the Watts-Strogatz model with the parameters $\beta = 0.9$ as well as $k = 5$ and $k = 7$, respectively. Both graphs contain 10 000 vertices and the first graph possesses 20 000 edges while the second graph has 30 000 edges. The successive input graph is generated with the help of the Barabási–Albert model with the parameter $k = 5$ and has 10 000 vertices as well as about 50 000 edges.

It follows another input graph generated by the Watts-Strogatz model with the parameter $k = 10$ consisting of 10 000 vertices and 50 000 edges. The next graph has been generated using the Erdős–Rényi model with a fixed number of edges and is composed of 10 000 vertices as well as 50 000 edges. Then, there is a graph created with the Barabási–Albert model and the parameter $k = 7$ made up of 10 000 vertices and about 70 000 edges.

The subsequent graph has been generated by the Erdős–Rényi model with a fixed number of edges and consists of 10 000 vertices as well as 75 000 edges. The last two graphs have been generated with the Barabási–Albert model and the Erdős–Rényi model, respectively, and contain 10 000 vertices and about 100 000 edges. For the Barabási–Albert model, the parameter $k = 10$ has been chosen.

We can observe that the algorithms DT and DT+H do not differ very much and are the fastest on all input graphs. The runtime for both DT and DT+H increases only slowly with more edges in the graph. For the algorithms WBL and WBL+DO, however, the runtime increases more rapidly as there are more edges in the input graph. As before, the differences between the runtimes of the algorithm DT and the algorithm WBL may be explained by the fact that every additional edge causes considerable more work for the algorithm WBL while the additional amount of work for the algorithm DT might be neglected due to early cancellation of loops.

Until now, we have examined input graphs with roughly 50% directed edges and roughly 50% undirected edges. However, it is interesting to see what effect different proportions of directed edges have on the runtime of the algorithms. Figure 4.4 illustrates the effect of the proportion of directed edges on the runtimes of the algorithms.

There are measurements for three types of input graphs in Figure 4.4. Among those three types of input graphs are path graphs, complete graphs, and randomly generated sparse graphs with the same skeleton. As previously, the path graphs serve as worst-case input graph instances for the algorithm DT and the complete graphs are input graphs where the algorithm WBL performs exceptionally bad. The shown randomly generated sparse graphs yield representative results for randomly generated sparse graphs with different skeletons and different numbers of edges.

The first plot shows results for path graphs with 10 000 vertices and 9 999 edges, the second plot contains the results for complete graphs with 256 vertices and 32 640 edges, and the third plot consists of measurements for randomly generated sparse graphs with the same skeleton as well as 10 000 vertices and about 90 000 edges. On the x-axis of each plot, there are input graphs of the same graph type with different proportions of directed edges. The runtime for the algorithms is denoted on the y-axes of the plots and the y-axes

