ECAI 2012
Luc De Raedt et al. (Eds.)
© 2012 The Author(s).
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-098-7-366

Process Discovery via Precedence Constraints

Gianluigi Greco[1] and **Antonella Guzzo**[2] and **Luigi Pontieri**[3]

Abstract. A key task in process mining consists of building a graph of causal dependencies over process activities, which can then be used to derive more expressive models in some high-level modeling language. An approach to accomplish this task is presented where the learning process can exploit the background knowledge that, in many cases, is available to the analysts taking care of the process (re-)design. The method is based on encoding the information gathered from the log and the (possibly) given background knowledge in terms of *precedence constraints*, i.e., constraints over the topology of the graphs. Learning algorithms are eventually formulated in terms of reasoning problems over precedence constraints, and the computational complexity of such problems is thoroughly analyzed by tracing their tractability frontier. The whole approach has been implemented in a prototype system leveraging a solid constraint programming platform, and results of experimental activity are reported.

1 Introduction

By analyzing a set of traces registering the sequence of tasks performed along several enactments of a transactional system, the goal of process discovery techniques is to derive a model explaining all the episodes recorded in it [13]. Even though such techniques are receiving increasing attention in the research literature, they are still at an early stage of adoption within enterprises. Indeed, analysts are likely to prefer traditional "top-down" design approaches, where models are eventually built by refining and formalizing a number of desiderata and specifications reflecting the prior knowledge they possess about the process to be automatized. Such prior knowledge is, in fact, neglected by current "bottom-up" process discovery techniques, which automatically derive a process model by just focusing on the statistics hidden in the data at hand. As a result, mined models may well violate conceptual specifications and domain-constraints, hence turning out to be useless in real-life applications.

Top-down design methods and bottom-up process discovery techniques have been completely separate worlds, so far. However, the use of background knowledge to improve the quality of results has already been considered in a number of traditional data mining tasks (on relational data and on sequence data), such as pattern mining [9] and clustering [4]. Unfortunately, such techniques have not found a systematic counter-part in the process mining setting. In particular, the issue of studying mechanisms to interface the world of traditional process design approach and the world of process discovery algorithms (and which can benefit of the advantages of both perspectives) has been largely unexplored in the literature—see Section 6, for an overview of existing approaches.

[1] Dipartimento di Matematica, Università della Calabria, I-87036 Rende, Italy, email: ggreco@mat.unical.it
[2] D.E.I.S., Università della Calabria, I-87036 Rende, Italy, email: guzzo@deis.unical.it
[3] ICAR-CNR, I-87036 Rende, Italy, email: pontieri@icar.cnr.it

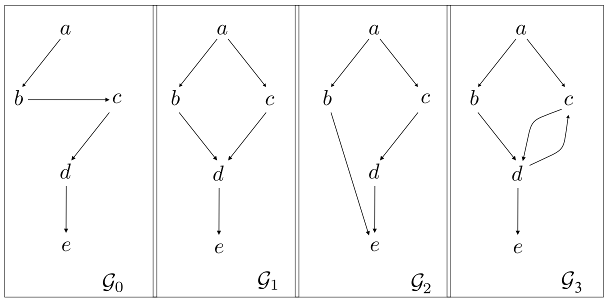

Figure 1. Dependency graphs in the running example.

Process Discovery. No matter of the specific process-oriented features being supported, process discovery algorithms can be abstractly seen as sequentially carrying out two different sub-tasks: First, they analyze the log and apply some form of reasoning to learn the causal dependencies that are likely to hold among the activities of the process, which are often presented in form of *log-based ordering relations* [14].Then, they exploit the knowledge thereby acquired within mining algorithms that take into account advanced facets of process enactments and return process models formalized in expressive modeling languages (such as *Petri nets* [14]). In the following, we focus on the former of the two tasks, by elaborating techniques to mine causal dependencies from process logs. Accordingly, the output of such techniques are not (full) process models, but *dependency graphs*, i.e., directed graphs whose nodes one-to-one correspond with the activities and such that an edge from an activity a to an activity b means that, in some enactment, we expect that an actual flow of information can occur from a to b. For example, the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Figure 1 are two possible dependency graphs for the traces $abcde$ and $acbde$. Instead, $\mathcal{G}_0$ does not properly reflect the flow associated with the trace $acbde$, where b actually occurs after c.

Dependency-graph discovery is a challenging problem in the case of concurrent processes, as traces flatten all the information related to the execution of "parallel" activities. Indeed, in the period of time elapsing between the execution of two activities, with one requiring some output produced by the other, the system might register the interleaved execution of other activities involved over different branches of the process. Eventually, the fact that an activity always precedes another over the given traces might be by chance, and it does not necessarily witness a causal dependency between them. For example, from the fact that b precedes d in the two traces $abcde$ and $acbde$, we cannot infer that b is a pre-requisite for d: In principle, b can be executed over a different branch and without any causal dependency with d, and hence $\mathcal{G}_2$ might be a valid control flow graph.

Contribution. Since several dependency graphs can in principle be associated with a given log by discovery algorithms, the background knowledge available to the analyst would naturally play a role here, in order to single out, among the graphs modeling the statistics in the data, the one that best fits process requirements and specifications. In fact, a 'hybrid" approach to process discovery is particularly useful if the log at hand is far from being *complete*, i.e., when several possible process behaviors are not registered in it, so that the knowledge of the analyst can compensate the lack of data available for mining activities. For instance, in the example discussed above, if it is a priori known that b and d are parallel activities, then we can discard $\mathcal{G}_1$ as a dependency graph, even though no trace is given where d actually occurred before b. The goal of this paper is precisely to elaborate dependency-graph discovery algorithms that can benefit from the availability of such prior knowledge. In more detail[4]:

