Process Discovery Using In-database Minimum Self Distance Abstractions

Alifah Syamsiyah
Eindhoven University of Technology
a.syamsiyah@tue.nl

Sander J.J. Leemans
Queensland University of Technology
s.leemans@qut.edu.au

ABSTRACT

Process executions generate event data that are typically stored in legacy information systems, such as databases. However, process discovery, which requires such event data, is performed in main memory. To bridge this gap, existing techniques must transform and extract event data, which can be expensive steps. This issue has been addressed by processing the event data directly in their origin. However, existing methods rely only on the simplest event data abstraction: the Directly Follows (DF) abstraction. This paper improves upon these existing works by considering another abstraction, the Minimum Self Distance (MSD) abstraction, which enables discovery of a larger class of models than the DF alone. That is, we propose *IMw*, a process discovery technique without logs and uses both the MSD and DF abstractions. Furthermore, this work proposes an approach to compute the MSD abstraction in-database, thus avoiding the need for transforming and moving event data. We evaluate *IMw* with real-life logs, and the experimental results show that *IMw* with in-database abstraction is faster than the traditional approach, aware of dynamic updates on event data, and able to discover models with pareto-optimal results, compared to existing techniques.

CCS CONCEPTS

• **Applied computing → Business process management**;

KEYWORDS

Process discovery, in-database abstraction, minimum self distance

ACM Reference Format:
Alifah Syamsiyah and Sander J.J. Leemans. 2020. Process Discovery Using In-database Minimum Self Distance Abstractions. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373846

1 INTRODUCTION

The omnipresence of event data allows organizations to diagnose problems based on facts rather than fiction. *Process mining* is a family of methods to exploit such event data in a meaningful way, for example to provide insights, identify bottlenecks, check compliance, suggest improvements, etc. [19]. *Process discovery*, which is an important part of process mining, aims to automatically generate process models based on event data. The generated process models must be as accurate as possible and reflect the real process underlying the event data.

Traditional process discovery techniques require logs to be loaded into memory before discovery can be performed. Furthermore, several cloud-based tools, such as Celonis, require event data to be converted from customer source systems into a standard format and to be copied to a dedicated process mining environment [1]. In this standardized environment, larger datasets can be handled faster than using traditional techniques. A characteristic that is common to traditional techniques and certain cloud-based tools entails that both require event data to be extracted, converted, and loaded again into their respective systems, environments and data formats. Ironically, these phases can take up to 80% of the project duration [8].

To overcome this issue of event data preprocessing, some efforts [7, 16, 18] introduced an approach to process data directly in their origin (e.g. databases). The idea is to split process discovery into an abstraction phase and a mining phase. The abstraction phase produces event-based abstractions in the event data origin. This abstraction process is performed directly on databases without the need to transform event data into a special format or move it. Afterwards, just like traditional process discovery techniques, these abstractions are the inputs for the mining phase, performed on a standard computer. Such abstractions are more compact than the event data they are derived from and therefore require much less main memory than the events they were derived from. Furthermore, abstractions are often of a standard form which is compatible with many discovery algorithms (unlike cloud-based tools). However, existing works typically only rely on the simplest abstraction, called the Directly Follows (DF) relation.

As formally shown in [11], the class of models that can be discovered from DF abstractions is limited. Consider the following example: in the Netherlands, normal pregnancies are handled by midwives. A midwife may perform several ultrasounds to check the baby's health. In addition, a general practitioner (GP) takes a role in assuring that a pregnant woman does not suffer from maternity diseases such as anemia, gestational diabetes, etc. A GP may ask for a blood test if necessary. In a healthy pregnancy, a check-up by GP is only done once and happens in parallel with check-ups by the midwife. In case of any abnormality, examination by GP and blood test may be repeated several times. Based on the above description, we intend to have a process model as shown in Figure 1a. However, process discovery techniques which only rely on DF abstractions might return a completely different process model as displayed in

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373846

Figure 1b. In the latter model, the discovery algorithm obliges that check-ups by midwife must be performed in parallel with check-ups by GP. Moreover, an ultrasound must be accompanied by a blood test as well.