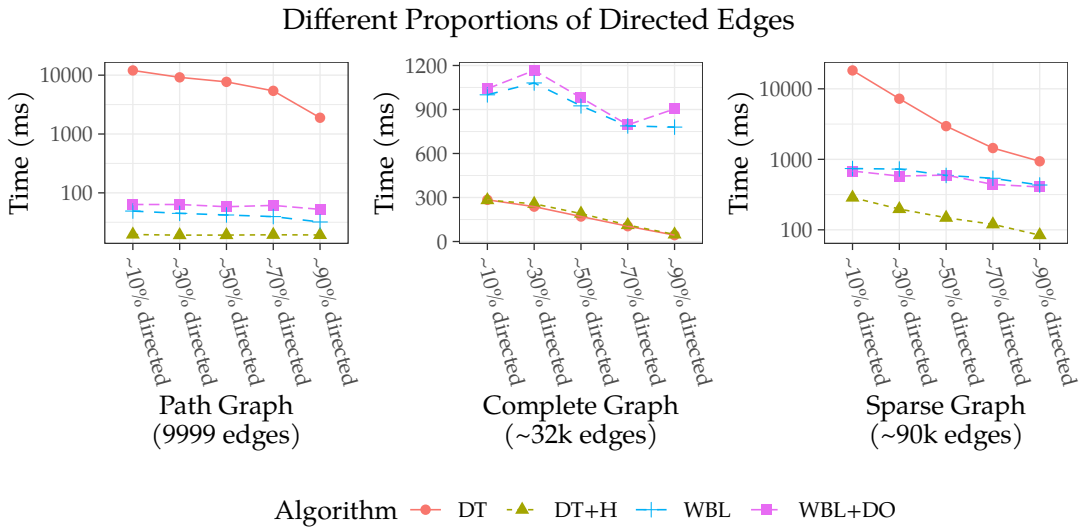


Figure 4.4: The effect of different proportions of directed edges on the runtimes of the algorithms. Each plot contains measurements for one type of input graphs with different proportions of directed edges. The first plot shows results for path graphs with 10 000 vertices and 9 999 edges, the second plot depicts the results for complete graphs with 256 vertices and 32 640 edges, and the third plot represents the results for randomly generated sparse graphs with the same skeleton and 10 000 vertices as well as about 90 000 edges.

of the first and the third plot use a logarithmic scale.

We observe that the algorithm DT benefits from a great proportion of the edges being directed, independently of the type of the input graph. Considering the sparse random graphs, the runtime of the algorithm DT even decreases by a factor of ten when comparing the input graph with 10% directed edges to the input graph where 90% of the edges are directed. The runtimes of the algorithms DT+H, WBL, and WBL+DO decrease only by a small amount when comparing input graphs with a small proportion of directed edges to input graphs with many of their edges being directed.

Although the algorithms WBL and WBL+DO are generally faster on input graphs with 90% directed edges than on input graphs with 10% directed edges, there is an increase in their runtime when comparing the complete graph with 10% directed edges to the complete graph with 30% directed edges. Such an increase might be due to the random component in the graph generation procedure as the directed edges for two graphs of the same type with different proportions of directed edges may not be the same.

The algorithm DT benefits the most from a great proportion of directed edges. Recall that the algorithm DT can avoid the nested loop over the neighborhood of a vertex when searching for a potential sink if that vertex has at least one outgoing arc. Hence, more directed edges in an input graph increase the chance for the algorithm DT to significantly reduce the work of finding a potential sink.

Summary of the Results

In the following, the findings of the experimental evaluation are briefly summarized. First of all, we have seen that there is no superior algorithm. This is quite remarkable because considering the worst-case time complexity of the algorithms, there is a reason to assume that the algorithm WBL should outperform the algorithm DT on almost every extendable input graph instance. However, despite a worse worst-case time complexity, the algorithm DT outperforms the algorithm WBL on very dense input graphs and on non-extendable input graphs. The algorithm DT has the possibility to become faster than the algorithm WBL because the algorithm DT might break out of its loops early.

The algorithm WBL, on the other hand, has no possibility for early termination and causes additional overhead in the initialization step. Especially updating the α- and β-values of the data structure is costly on dense graphs. Nevertheless, when considering extendable sparse graphs which are of high practical relevance, the algorithm WBL is always superior to the algorithm DT.

Although the algorithm DT may benefit from the early loop cancellation, there is no guarantee that its loops can be cancelled early. In other words, the algorithm DT carries the risk of choosing an unfavorable iteration order and may become very slow even on sparse input graphs as we have seen on the path graphs. For the path graph, the algorithm DT is slower than the algorithm WBL by a factor of 100 in some scenarios (e.g., see the path graph with 10% directed edges in Figure 4.4). The algorithm WBL, however, is only slower than the algorithm DT by a factor of four on the complete graphs in Figure 4.4.

The application of the heuristic deployed in the algorithm DT+H can help to avoid poorly chosen iteration orders when iterating over the vertices of the input graph. Employing the heuristic produces little overhead and strongly reduces the chance for poorly chosen iteration orders. The algorithm DT is only faster than the algorithm DT+H on some directed input graph instances that have little practical relevance.