- ▷ We propose a formal framework to specify additional properties on the dependency graphs that can be produced as output by process mining algorithms. The framework is based on expressing a set of *precedence constraints* over sets of activities. For instance, one can look for dependency graphs where certain precedences over activity sets hold or do not hold, directly or transitively, or where certain activities are parallel.
- ▷ As a result of our formulation, process discovery is conceptually carried out via a *learning task* (i.e., building all possible dependency graphs for a given input log) followed by a *reasoning task* (i.e., to filter out those graphs that do not satisfy the precedence constraints defined by the analyst). In general, exponentially many dependency graphs might be built in the learning phase, which would make a literal implementation of such a two-phase approach unfeasible. In fact, we show that the learning task can be *declaratively* formulated in the same language (of precedence constraints) used for the reasoning task. This addresses the above drawback, by leading to a common environment where the two tasks are combined synergically and carried out simultaneously.
- ▷ We analyze the computational complexity of the setting, by considering various qualitative properties on the kinds of constraint being allowed, and by tracing the tractability frontier w.r.t. them.
- ▷ All the techniques discussed in the paper have been implemented and integrated in a prototype system, where the task of reasoning about precedence constraints is delegated to a well-known constraint solver system available in the literature. Results for the experimental activity we have conducted in order to validate the effectiveness of the proposed approach are also reported.

2 Process Logs and Dependency Graphs

In this section, we recall a representation of process logs which is commonly adopted in process mining (see, e.g., [14, 8]).

Logs of Acyclic Processes. Let $\mathcal{A}$ be an alphabet of symbols, univocally identifying the *activities* of some underlying process. A *process instance* $\mathcal{I}$ over $\mathcal{A}$ is a directed acyclic graph (V, E) with $V \subseteq \mathcal{A}$ and where a distinguished activity $a_\perp \in V$ exists from which every other activity can be reached—intuitively, $a_\perp$ is the starting activity for the enactment of $\mathcal{I}$. A trace t is a string over $\mathcal{A}$, and hence has the form $t[1]t[2]...t[n]$, with $t[i] \in \mathcal{A}$ being an activity for each $i \in \{1, ..., n\}$. If t is a topological sort of the process instance $\mathcal{I}$, then t is called a *trace of* $\mathcal{I}$, denoted by $\mathcal{I} \vdash t$; in this case, t does not contain multiple occurrences of the same activity, i.e., $t[i] \neq t[j]$, for each $i \neq j$. A *log* L is a multi-set of traces. For a log L, $\mathcal{A}(L)$ is the set of all the activities occurring over the traces in L. W.l.o.g., we hereinafter assume that $t[1] = a_\perp$, for each trace $t \in L$.

[4] Due to space constraints, the reader is referred to [7], for full proofs, details on the encoding, and further experimental activity.

In process mining, the goal is to derive a process model supporting the enactments of the traces of a log L. Several algorithms have been proposed to this end (see [13] and the references therein). One of their crucial abilities is to discover causal precedences among activities in $\mathcal{A}(L)$, which we encode via directed graphs as follows.

Definition 1. Let L be a log, where no trace contains multiple occurrences of the same activity. Then, a graph $\mathcal{G} = (V, E)$ is a *support-graph for* L, denoted by $\mathcal{G} \vdash L$, if for each $t \in L$, there is a subgraph $\mathcal{I}$ of $\mathcal{G}$ such that $\mathcal{I}$ is a process instance over $\mathcal{A}(L)$ and $\mathcal{I} \vdash t$. □

Example 2. Consider the trace $t_0 = abcde$ over the set of activities $\mathcal{A}(\{t_0\}) = \{a, b, c, d, e\}$. Then, the graphs $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ reported in Figure 1 are such that $\mathcal{G}_0 \vdash \{t_0\}$, $\mathcal{G}_1 \vdash \{t_0\}$, and $\mathcal{G}_2 \vdash \{t_0\}$. ◁

Arbitrary Logs. A string t containing multiple occurrences of the same activity cannot be a trace of any process instance $\mathcal{I}$, as $\mathcal{I}$ is acyclic. Thus, if the underlying process involves loops, we need a mechanism to virtually unfold them: For each trace t, let $\bar{t}$ denote the trace obtained from t by substituting, with the fresh (virtual) activity $a\langle i\rangle$, the i-th occurrence in t of any activity a. Moreover, for a log L, let $\bar{L} = \{\bar{t} \mid t \in L\}$. Then, we say that a graph $\mathcal{G} = (V, E)$ is the *folding* of a graph $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ with $\bar{V} = \mathcal{A}(\bar{L})$ if $V = \{a \mid a\langle i\rangle \in \bar{V}\} = \mathcal{A}(L)$ and $E = \{(a, b) \mid (a\langle i\rangle, b\langle j\rangle) \in \bar{E}\}$.

Definition 3. Let L be a log. Then, a *dependency graph (DG)* $\mathcal{G}$ *for* L is the folding of a graph $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \vdash \bar{L}$. □

Note that acyclic dependency graphs for L exist if, and only if, L contains no trace with repetitions of activities. Moreover, $\mathcal{G}$ is an acyclic dependency graph for L if, and only if, $\mathcal{G} \vdash L$.

Example 4. $\mathcal{G}_0$, $\mathcal{G}_1$, and $\mathcal{G}_2$ are dependency graphs for $\{t_0\}$. Instead, they are not dependency graphs for $\{t_1\}$, with $t_1 = abcdcde$. Indeed, these graphs are acyclic while c and d occur twice in t_1.