This problem occurs because the DF abstraction of both process models is equivalent (Figure 1c). Hence, the DF abstraction does not contain enough information to distinguish the two process models, even though their semantics are clearly different. In [11], a new abstraction was proposed, the *Minimum Self Distance* (MSD), which expresses whether an activity can appear between two executions of another activity with a minimum number of events in between (but no algorithm that uses this abstraction was proposed). For the above example, the MSD abstractions of both process models are different, as denoted in figures 1d and 1e, thus the correct process model can be discovered.

In this paper, we use the MSD abstraction to extend the class of processes that can be discovered, we introduce a new discovery algorithm that uses the MSD abstraction, and we show how the MSD abstraction can be constructed from a database. To this end, we propose a solution which consists of two parts: first, we propose a technique to construct the MSD abstraction of event data, while the data remain in their database. Second, we propose a new process discovery technique, *IMw* (*Inductive Miner without logs*), which utilizes the DF and MSD abstractions and returns models with formal guarantees.

Compared to existing process discovery techniques, our work improves applicability by providing an end-to-end approach from events stored in databases to visualized process models (thus avoiding event data extraction and transformation). Our approach also enables process discovery on large and complex data sets, since it only imports the DF and MSD abstractions (and not the logs) into memory. In Section 6 we show that the in-database abstraction accelerates the computation of process discovery and requires less main memory than a traditional approach with in-memory abstraction. Finally, in Section 6 we show that even though *IMw* uses less information from event logs, it can discover models that are pareto-optimal with respect to models discovered by other techniques that use the full event log.

This paper is structured as follows: in Section 2 we discuss related work. Terminology is introduced in Section 3. In Section 4, we explain the in-database abstraction for process discovery. Then, in Section 5, we present the *IMw* framework which uses the MSD abstraction. Finally, in Section 6 we show the experimental results and we conclude this paper in Section 7.

2 RELATED WORK

Process Discovery. The Inductive Miner framework is a process discovery framework that recursively identifies the "most important" behaviour and splits the event log, until a base case is encountered. Based on this framework, many variants (i.e. discovery techniques) have been proposed to handle various types of event logs and discovery challenges, such as [12, 14]. The framework is robust and provides several formal guarantees, such as soundness (i.e. lack of deadlocks and other anomalies) of the returned models. For handling large event logs, the Inductive Miner - directly follows (IMd) framework applies the same steps as the IM framework, but uses the DF abstraction as its recursion artefact, rather than an event log [13]. In this paper, *IMw* extends IMd by not only using the DF but also the MSD abstraction to distinguish more types of behaviour, thus yielding a similar framework.

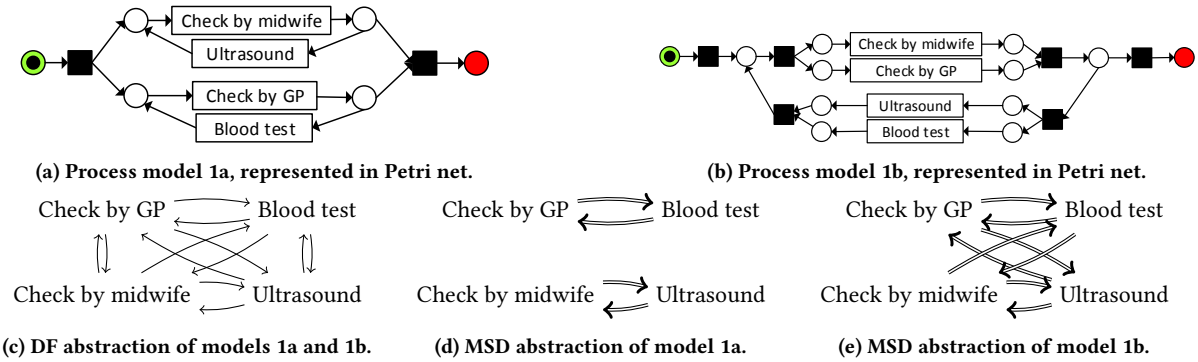
Other process discovery techniques have been proposed, but these are either not applicable directly to abstractions (e.g. Evolutionary Tree Miner [3], Split Miner [2], Indulpet Miner [14]) or do not provide basic guarantees such as soundness (e.g. Split Miner [2], Integer Linear Programming Miner [22], Heuristics Miner [28]).