When looking at the refinement of the algorithm WBL, namely WBL+DO, the results suggest that the computation of the degeneracy ordering performed in the algorithm WBL+DO does not provide a great benefit. However, an important thing to note is that all the input graph instances are encoded as a LightGraphs object and thus the neighbors of each vertex are stored in a sorted list. That is, the initialization of the data structure in the algorithm WBL does not insert edges in a random order. In order to determine whether the computation of the degeneracy ordering actually provides an advantage, the algorithm WBL+DO should be compared to a random insertion procedure for the algorithm WBL. Such a comparison can be found in Figure 4.5.

The input graph instances shown in Figure 4.5 are the same input graph instances as in Figure 4.3, i.e., they contain about 50% directed edges and about 50% undirected edges. The algorithm WBL, however, now inserts the edges of the input graphs into the data structure in a random order.

It can be clearly seen that the algorithm WBL+DO outperforms the algorithm WBL on all input graph instances (note that the y-axis of the first plot uses a logarithmic scale). The results presented in Figure 4.5 are representative for other proportions of directed edges as well. We can conclude that the computation of the degeneracy ordering performed in the algorithm WBL+DO is indeed worth the effort if the edges of the input

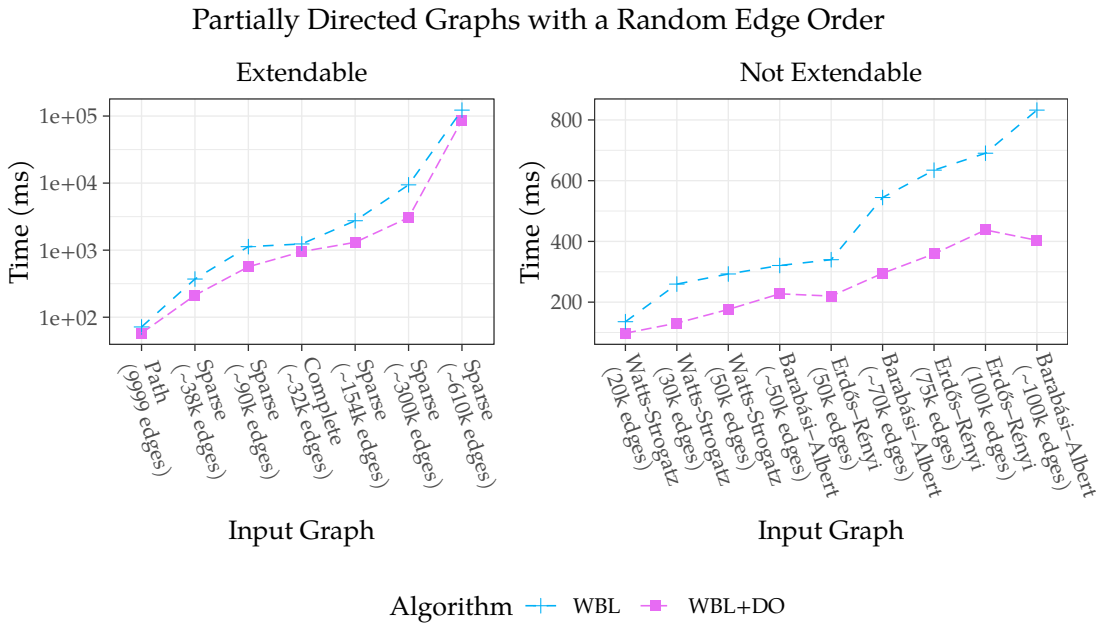


Figure 4.5: A comparison of the algorithms WBL and WBL+DO on input graphs with their edges being stored in a random order. All input graphs contain about 10 000 vertices except for the complete graph which contains 256 vertices. The left plot presents results for extendable input graph instances while the plot on the right consists of measurements for non-extendable input graph instances.

graph are not already sorted. Thus, if it is not known whether the edges of an input graph are sorted, the algorithm WBL+DO should be preferred over the algorithm WBL as the algorithm WBL+DO has a stable performance independent of edge orders.