Note that the trace $\bar{t}_1$ is the string $a\langle 1\rangle b\langle 1\rangle c\langle 1\rangle d\langle 1\rangle c\langle 2\rangle d\langle 2\rangle e\langle 1\rangle$, which is a topological sort of the acyclic graph $\bar{\mathcal{G}} = (\bar{V}, \bar{E})$ with $\bar{V} = \mathcal{A}(\{\bar{t}_1\})$ and $\bar{E} = \{(a\langle 1\rangle, b\langle 1\rangle), (a\langle 1\rangle, c\langle 1\rangle), (d\langle 1\rangle, c\langle 2\rangle), (c\langle 2\rangle, d\langle 2\rangle), (b\langle 1\rangle, d\langle 2\rangle), (d\langle 2\rangle, e\langle 1\rangle)\}$. Thus, $\bar{\mathcal{G}} \vdash \{\bar{t}_1\}$. Eventually, note that the graph $\mathcal{G}_3$ depicted in Figure 1 is the folding of $\bar{\mathcal{G}}$. It follows that $\mathcal{G}_3$ is a dependency graph for $\{t_1\}$. ◁

3 Precedence Constraints for Process Discovery

Let $\mathcal{A}$ be a set of activities. A *precedence constraint* over $\mathcal{A}$ is an assertion aimed at expressing a relationship of precedence among some of the activities in $\mathcal{A}$. To define the syntax, we distinguish positive and negative constraints. A positive constraint π is either an expression of the form $S \to a$ (called *edge* constraint), or an expression of the form $S \rightsquigarrow a$ (called *path* constraint), where $S \subseteq \mathcal{A}$, with $|S| \geq 1$, is a non-empty set of activities and $a \in \mathcal{A}$ is an activity. For a positive constraint π, $\neg\pi$ is a *negative* precedence constraint.

Precedence constraints are interpreted over directed graphs as follows. Let $\mathcal{G} = (V, E)$ be a directed graph such that $V \subseteq \mathcal{A}$ and $E \subseteq \mathcal{A} \times \mathcal{A}$. Then,

(1) $\mathcal{G}$ satisfies an edge constraint $S \to a$, if there is an activity $a_0 \in S$ such that $(a_0, a) \in E$;
(2) $\mathcal{G}$ satisfies a path constraint $S \rightsquigarrow a$, if there is a sequence of activities $a_0, a_1, ..., a_n = a$, with $n > 0$, such that $a_0 \in S$ and $(a_i, a_{i+1}) \in E$, for each i with $0 \leq i < n$;
(3) $\mathcal{G}$ satisfies a negated constraint $\neg\pi$, if $\mathcal{G}$ does not satisfy π.

If $\mathcal{G}$ satisfies each constraint in a set Π of precedence constraints, we say that $\mathcal{G}$ is *model* of Π, denoted by $\mathcal{G} \models \Pi$.

A foundational task in process mining consists of automatically building a dependency graph $\mathcal{G}$ for some log L given as input. In this context, precedence constraints can be naturally exploited to formalize additional requirements that dependency graphs discovered from L have to satisfy. This gives rise to the following two problems (which for $\Pi = \emptyset$ reduce to the standard ones considered in the literature), where we explicitly distinguish the variant where the desired dependency graph is required to be acyclic.

DG-MINING: Given a log L and a set Π of precedence constraints over $\mathcal{A}(L)$, compute a dependency graph $\mathcal{G}$ for L with $\mathcal{G} \models \Pi$.

ACYCLIC-DG-MINING: Given a log L and a set Π of precedence constraints over $\mathcal{A}(L)$, compute an acyclic dependency graph $\mathcal{G}$ for L with $\mathcal{G} \models \Pi$.

Note that, as an acyclic dependency graph can be just viewed as a partial order over the given set of activities, ACYCLIC-DG-MINING for $\Pi = \emptyset$ shares some technical similarities with the problem of discovering a *partial order* from sequential data (see, e.g., [10]). Thus, while having a completely different focus, our framework can be viewed as a generalization of this latter setting to handle user-specified constraints—when cycles are admitted (in the DG-MINING problem), the two settings are completely different instead.

Example 5. Consider the set $\Pi_0 = \{ \neg(\{b\} \rightsquigarrow d), \neg(\{d\} \rightsquigarrow b)\}$ of precedence constraints, stating that b and d are "parallel" activities. Then, consider the trace $t_0 = abcde$ of Example 2 and the graphs in Figure 1. Note that $\mathcal{G}_2$ is a model of Π_0, while $\mathcal{G}_0$ and $\mathcal{G}_1$ are not as they violate the constraint $\neg(\{b\} \rightarrow d)$. Thus, $\mathcal{G}_2$ is a solution to DG-MINING (on input $\{t_0\}$ and Π_0). In fact, it is also solution to ACYCLIC-DG-MINING. $\lhd$

The problems defined above comprise a learning task (dependency graph mining) and a reasoning task (to check whether a graph satisfies precedence constraints). In fact, we next show that even the learning task can be *declaratively* formulated in terms of reasoning about precedence constraints, thus defining a common framework where the two tasks are synergically combined and might be simultaneously carried out. The basic idea is to characterize the notion of support (in Definition 1) in terms of precedence constraints.

Definition 6 (Logs$\mapsto$Constraints). Let L be a log. For each trace $t[1]...t[n] \in L$, let $\pi(t) = \{ \{t[1], ..., t[i-1]\} \rightarrow t[i] \mid 1 < i \leq n \}$. Moreover, let $\pi(L) = \bigcup_{t \in L} \pi(t)$. $\square$

Intuitively, we just state that each activity in the trace t can be directly reached by at least one of its predecessors in t. This suffices to precisely characterize the notion of dependency graph, as illustrated below. We start with the case of processes with no loops.

Proposition 7. *Let L be a log where no trace contains multiple occurrences of the same activity. Let $\mathcal{G}$ be a graph (resp., acyclic graph) over $\mathcal{A}(L)$, and Π be a set of precedence constraints over $\mathcal{A}(L)$. Then,* (1) *$\mathcal{G}$ is a solution to* DG-MINING *(resp.,* ACYCLIC-DG-MINING*) on input L and Π* $\iff$ (2) $\mathcal{G} \models \pi(L) \cup \Pi$.