IMd, due to its independence from event logs, has also been adapted to the context of process discovery on event streams [24, 25]. These techniques assume event data to be dynamic sequences of events and use these sequences to approximate a DF abstraction, after which IMd discovers a process model from the DF abstraction. Furthermore, streaming techniques assume that an infinite amount of data is available and needs to be considered in finite memory, thus it is not possible to store all data and generate a model based on the full event log, as it is assumed that this full log cannot be stored. The two abstractions (DF and MSD) from event logs used in this paper might be applicable to streaming settings as well, however this is challenging as the MSD abstraction computation requires the creation and storage of intermediate structures that need a non-constant amount of memory.

Process Mining in Databases. Process mining in databases has been investigated in several studies. A study in [27] introduced a tool called XESame. To access the data in XESame, one needs to materialize the data by selecting and matching it with XES [9] elements. A more advanced technique using ontologies was proposed in [4, 5]. In this work, data are accessed using query unfolding and rewriting techniques, called ontology-based data access. Furthermore, a work proposed in [6] utilized a database redo log as a means to extract events. All of these work, however, only focus on event data extraction from databases, which we want to avoid.

Building on the idea of storing event data in databases, RXES [23] was introduced as a fixed database schema emulating the XES standard. However, its application to a real process mining algorithm was not investigated. To improve RXES, DB-XES was introduced in [17]. Based on the DB-XES schema, a DF abstraction is computed using a database trigger, that is, automatically when a new event is stored. This update mechanism allows users to dynamically add event data without recomputing the whole abstraction. However, as the schema is fixed, an existing system needs to be converted to the DB-XES format while our approach is flexible to any database schemes.

The DF auto-update in DB-XES is adapted in [18] to establish Incremental Inductive Miner, which utilizes IMd and adds the ability to cope with recurrent process discovery. Although the result is promising, this work only covers the simplest family of process trees [11]. Moreover, the standard SQL-based query used to compute the DF abstraction is very costly due to many self joins. This has been improved by [7, 16] which propose a database operator to specifically compute the DF abstraction. In this paper, we improve the work in [7, 16, 18] by computing the MSD abstraction in-database and proposing a new discovery framework (*IMw*) that uses more information: that is, the MSD abstraction.



(a) Process model 1a, represented in Petri net. **(b) Process model 1b, represented in Petri net.**

(c) DF abstraction of models 1a and 1b. **(d) MSD abstraction of model 1a.** **(e) MSD abstraction of model 1b.**

Figure 1: An example where two process models have the same DF abstraction but different MSD abstractions.

3 PRELIMINARIES

Event Logs. Let $\mathcal{U}_{\mathcal{E}}$ be the universe of events and $\mathcal{U}_{\mathcal{A}}$ be the universe of activities. Let $\mathcal{E} \subseteq \mathcal{U}_{\mathcal{E}}$ be a collection of events. For any event $e \in \mathcal{E}$, $\#_{case}(e)$ is the case identifier of e, $\#_{act}(e) \in \mathcal{U}_{\mathcal{A}}$ is the activity name of e, and $\#_{time}(e)$ is the timestamp when e is executed. A trace $\sigma \in \mathcal{E}^*$ is a sequence of events such that each event occurs only in a single trace, i.e. $e_1 = \sigma(i) \wedge e_2 = \sigma(j) \implies e_1 = e_2 \equiv i = j \wedge \neg\exists_{\sigma', \sigma' \neq \sigma}\, e_1 \in \sigma' \vee e_2 \in \sigma'$. Furthermore, each event in a trace refers to the same case identifier, i.e. $e_1 = \sigma(i) \wedge e_2 = \sigma(j) \implies \#_{case}(e_1) = \#_{case}(e_2)$ and we assume all events are totally ordered, i.e. $e_1 = \sigma(i) \wedge e_2 = \sigma(j) \implies i < j \equiv \#_{time}(e_1) < \#_{time}(e_2)$. An event log $\mathcal{L} \subseteq \mathcal{E}^*$ is a set of traces.