4.4 Limitations of the Evaluation

We close this chapter by discussing limitations of this evaluation. Although the evaluated algorithms have been tested on several input graph instances, the dataset can still be enlarged. In particular, the generation approach to obtain extendable PDAGs introduced in Section 3.2 generates only PDAGs with chordal skeletons (i.e., PDAGs with no v-structures) to guarantee that the generated PDAG is extendable. It would be interesting to enlarge the dataset by adding extendable PDAGs with different numbers of v-structures to it. However, it should be noted that the evaluation has shown that the structure of an input graph is not as relevant as the number of vertices and edges in the input graph.

Furthermore, due to missing hardware capacities and time restrictions, there are no graphs with more than 100 000 vertices in the generated dataset. Even though graphs with more than 100 000 vertices are rare in practice, it would be interesting to see how well the evaluated algorithms perform on graphs with more than 100 000 vertices and different amounts of edges.

5

Conclusion

In this thesis, we conducted an experimental evaluation of the algorithms for computing consistent DAG extensions given by Dor and Tarsi (1992) and Wienöbst, Bannach, and Liśkiewicz (2021a). Furthermore, we performed a theoretical investigation of algorithms for enumerating all consistent DAG extensions of an input graph.

For the experimental evaluation, a benchmarking framework including efficient algorithms for the computation of consistent DAG extensions and a versatile dataset of input graph instances has been implemented. The algorithms for computing consistent DAG extensions were experimentally evaluated on more than 1 000 input graph instances and the framework contains graph generators which can be used to enlarge the dataset of input graph instances at any time.

The results of the experimental evaluation indicate that the algorithm DT is faster than the algorithms WBL and WBL+DO on non-extendable graphs and on dense graphs which is remarkable because the algorithm DT has a worse worst-case time complexity than the algorithms WBL and WBL+DO. However, the algorithms WBL and WBL+DO are both superior to the algorithm DT on extendable sparse graphs which belong to the class of graphs that are relevant in practice. It is also noteworthy that we can observe scenarios in which the algorithm DT is slower than the algorithms WBL and WBL+DO by a factor of about 100.

Consequently, the algorithm DT is unsuitable in practice. Two modifications of the algorithm DT have been proposed that both outperform the algorithm DT in most evaluated scenarios. Out of those two modifications, we highlight the algorithm DT+H which uses a heuristic when searching for a potential sink. The algorithm DT+H applies a heuristic that induces only a small overhead while simultaneously reducing the chance for exceptionally bad performances. The results of the experimental evaluation demonstrate that the algorithm DT+H outperforms the algorithms DT, WBL, and WBL+DO on many practically relevant input graph instances.

Furthermore, the experimental evaluation illustrated that the algorithms WBL and WBL+DO guarantee a stable performance on different types of input graph instances. That is, there are no exceptionally bad performances when comparing different types of graphs with the same number of vertices and roughly the same number of edges. For the algorithm DT, however, we have seen that the runtime skyrockets on some types of input graphs such as path graphs with a badly chosen iteration order. We can conclude

that the algorithm DT yields a volatile performance while the algorithm DT+H is more stable than the algorithm DT but there is no guarantee for a stable performance as in the algorithms WBL and WBL+DO.

The comparison of the algorithms WBL and WBL+DO showed that the order of edge insertions into the internal data structure of the algorithms has indeed an impact on their runtime. We have seen that the algorithm WBL+DO outperforms the algorithm WBL in every scenario if the algorithm WBL inserts the edges from the input graph into its data structure in a random order. However, if the edges of the input graph are inserted into the data structure in a sorted order, the lead of the algorithm WBL+DO diminishes and the algorithm WBL is even faster than the algorithm WBL+DO in some scenarios. In summary, we generally recommend the usage of the algorithm DT+H or, in case a guarantee for a stable performance is desired, the application of the algorithm WBL+DO. Currently, software packages for analyzing causal relationships that include algorithms for computing consistent DAG extensions implement the algorithm by Dor and Tarsi (1992) without applying a heuristic (Scutari, 2010; Kalisch et al., 2012).