Proof Sketch. (1)⇒(2). Assume that *(1)* holds, i.e., $\mathcal{G} \vdash L$ and $\mathcal{G} \models \Pi$. In particular, for each trace $t[1]...t[n] \in L$, there is a subgraph $\mathcal{I}$ of $\mathcal{G}$ such that $\mathcal{I} = (V, E)$ is a process instance over $\mathcal{A}(L)$ and $\mathcal{I} \vdash t$. Let $i \in \{2, ..., n\}$, and notice that there is a path from $t[1]$ to $t[i]$, by definition of process instance. It follows that there is an edge of the form $(t[j], t[i]) \in E$. If $j < i$, then we conclude that the constraint $\pi(t)$ is satisfied by $\mathcal{G}$. Otherwise, it must be the case that $j > i$. However, this is impossible as t is a topological sort of $\mathcal{I}$. Hence, $\mathcal{G} \models \pi(t)$, for each trace $t \in L$. Thus, $\mathcal{G}$ is also a model for $\pi(L)$, and hence $\mathcal{G} \models \pi(L) \cup \Pi$.

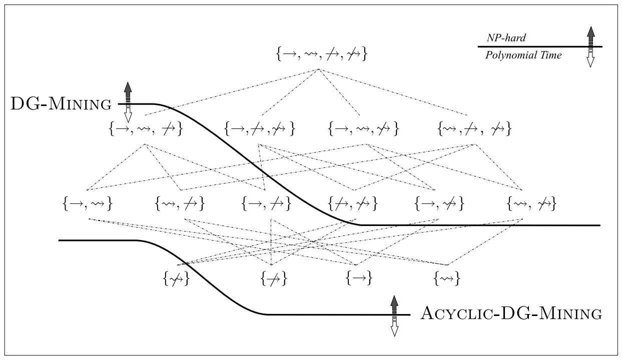
NP-hard
Polynomial Time
$\{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$
DG-MINING
$\{\rightarrow, \rightsquigarrow, \not\rightarrow\}$
$\{\rightarrow, \not\rightarrow, \not\rightsquigarrow\}$
$\{\rightarrow, \rightsquigarrow, \not\rightsquigarrow\}$
$\{\rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$
$\{\rightarrow, \rightsquigarrow\}$
$\{\rightsquigarrow, \not\rightarrow\}$
$\{\rightarrow, \not\rightarrow\}$
$\{\not\rightarrow, \not\rightsquigarrow\}$
$\{\rightarrow, \not\rightsquigarrow\}$
$\{\rightsquigarrow, \not\rightsquigarrow\}$
$\{\not\rightsquigarrow\}$
$\{\not\rightarrow\}$
$\{\rightarrow\}$
$\{\rightsquigarrow\}$
ACYCLIC-DG-MINING

Figure 2. A set $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$ above (resp., below) the frontier means that the problem is NP-hard (resp., in P) on the class $\mathcal{C}[\mathtt{S}]$.

(2)⇒(1). To complete the proof, assume that *(2)* holds. We have to show that $\mathcal{G} \vdash L$ holds. Let $\mathcal{G}$ be the graph (V, E). Let $t[1]...t[n]$ be a trace in L, and let $\mathcal{G}_t = (V_t, E_t)$ be the graph such that $V_t = \mathcal{A}(\{t\})$ and $E_t = \{(t[i], t[j]) \in E \mid i < j\}$. Of course, $\mathcal{G}_t$ is acyclic, and t is actually a topological sort of $\mathcal{G}_t$ by construction. We now claim that each activity in $t[i] \in V_t \setminus \{t[1]\}$ can be reached from $t[1]$. This is shown by induction on the index $i > 1$. In the case where $i = 2$, $(t[1], t[2])$ must belong to $\mathcal{G}_t$ in order to satisfy the constraint $\{t[1]\} \rightarrow t[2]$ in $\pi(t)$. Then, assume that activities $t[2], ..., t[i-1]$ can be reached from $t[1]$. Then, because of the constraint $\{t[1], ..., t[i-1]\} \rightarrow t[i]$ in $\pi(t)$, we again have that $t[i]$ can be reached from $t[1]$ as well. Hence, $\mathcal{G}_t$ is a process instance over $\mathcal{A}(L)$ such that $\mathcal{G}_t \vdash t$, for each trace $t \in L$. That is, $\mathcal{G} \vdash L$. $\square$

Example 8. Let Π_0 be the set of constraints in Example 5, and let $\pi(abcde)$ be the set of constraints associated with the trace $abcde$ of Example 2. Combining the two sets into the novel set $\Pi'_0 = \pi(abcde) \cup \Pi_0$, we have that $\mathcal{G}_2$ is a model of Π'_0. Thus, by Proposition 7, $\mathcal{G}_2$ is a dependency graph for $\{abcde\}$ and satisfies Π_0. $\lhd$

In the case of arbitrary logs, the mapping is established via the concept of folding as a simple extension of the above result. Indeed, in the light of Definition 3, we need to show that: $\mathcal{G}$ is a folding of $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \vdash \bar{L}$ and $\mathcal{G} \models L \Leftrightarrow \mathcal{G}$ is a folding of $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \models \pi(\bar{L})$ and $\mathcal{G} \models \Pi$. In fact, the result immediately follows by applying Proposition 7 on the log $\bar{L}$.

Corollary 9. *Let L be a log, $\mathcal{G}$ be a graph over $\mathcal{A}(L)$, and Π be a set of precedence constraints over $\mathcal{A}(L)$. Then,* (1) *$\mathcal{G}$ is a solution to* DG-MINING *on input L and Π* $\iff$ (2) $\mathcal{G} \models \Pi$ *and $\mathcal{G}$ is the folding of a graph $\bar{\mathcal{G}}$ such that $\bar{\mathcal{G}} \models \pi(\bar{L})$.*

4 Complexity Analysis

We now turn to study the complexity of the problems DG-MINING and ACYCLIC-DG-MINING. Let $\mathtt{S}$ be a subset of the set of symbols $\{\rightarrow, \rightsquigarrow, \not\rightarrow, \not\rightsquigarrow\}$. Let $\mathcal{C}[\mathtt{S}]$ denote all the possible sets of constraints that can be built over an underlying set $\mathcal{A}$ of activities such that if $\rightarrow \notin \mathtt{S}$ (resp., $\rightsquigarrow \notin \mathtt{S}$, $\not\rightarrow \notin \mathtt{S}$, $\not\rightsquigarrow \notin \mathtt{S}$), then no edge (resp., path, negated edge, negated path) constraint is in $\mathcal{C}[\mathtt{S}]$). And, let us denote by DG-MINING[S] and ACYCLIC-DG-MINING[S] the restrictions of the two problems over any set of precedence constraints Π such that $\Pi \subseteq \mathcal{C}[\mathtt{S}]$. Then, we have the following (see also Figure 2).