For instance, $\mathcal{L}_E = [\langle a, b, c, a\rangle, \langle a, d, a\rangle]$ is an event log with two traces and seven events associated to the activity labels a, b, c and d. Note that here we use a simplification of event logs where activity labels are shown instead of events.

Directly Follows Abstraction. Let $\mathcal{L} \subseteq \mathcal{E}^*$ be an event log over $\mathcal{E} \subseteq \mathcal{U}_{\mathcal{E}}$ and let $\top, \bot \notin \mathcal{U}_{\mathcal{A}}$. The directly follows relation $\succ_{\mathcal{L}}$: $(\mathcal{U}_{\mathcal{A}} \cup \{\top\}) \times (\mathcal{U}_{\mathcal{A}} \cup \{\bot\}) \to \mathbb{N}$ counts the number of times activity a is directly followed by activity b, and when an activity a is a start or end activity:

$$\succ_{\mathcal{L}}(a, b) = \sum_{\sigma \in \mathcal{L}} \sum_{i=1}^{|\sigma|-1} \begin{cases} 1 & \text{if } \#_{act}(\sigma(i)) = a \ \wedge\ \#_{act}(\sigma(i+1)) = b \\ 0 & \text{otherwise} \end{cases}$$

$$\succ_{\mathcal{L}}(\top, a) = \sum_{\sigma \in \mathcal{L}} 1 \text{ if } \#_{act}(\sigma(1)) = a, 0 \text{ otherwise}$$

$$\succ_{\mathcal{L}}(a, \bot) = \sum_{\sigma \in \mathcal{L}} 1 \text{ if } \#_{act}(\sigma(|\sigma|)) = a, 0 \text{ otherwise}$$

For instance, the DF abstraction of our example log $\mathcal{L}_E$ is

$$\begin{tikzcd} d \arrow[r, shift left, "1"] & a \arrow[l, shift left, "1"] \arrow[r, shift left, "1"] \arrow[d, shift right, "2"'] & b \arrow[l, shift left, "1"] \arrow[r, shift left, "1"] & c \arrow[l, shift left, "1"] \\ & a \arrow[u, shift right, "2"'] & & \end{tikzcd}$$

Minimum Self Distance Abstraction. Let $\mathcal{L} \subseteq \mathcal{E}^*$ be an event log over $\mathcal{E} \subseteq \mathcal{U}_{\mathcal{E}}$. The minimum self distance $\text{msd}_{\mathcal{L}} : \mathcal{U}_{\mathcal{A}} \to \mathbb{N}$ of activity a is the minimum number of events in between two executions of a [11] (where $\bullet$ denotes trace concatenation and we assume $\min(\emptyset) = \infty$):

$$\text{msd}_{\mathcal{L}}(a) = \min\{|t_2| \mid t_1 \bullet \langle e_n\rangle \bullet t_2 \bullet \langle e_m\rangle \bullet t_3 \in \mathcal{L} \wedge \#_{act}(e_n) = \#_{act}(e_m) = a\}$$