Considering the problem of enumerating all consistent DAG extensions of an input graph, we started with the investigation of solving this problem for arbitrary PDAG inputs. Although Meek (1995) already hinted at an algorithm for solving the enumeration problem, the algorithm was not described in detail. We described precisely how a possible implementation of the algorithm could look like and gave an analysis of its runtime.

Furthermore, we argued that the delay between the output of two consecutive consistent DAG extensions of the algorithm can be bounded by $O(\Delta m^3)$ for CPDAG inputs. Going one step further, an algorithm for enumerating all consistent DAG extensions of a CPDAG that reduces the delay to $O(nm)$ was presented in this thesis.

It remains an interesting question for future work whether the delay for CPDAGs can be further reduced to a linear-time delay such as $O(n + m)$. Possibly, the LBFS algorithm can be modified to compute not only one consistent DAG extension but instead all consistent DAG extensions of a CPDAG with a linear-time delay. Another promising approach might be to compute any consistent DAG extension G of the input CPDAG in linear time first and then apply the transformational characterization given by Chickering (1995) where so-called covered edges in G are being reversed to obtain DAGs which are Markov equivalent to G. Naturally, the question whether the delay between the output of two consecutive consistent DAG extensions for an arbitrary PDAG can be bounded by $O(nm)$ is relevant for future research as well.

Bibliography

Abu-Khzam, F.N., Langston, M.A., Mouawad, A.E., and Nolan, C.P. (2010). A Hybrid Graph Representation for Recursive Backtracking Algorithms. In Proceedings of the 4th International Conference on Frontiers in Algorithmics, FAW'10. Wuhan, China: Springer-Verlag, pp. 136–147. ISBN: 3642145523.

Albert, R. and Barabási, A.-L. (2002). Statistical mechanics of complex networks. Reviews of Modern Physics *74*, 47–97. ISSN: 1539-0756. DOI: 10.1103/revmodphys.74.47. URL: http://dx.doi.org/10.1103/RevModPhys.74.47.

Andersson, S.A., Madigan, D., and Perlman, M.D. (1997). A characterization of Markov equivalence classes for acyclic digraphs. The Annals of Statistics *25*, 505 –541. DOI: 10.1214/aos/1031833662. URL: https://doi.org/10.1214/aos/1031833662.

Andreou, M.I., Papadopoulou, V.G., Spirakis, P.G., Theodorides, B., and Xeros, A. (2005). Generating and Radiocoloring Families of Perfect Graphs. In Experimental and Efficient Algorithms, (Nikoletseas, S.E., ed.). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 302–314. ISBN: 978-3-540-32078-4.

Chalak, K. and White, H. (Jan. 2007). An Extended Class of Instrumental Variables for the Estimation of Causal Effects. Boston College Department of Economics, Boston College Working Papers in Economics.

Chickering, D.M. (1995). A Transformational Characterization of Equivalent Bayesian Network Structures. In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, UAI'95. Montréal, Qué, Canada: Morgan Kaufmann Publishers Inc., pp. 87–98. ISBN: 1558603859.

Chickering, D.M. (Mar. 2002). Learning Equivalence Classes of Bayesian-Network Structures. J. Mach. Learn. Res. *2*, 445–498. ISSN: 1532-4435. DOI: 10.1162/153244302760200696. URL: https://doi.org/10.1162/153244302760200696.

Chickering, D.M. (Mar. 2003). Optimal Structure Identification with Greedy Search. J. Mach. Learn. Res. *3*, 507–554. ISSN: 1532-4435. DOI: 10.1162/153244303321897717. URL: https://doi.org/10.1162/153244303321897717.

Dor, D. and Tarsi, M. (1992). A simple algorithm to construct a consistent extension of a partially oriented graph. Technicial Report R-185, Cognitive Systems Laboratory, UCLA.

Erdős, P. and Rényi, A. (1959). On Random Graphs I. Publicationes Mathematicae Debrecen *6*, 290–297.