Theorem 10. *The following dichotomies hold:*

- *If* $\mathtt{S} \subseteq \{\not\rightsquigarrow\}$, *then* ACYCLIC-DG-MINING[S] *is feasible in* P. *Otherwise, the problem is* NP*-hard.*
- *If* $\mathtt{S} \subseteq \{\rightarrow, \rightsquigarrow, \not\rightsquigarrow\}$, *then* DG-MINING[S] *is feasible in* P. *Otherwise, the problem is* NP*-hard.*

Note that the decision versions of DG-MINING[S] and ACYCLIC-DG-MINING[S] are in NP, no matter of S. Indeed, as the size of any solution is polynomially bounded, we have just to prove that deciding whether a graph is actually a solution is feasible in polynomial time. In fact, by Proposition 7 (resp., Corollary 9), this can be reduced to verify whether $\mathcal{G}$ (resp., $\bar{\mathcal{G}}$) is a model of a certain set of precedence constraints, which is of course in P.

Due to space constraints, we next focus on hardness results only. In particular, we start with the ACYCLIC-DG-MINING problem, and we establish the following two hardness results even if $L = \emptyset$.

Theorem 11. ACYCLIC-DG-MINING[$\{\rightarrow\}$] *and* ACYCLIC-DG-MINING[$\{\rightsquigarrow\}$] *are* NP-*hard, even if* $L = \emptyset$.

Proof Sketch. Deciding whether a CNF formula $\Phi = c_1 \wedge \ldots \wedge c_m$ is satisfiable is an NP-hard problem, even if each clause c_j is assumed to be of the form $t_{j,1} \vee t_{j,2} \vee t_{j,3}$, where $t_{j,i}$ $(1 \leq i \leq 3)$ is either a variable (e.g., X_h) or a negated variable (e.g., $\neg X_h$), and where $t_{j,1}$, $t_{j,2}$, and $t_{j,3}$ are not necessarily distinct.

Based on Φ, we build the set $\mathcal{A}(\Phi)$ consisting of the clauses and the variables in Φ (viewed as activities), i.e., $\mathcal{A}(\Phi) = \{c_1, ..., c_m, \} \cup \{t_{j,1}, t_{j,2}, t_{j,3} \mid 1 \leq j \leq m\}$. Moreover, we build an associated set $\Pi(\Phi) \subseteq \mathcal{C}[\{\rightarrow\}]$ of edge constraints as follows: For each clause c_j, $\Pi(\Phi)$ contains the constraint $\{t_{j,1}, t_{j,2}, t_{j,3}\} \rightarrow c_j$; For each pair of clauses c_j and $c_{j'}$ such that $t_{j,i} = \neg t_{j',i'}$ for any two indices $1 \leq i, i' \leq 3$, $\Pi(\Phi)$ contains the constraints $\{c_j\} \rightarrow t_{j',i'}$ and $\{c_{j'}\} \rightarrow t_{j,i}$; No further constraint is in $\Pi(\Phi)$. Now, it can be checked that Φ *is satisfiable* $\Leftrightarrow$ *there is an acyclic graph* $\mathcal{G}$ *such that* $\mathcal{G} \models \Pi(\Phi)$. Since the reduction is feasible in polynomial time, it follows that ACYCLIC-DG-MINING[$\{\rightarrow\}$] is NP-hard, even if the input log contains no trace. To conclude, we just notice that the salient properties of the reduction are not altered if we replace each edge constraint in $\Pi(\Phi)$ with the analogous path constraint. Hence, ACYCLIC-DG-MINING[$\{\rightsquigarrow\}$] is NP-hard. □

For negated edge constraints, hardness can emerge only for non-empty logs, as a graph without edges is a trivial solution if $L = \emptyset$.

Theorem 12. ACYCLIC-DG-MINING[$\{\not\rightarrow\}$] *is* NP-*hard.*

Proof Sketch. Let $\Pi = \{\{b_i^1, ..., b_i^{k_i}\} \rightarrow a_i \mid i \in \{1, ..., m\}\} \subseteq \mathcal{C}[\{\rightarrow\}]$ be a set of edge constraints over a set $\mathcal{A}$ of activities. For each constraint $\{b_i^1, ..., b_i^{k_i}\} \rightarrow a_i$, let $c_i \notin \mathcal{A}$ be a fresh activity associated with it. Based on Π, we build a log $L(\Pi)$ with traces $t_1, ..., t_m$ such that $t_i = a_\perp c_i b_i^1, ..., b_i^{k_i} a_i$, for each $i \in \{1, ..., m\}$. Moreover, consider the set Π' of negated edge constraints including $\{a_\perp\} \not\rightarrow a$, for each activity $a \notin \{c_1, ..., c_m\}$, and $\{c_i\} \not\rightarrow a$, for each $a \notin \{b_i^1, ..., b_i^{k_i}\}$ and each $i \in \{1, ..., m\}$. Then, it can be shown that *there is an acyclic graph* $\mathcal{G}$ *such that* $\mathcal{G} \models \Pi \Leftrightarrow$ *there is an acyclic graph* $\mathcal{G}'$ *such that* $\mathcal{G}' \vdash L(\Pi)$ *and* $\mathcal{G}' \models \Pi'$. Hence, the result follows by the NP-hardness of ACYCLIC-DG-MINING[$\{\rightarrow\}$], which has been pointed out in Theorem 11. □

Finally, we turn to the DG-MINING problem, where negated path constraints can be used to enforce acyclicity.