Then, activity b is a *witness* of this minimum self distance of a, denoted by $\alpha_{\text{msd}}(a, b)$, if and only if it can appear in between two minimum-distant executions of a:

$$\alpha_{\text{msd}}(a, b) \equiv \exists_{\langle \ldots, e_n, \ldots_1, e_m, \ldots\rangle \in \mathcal{L}}\, e_l \in \ldots_1 \wedge |\ldots_1| = \text{msd}_{\mathcal{L}}(a) \wedge \#_{act}(e_n) = \#_{act}(e_m) = a \wedge \#_{act}(e_l) = b$$

We visualise the MSD relation as an *MSD graph*, whose nodes are the activities and two nodes a and b are connected with a directed edge if $\alpha_{\text{msd}}(a, b)$ holds. For instance, the MSD graph of our example log $\mathcal{L}_E$ is $d \Longleftarrow a \qquad b \qquad c$, where each double arrow from nodes a to b indicates that $\alpha_{\text{msd}}(a, b)$ holds.

Process Trees. Process trees are a process modelling formalism, which express their behaviour in a hierarchical way, over a universe of activities $\mathcal{U}_{\mathcal{A}}$. As process trees are inherently free of deadlocks and other anomalies, the discovery technique introduced in this paper will construct process trees. We define process trees inductively: a *leaf* $a \in \mathcal{U}_{\mathcal{A}}$ is a process tree and represents the language $\{\langle a\rangle\}$. Let $M_1 \ldots M_n$ be process trees, then an *operator node* $\oplus(M_1, \ldots M_n)$ is a process tree and represents a combination of the languages of its children $M_1 \ldots M_n$, based on the process tree operator $\oplus$.

In this paper, we consider the process tree operators $\times$ (which denotes the exclusive choice between children), $\rightarrow$ (the sequential composition of children), $\wedge$ (the concurrent composition of children) and $\circlearrowleft$ (which denotes a sequential composition of a first child, followed by a repeated combination of a non-first child and the first child). For a formal definition of process trees, please refer to [11]. For instance, the language of the process tree $\times(a, \rightarrow(b, \circlearrowleft(c, d)))$ is $\{\langle a\rangle, \langle b, c\rangle, \langle b, c, d, c\rangle, \langle b, c, d, c, d, c\rangle \ldots\})$.

4 IN-DATABASE ABSTRACTION FOR PROCESS DISCOVERY

Common process discovery techniques perform the following five steps in order to produce process models: (1) extraction and conversion, (2) loading, (3) abstraction, (4) mining, and (5) visualization. Figure 2 illustrates a typical process discovery workflow based on

these steps (indicated with solid arrows). A standard process discovery task starts from a loaded event log file in main memory. Therefore, raw data stored in legacy information systems such as databases first have to be *extracted* and *converted* into a file-based format, such as CSV or XES [9] format. Then, the event log file is placed in a computer's memory during a *loading* phase. Obviously, this phase is limited by the computer's memory. Furthermore, the event log is transformed into a more compact data structure. We call this phase the *abstraction* phase. In this paper, we consider DF and MSD as two examples of abstractions. The DF abstraction represents how often an activity is followed directly by another activity, while the MSD abstraction represents which activity might appear in between two as-close-as-possible appearances of another activity. It is clear that the size of these abstractions does not depend on the number of events, but only on the number of activities. Finally, during a *mining* phase, process discovery algorithms generate a process model based on the abstractions. The discovered model is then shown to users after a *visualization* phase for further analysis.

In this paper, we consider scenarios where process discovery is performed on large event data: such data with billions of events challenges current process discovery techniques as the loading phase is limited by available memory. Therefore, we extend the ideas from [7, 16, 18] which skip the extraction and loading phases by moving the abstraction phase into databases where event data are located.

Moving the abstraction phase into databases enables us to compute DF and MSD relations directly in the databases and only load those relations into memory for discovery. Consequently, as the DF and MSD relations do not depend on the number of events, the occupied memory for process discovery is less than if the whole log is imported. In contrast, we do not move the mining and visualization phases into databases, as the mining phase is typically already quick (as will be shown in the evaluation section), thus it can be executed on the fly. Furthermore, the visualization phase must be performed in a process mining tool since its purpose is to show process discovery results to users.