Erdős, P. and Rényi, A. (1960). On the Evolution of Random Graphs. In Publications of the Mathematical Institute of the Hungarian Academy of Sciences, pp. 17–61.

Fulkerson, D.R. and Gross, O.A. (1965). Incidence matrices and interval graphs. Pacific Journal of Mathematics *15*, 835 –855. DOI: pjm/1102995572. URL: https://doi.org/.

Greenland, S., Pearl, J., and Robins, J.M. (1999). Causal Diagrams for Epidemiologic Research. Epidemiology *10*. ISSN: 1044-3983. URL: https://journals.lww.com/epidem/Fulltext/1999/01000/Causal_Diagrams_for_Epidemiologic_Research.8.aspx.

Hauser, A. and Bühlmann, P. (Aug. 2012). Characterization and Greedy Learning of Interventional Markov Equivalence Classes of Directed Acyclic Graphs. J. Mach. Learn. Res. *13*, 2409–2464. ISSN: 1532-4435.

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. URL: http://jmlr.org/papers/v16/he15a.html.

Kahn, A.B. (Nov. 1962). Topological Sorting of Large Networks. Commun. ACM *5*, 558–562. ISSN: 0001-0782. DOI: 10.1145/368996.369025. URL: https://doi.org/10.1145/368996.369025.

Kalisch, M., Mächler, M., Colombo, D., Maathuis, M.H., and Bühlmann, P. (2012). Causal Inference Using Graphical Models with the R Package pcalg. Journal of Statistical Software, Articles *47*, 1–26. ISSN: 1548-7660. DOI: 10.18637/jss.v047.i11. URL: https://www.jstatsoft.org/v047/i11.

Maathuis, M.H., Kalisch, M., and Bühlmann, P. (2009). Estimating high-dimensional intervention effects from observational data. The Annals of Statistics *37*, 3133 –3164. DOI: 10.1214/09-AOS685. URL: https://doi.org/10.1214/09-AOS685.

Matula, D.W. and Beck, L.L. (July 1983). Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. J. ACM *30*, 417–427. ISSN: 0004-5411. DOI: 10.1145/2402.322385. URL: https://doi.org/10.1145/2402.322385.

Meek, C. (1995). Causal Inference and Causal Explanation with Background Knowledge. In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, UAI'95. Montréal, Qué, Canada: Morgan Kaufmann Publishers Inc., pp. 403–410. ISBN: 1558603859.

Morgan, S.L. and Winship, C. (2014). Counterfactuals and Causal Inference: Methods and Principles for Social Research. 2nd ed. Analytical Methods for Social Research. Cambridge University Press. DOI: 10.1017/CBO9781107587991.

Pearl, J. (2009). Causality: Models, Reasoning and Inference. 2nd ed. Cambridge University Press. ISBN: 9780511803161. DOI: 10.1017/CBO9780511803161.

Perkovic, E. (2020). Identifying causal effects in maximally oriented partially directed acyclic graphs. In Proceedings of the 36th Conference on Uncertainty in Artificial Intelligence (UAI), (Peters, J. and Sontag, D., eds.). Vol. 124. Proceedings of Machine Learning Research. PMLR, pp. 530–539. URL: http://proceedings.mlr.press/v124/perkovic20a.html.

Ravasz, E., Somera, A.L., Mongru, D.A., Oltvai, Z.N., and Barabási, A.-L. (2002). Hierarchical Organization of Modularity in Metabolic Networks. Science *297*, 1551–1555. ISSN: 0036-8075. DOI: 10.1126/science.1073374. eprint: https://science.sciencemag.org/content/297/5586/1551.full.pdf. URL: https://science.sciencemag.org/content/297/5586/1551.

Rose, D., Tarjan, R., and Lueker, G.S. (1976). Algorithmic Aspects of Vertex Elimination on Graphs. SIAM J. Comput. *5*, 266–283.

Scutari, M. (2010). Learning Bayesian Networks with the bnlearn R Package. Journal of Statistical Software, Articles *35*, 1–22. ISSN: 1548-7660. DOI: 10.18637/jss.v035.i03. URL: https://www.jstatsoft.org/v035/i03.