Theorem 13. DG-MINING[$\{\rightarrow, \not\rightsquigarrow\}$], DG-MINING[$\{\rightsquigarrow, \not\rightsquigarrow\}$], *and* DG-MINING[$\{\not\rightarrow, \not\rightsquigarrow\}$] *are* NP-*hard.*

Proof Sketch. Let Π be in $\mathcal{C}[\{\rightarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow\}]$, $\mathcal{C}[\{\not\rightarrow\}]$), and consider the problem of deciding whether there is an acyclic graph $\mathcal{G}$ such that $\mathcal{G} \models \Pi$. Based on Π, we build the set Π' of constraints including all the constraints in Π, plus the novel constraint $\{a\} \not\rightsquigarrow a$, for each activity a. Of course, Π' belongs to $\mathcal{C}[\{\rightarrow, \not\rightsquigarrow\}]$ (resp., $\mathcal{C}[\{\rightsquigarrow, \not\rightsquigarrow\}]$, $\mathcal{C}[\{\not\rightarrow, \not\rightsquigarrow\}]$). Moreover, the role of the fresh constraints is just to enforce the acyclicity of the desired graph. Indeed, $\mathcal{G} \models \Pi'$ if, and only if, $\mathcal{G} \models \Pi$ and $\mathcal{G}$ is acyclic. The result then follows from Theorem 11 and Theorem 12. □

5 Implementation and Experimental Results

The complexity analysis we have conducted evidenced that polynomial time algorithms are unlikely to exist for the problem of computing models of sets of precedence constraints. This bad news calls for sophisticated solution approaches that perform well in practice.

Our solution approach is to encode precedence constraints in terms of "standard" *constraints satisfaction problems*. The encoding comprises three parts—details are in [7]:

Basic Encoding: Edges and paths are associated with the variables. That is, there are Boolean variables, i.e., with domain $\{0, 1\}$, of the form $edge_{X,Y}$ and $path_{X,Y}$, for each pair of activities X and Y, and denoting the existence of the corresponding edge and path. Then, precedence constraints are formulated as standard constraints over such variables. For instance, the two precedence constraints associated with the trace abc can be encoded as $edge_{a,b} \geq 1$ and $edge_{a,c} + edge_{b,c} \geq 1$. As an other example, to look for a graph over $\{a, b, c\}$ with no path from c to a, we can use the constraints: $path_{c,a} = 0$, $path_{c,a} \geq edge_{c,a}$, $path_{c,a} \geq edge_{c,b} + edge_{b,a} - 1$.

Built-in Constraints: A number of built-in constraints are considered, which reflect some standards in process definition. For instance, for each activity X not occurring at the end of some trace, it is required that there is at least an outgoing edge.

Optimization: Since several dependency graphs might satisfy a given set of precedence constraints, we support the definition of an objective function over the space of all candidate dependency graphs, as to compute the best one over them. Each edge is associated with a weight as follows. For each pair of activities a, a', first a "causality" score $\sigma(a, a')$ is computed according to the approach in [1]. Roughly, $\sigma(a, a')$ measures the number of traces where a precedes a', with a scaling factor being associated with each trace t, which exponentially decreases at the growing of the distance from a to a' within t. If $\sigma(a, a')$ happens to be above a given threshold, then the weight of the edge from a to a' is fixed to $\sigma(a, a')$; otherwise, the weight is fixed to a negative value penalizing those graphs where such an edge occurs. The goal is to compute the dependency graph with maximum overall weight.

By exploiting such encodings, the whole approach has been implemented in a system prototype reusing existing constraint programming platforms for computing models for sets of precedence constraints. In fact, these platforms have been developed to solve NP-hard problems declaratively specified, and embody sophisticated solution algorithms allowing them to scale over large datasets, as their recent application in data mining contexts have demonstrated (e.g., [9]). Our implementation leverages the *Gecode* platform.

General test setting. In order to test the prototype on a meaningful application scenario, we considered the product recall process in [12], featuring some major actions that must be undertaken in response to a recall incident (triggered by, e.g., consumer/supplier notifications or quality tests). Investigations on the reported problem and suitable risk analyses must be performed (macroactivity `PROLOGUE` —see [12], for details on specific activities), in order to decide if the product must be recalled or not. The dependency graph for the activities occurring in the former case is shown in Figure 3.

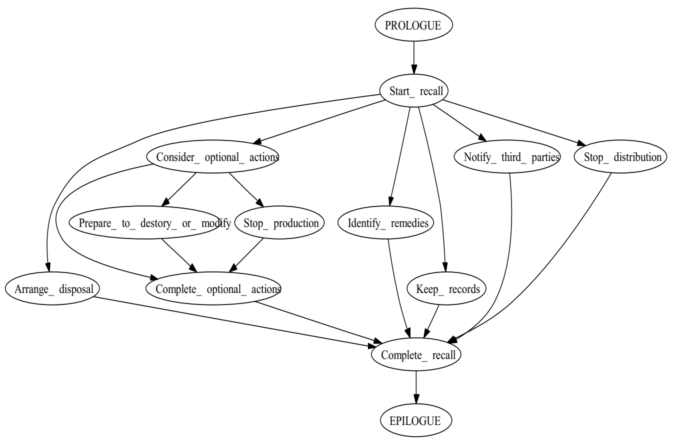

Figure 3. Dependency graph for the product recall process.

$trace\%$	[1]	[14]	[11]	[15]	[6]	$Here$
10	0.657	0.731	0.281	0.848	0.667	**0.971**
20	0.830	0.869	0.432	0.924	0.736	**0.995**
30	0.893	0.924	0.391	0.914	0.720	**1.000**
40	0.931	0.943	0.348	0.914	0.730	**1.000**
50	0.965	0.968	0.354	0.904	0.727	**1.000**
60	0.979	0.968	0.417	0.903	0.774	**1.000**
70	0.984	0.990	0.556	0.882	0.763	**1.000**
80	0.984	0.992	0.500	0.893	0.779	**1.000**
90	**1.000**	**1.000**	0.510	0.882	0.779	**1.000**
100	**1.000**	**1.000**	0.605	0.882	0.782	**1.000**
Avg	0.911	0.929	0.439	0.897	0.752	**0.997**

Figure 4. F-measure scores on different log samples, both in absence ([1], [14], [11]) and in presence ([15],[6],$Here$) of background knowledge.

In order to valuate results' quality, the set D_{out} of dependencies discovered is contrasted to the set D_{in} of real dependencies, as in the a-priori known process model, through classical *F-measure* metrics, defined as $\frac{2 \times P \times R}{P+R}$, where *P*(*recision*) is the fraction of dependencies in the mined model that appear in the real one and *R*(*ecall*) is the fraction of real dependencies occurring in the mined model, i.e., $P = |D_{out} \cap D_{in}|/|D_{out}|$ and $R = |D_{out} \cap D_{in}|/|D_{in}|$.

Results of different empirical analyses, conducted with this application scenario, are discussed next in separate subsections.

5.1 Tests with variable amounts of log data

Given the interest in analyzing situations where log completeness does not hold, we first built, as an ideal input for process mining algorithms, a *fully complete* log L for the process above, where each possible trace is registered once. Experiments were conducted over logs extracted from L by randomly picking $x\%$ of its traces, for $x \in \{10, ..., 90, 100\}$. In particular, 10 different logs were sampled for each x, and tests were performed on them all. Moreover, we assume that the analyst only knows, a priori, that activity `Complete_optional_actions` is parallel with all the activities `Identify_remedies`, `Keep_records`, `Stop_distribution`, `Notify_third_parties`, `Arrange_disposal`.

In the analysis, we compared the performances of our approach with some classical process discovery methods [1, 14, 11], and with two recent ones [15, 6] founding on a constraint-based or declarative specification of the discovery problem, capable of incorporating a-priori knowledge on activity dependencies (see Section 6). When testing these competitor methods, we took advantage of their respective implementations available in the *ProM* framework [16].

Figure 4 summarizes the results found with different amounts of log data, possibly accompanied by additional background constraints (see the rightmost two columns). Precisely, for each percentage value $x\%$, it is reported the average *F-measure* over all the 10 samples with $x\%$ of the traces[5]. In general, all methods get poor achievements on small samples, and tend to improve when using more input traces. This effect is clearer with the classical methods [1, 14]. Surprisingly, low scores are obtained by the heuristics-based global-search method of [11] and by the declarative approaches [15, 6], which do not seem to really take advantage of background knowledge. The converse happens with our approach, which achieves remarkable outcomes in presence of a-priori knowledge, even on very small samples. This is particularly interesting, since this knowledge does not imply causality links (i.e., positive edge/path constraints), but it just concerns activity independence (i.e., negated path constraints).

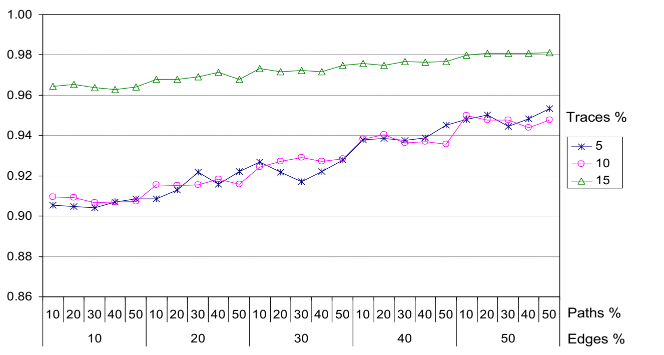

Figure 5. F-measure on different log samples, when using various amounts of (a-priori given) edge constraints and path constraints.

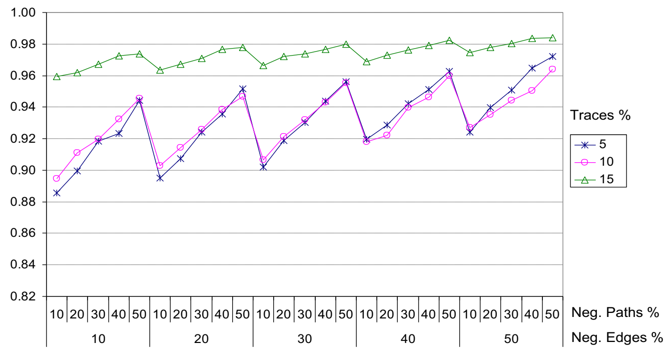

Figure 6. F-measure on different log samples, when using various amounts of (a-priori) negative edge constraints and negative path constraints.

5.2 Varying the amount of a-priori knowledge

The impact of a-priori knowledge has been further studied, by conducting tests with varying amounts of precedence constraints, in addition to those derived from the log. As a way to stress the approach on incomplete data, we considered small portions of the original log, gathering 5% to 15% of the traces. Background knowledge on activity relationships was drawn directly from the a-priori known model of the process, in the form of the four binary relations, encoding singleton-body precedence constraints: (i) *edges* and (ii) *paths* (i.e., pairs of activities, where the second depends on the first either directly or indirectly, resp.), and their associated complementary relations of (iii) *negative edges* and (iv) *negative paths*.

Figures 5 and 6 depict average F-measure scores for different percentages of both log traces and of the basic a-priori constraints de-

[5] In each test, we have selected the best performing model among those found with different parameters' settings of the method in [15], and the highest-fitness model found with the default configuration of the method in [11].