Şeker, O., Heggernes, P., Ekim, T., and Taşkın, Z.C. (2017). Linear-Time Generation of Random Chordal Graphs. In Algorithms and Complexity, (Fotakis, D., Pagourtzis, A., and Paschos, V.T., eds.). Cham: Springer International Publishing, pp. 442–453. ISBN: 978-3-319-57586-5.

Spirtes, P., Glymour, C., and Scheines, R. (Jan. 1993). Causation, Prediction, and Search. Vol. 81. ISBN: 978-1-4612-7650-0. DOI: 10.1007/978-1-4612-2748-9.

Talvitie, T. and Koivisto, M. (2019). Counting and Sampling Markov Equivalent Directed Acyclic Graphs. Proceedings of the AAAI Conference on Artificial Intelligence *33*, 7984–7991. DOI: 10.1609/aaai.v33i01.33017984. URL: https://ojs.aaai.org/index.php/AAAI/article/view/4799.

Tarjan, R. and Yannakakis, M. (1984). Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. SIAM J. Comput. *13*, 566–579.

Tennant, P.W.G., Murray, E.J., Arnold, K.F., Berrie, L., Fox, M.P., Gadd, S.C., Harrison, W.J., Keeble, C., Ranker, L.R., Textor, J., et al. (Dec. 2020). Use of directed acyclic graphs (DAGs) to identify confounders in applied health research: review and recommendations. International Journal of Epidemiology *50*, 620–632. ISSN: 0300-5771. DOI: 10.1093/ije/dyaa213. eprint: https://academic.oup.com/ije/article-pdf/50/2/620/37947634/dyaa213.pdf. URL: https://doi.org/10.1093/ije/dyaa213.

Textor, J., Zander, B. van der, Gilthorpe, M.S., Liśkiewicz, M., and Ellison, G.T. (Jan. 2017). Robust causal inference using directed acyclic graphs: the R package 'dagitty'. International Journal of Epidemiology *45*, 1887–1894. ISSN: 0300-5771. DOI: 10.1093/ije/dyw341. eprint: https://academic.oup.com/ije/article-pdf/45/6/1887/11120744/dyw341.pdf. URL: https://doi.org/10.1093/ije/dyw341.

Verma, T. and Pearl, J. (1990). Equivalence and Synthesis of Causal Models. In Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence, UAI '90. USA: Elsevier Science Inc., pp. 255–270. ISBN: 0444892648.

Verma, T. and Pearl, J. (1992). An Algorithm for Deciding if a Set of Observed Independencies Has a Causal Explanation. Uncertainty in Artificial Intelligence, (Dubois, D., Wellman, M.P., D'Ambrosio, B., and Smets, P., eds.). Morgan Kaufmann, pp. 323–330. ISBN: 978-1-4832-8287-9. DOI: https://doi.org/10.1016/B978-1-4832-8287-9.50049-9. URL: https://www.sciencedirect.com/science/article/pii/B9781483282879500499.

Watts, D.J. and Strogatz, S.H. (1998). Collective dynamics of 'small-world' networks. Nature *393*, 440–442. ISSN: 1476-4687. DOI: 10.1038/30918. URL: https://doi.org/10.1038/30918.

Wienöbst, M., Bannach, M., and Liśkiewicz, M. (2021a). Extendability of Causal Graphical Models: Algorithms and Computational Complexity. Conference on Uncertainty in Artificial Intelligence (UAI).

Wienöbst, M., Bannach, M., and Liśkiewicz, M. (2021b). Polynomial-Time Algorithms for Counting and Sampling Markov Equivalent DAGs. Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI'21), 12198–12206.

Williams, T.C., Bach, C.C., Matthiesen, N.B., Henriksen, T.B., and Gagliardi, L. (2018). Directed acyclic graphs: a tool for causal studies in paediatrics. Pediatric Research *84*, 487–493. ISSN: 1530-0447. DOI: 10.1038/s41390-018-0071-3. URL: https://doi.org/10.1038/s41390-018-0071-3.