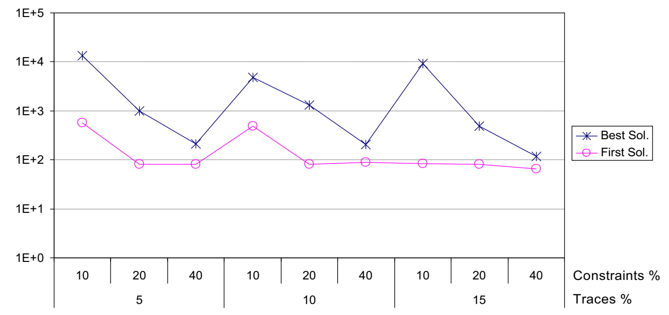

Figure 7. Time (seconds) spent to compute a solution (*First Sol.*) and an optimal one (*Best Sol.*).

scribed above. Note that, in the underlying tests, we drew 10 sub-logs for each traces' percentage and performed 10 trials on each of them, for each combination of all constraints' percentages. In general, increasing the quantity of whichever kind of a-priori constraints leads to higher accuracy scores—apart slight fluctuations for the case of 5% log samples with variable amounts of positive constraints (see Figure 5). In particular, a 15% random sample suffices to achieve nearly optimal recall and precision. Moreover, it is worth noting that the degree of improvement appears more marked with smaller log samples (hardly capturing all actual behaviors), and when exploiting negative constraints (see Figure 6).

5.3 Running times

In the process mining setting, running times for solution approaches do not usually represent a major issue (as algorithms are applied off-line, i.e., during the (re-)design phase), so that the focus is mainly on improving the performances of the approaches in terms of the quality of the results they produce. For the sake of completeness, we nonetheless believe here that it is of interest to provide the reader with a general idea of the computation times needed by our method, in particular by evidencing that, despite the theoretical intractability, quite good scalings can be obtained: Figures 7 reports the times for computing either an optimal dependency graph, or just the first one (hence, not necessarily optimal) discovered in the search space, with respect to the amount of constraints and of traces taken as input. Times are in seconds, and experiments have been conducted on a dedicated machine, equipped with an Intel dual-core processor, 2GB (DDR2 1033 MHz) of RAM, and running Windows XP Professional. Note that times reduce considerably when augmenting the quantity of background constraints, no matter of the log size—each time measure was still computed by averaging the results of 10 tests performed with different random samples, in order to reduce the sampling bias.

6 Discussion and Conclusions

The opportunity to exploit background knowledge in order to deal with incomplete logs was argued recently by [6], where a novel process discovery method is presented. After extracting temporal constraints, capturing dependence and parallelism relations between activities, negative events are generated artificially for each prefix of any log trace (each event indicates which activities are not allowed to appear next in the trace). Using both log traces and artificial negative events as input, a logic program is induced with algorithm TILDE, which is eventually converted into a Petri net. Importantly, domain experts can directly provide an a-priori set of temporal constraints, possibly stating that (i) two activities are parallel (resp., not parallel), and (ii) that one precedes/succeeds (resp., does not precede/succeed).

A different kind of "declarative" method has been proposed by [15], where a Petri-net model is discovered via an integer Linear Programming (LP) approach. An initial net, having no places, is refined iteratively by adding a place at a time. Each place is chosen greedily, by solving (via an off-the-shelf LP solver) a system of linear inequalities, asking for a place with a minimal (resp., maximal) number of incoming edges (resp., outgoing edges). To curb the growth of the mined model, the search can be guided by log relations derived from the same log (as in [14]), concerning direct activity dependencies and parallelism relationships. The user can enforce fine grain constraints, by manually modifying these relations.

In this paper, we have proposed a *constraint-based* discovery framework, where a-priori knowledge is encoded via *precedence constraints*, and the search of dependencies is stated as a constraints satisfaction (optimization) problem. Compared with both the proposals above, our setting is more general, in that it allows for specifying a broader range of constraints, including path constraints and constraints over activity sets. In fact, other declarative methods were proposed in [5, 3, 2], for learning a decision model discriminating compliant and non-compliant executions. This substantially differs from our perspective, where only compliant traces are available and the aim is to discover control-flow dependencies. Moreover, they have not explored the idea of exploiting a-priori knowledge.

REFERENCES

[1] A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process mining with the Heuristics Miner algorithm. Technical report, Eindhoven University of Technology, Eindhoven, 2006.

[2] E. Bellodi, F. Riguzzi, and E. Lamma. Probabilistic declarative process mining. In *Proc. of KSEM'10*, pages 292–303.

[3] F. Chesani, E. Lamma, P. Mello, M. Montali, F. Riguzzi, and S. Storari. Exploiting inductive logic programming techniques for declarative process mining. *Transactions on Petri Nets and Other Models of Concurrency*, 2: 278–295, Springer, 2009.

[4] S. de Amo and D. A. Furtado. First-order temporal pattern mining with regular expression constraints. *D.&K. Engineering*, 62:401–420, 2007.

[5] H. M. Ferreira and D. R. Ferreira. An integrated life cycle for workflow management based on learning and planning. *Int. J. Cooperative Inf. Syst.*, 15(4):485–505, 2006.

[6] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens. Robust process discovery with artificial negative events. *Journal of Machine Learning Research*, 10:1305–1340, 2009.

[7] G. Greco, A. Guzzo, and L. Pontieri. Process Discovery via Precedence Constraint. Full-version. Available at *https://www.mat.unical.it/~ggreco/pdvpc.pdf*.

[8] G. Greco, A. Guzzo, L. Pontieri, and D. Saccà. Discovering expressive process models by clustering log traces. *IEEE Trans. on Knowledge and Data Engineering*, 18(8):1010–1027, 2006.

[9] T. Guns, S. Nijssen, and L. De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175:1951–1983, 2011.

[10] H. Mannila, C. Meek. Global partial orders from sequential data. In *Proc. of KDD'00*, pages 161-168.

[11] A. K. Medeiros, A. J. Weijters, and W. M. Aalst. Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, 14:245–304, 2007.

[12] M.T. Wynn, C. Ouyang, A.H.M. ter Hofstede, and C.J. Fidge. Workflow support for product recall coordination. BPMcenter.org, 2009.

[13] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *D.&K. Engineering*, 47(2): 237–267, 2003.

[14] W. M. P. van der Aalst, A. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[15] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. *Fundamenta Informaticae*, 94:387–412, 2009.

[16] B. van Dongen, A. de Medeiros, H. Verbeek, A. Weijters, and W. van der Aalst. The ProM framework: A new era in process mining tool support. In *Proc. of ICATPN'05*, pages 1105–1116.