Beside large sets of event data, we also consider situations where process discovery is performed on dynamic event data, that is, situations in which events are added incrementally to databases. Using current process discovery techniques, one needs to repeat all phases due to the static characteristic of file-based event logs. Once a new event is added, the process analysts have to load the data again and wait for the abstraction, mining, and visualization phases. To avoid this recomputation, we introduce an *update function* which is responsible for automatically updating the MSD abstraction according to each insertion of a new event.

Figure 2 illustrates the proposed approach with in-database abstraction (indicated with dashed arrows). In the following, we first elaborate the ideas from [7, 16, 18] about the DF abstraction phase and DF update function. Then, we introduce our idea which proposes the MSD abstraction phase and MSD update function.

4.1 DF and MSD Abstraction Phase

To build DF abstractions from existing data in databases, we exploit a native operator introduced in [7, 16]. This operator is a function f that takes a table L as input such that L contains at least case identifiers, activity labels, and timestamps in the first three columns, and $f(L)$ returns the DF abstraction of the table. The native operator is compatible with any database schema, hence the conversion of event data into a specific database schema such as DB-XES [17] or the Celonis standardized system [1] is not necessary. Furthermore, this operator boosts performance of a DF query as SQL-based queries are not designed towards process mining approaches. In [7, 16], it has been shown that, while the execution of traditional SQL queries leads to third-order polynomial time complexity, the native operator approach is linear in sorted logs and $O(e \cdot log(e))$ in unsorted logs, where e is the number of events. In this paper, we extend this native operator f to also accommodate queries for constructing MSD abstractions in databases[1].

4.2 DF Update Function

One of the often-used abstractions for process discovery is DF, used in for instance the process discovery techniques Inductive Miner [10], Alpha Miner [21], Flexible Heuristics Miner [28], and Fodina [26]. DF abstractions can be computed during a single linear pass over event logs, visiting each event exactly once. Despite its simplicity, computing the DF over enormous event datasets is a time-consuming task. Therefore, we exploit the same technique as introduced in [18] to update the DF in an automatic way. The DF update function is implemented as a database trigger which is automatically called in every insertion of new events. It has been proven in [18] that updating the DF abstraction is possible through this trigger. Intuitively, the database keeps track of the last event in a trace to avoid a recurrence in reading an event sequence. This information is stored in a table, called as *intermediate structure*, whose primary key is a trace column and each trace refers to the last event in the trace. Therefore, the complexity of the DF update function is $O(1)$ as the trigger has access to the intermediate structure and a search operation using index in a table is a constant operation.

4.3 MSD Update Function

Similar to the DF abstraction, the MSD abstraction needs an intermediate structure to avoid recomputation. The MSD intermediate structure keeps track of the last index of an activity in a trace. In the following, we first define a function γ as the MSD intermediate structure, then we show that updating the MSD abstraction is possible by utilizing γ.

Last index of an activity in a trace (γ). Let $\mathcal{L} \subseteq \mathcal{E}^*$ be an event log over $\mathcal{E} \in \mathcal{U}_{\mathcal{E}}$, $\sigma \in \mathcal{L}$ be a trace in the event log, and $a \in \mathcal{U}_{\mathcal{A}}$ be an activity. The function $\gamma : \mathcal{L} \times \mathcal{U}_{\mathcal{A}} \rightarrow \mathbb{N}$ is a function that returns the last index of an activity a belonging to a trace σ, i.e.

$$\gamma(\sigma, a) = \begin{cases} -\infty, & \text{if } \neg\exists_{i \in \{1,\ldots,|\sigma|\}} \\ & \#_{act}(\sigma(i)) = a, \\ max_{i \in \{1,\ldots,|\sigma|\}} \#_{act}(\sigma(i)) = a & \text{otherwise.} \end{cases}$$

Given a log $\mathcal{L}$, the function γ defined above is sufficient to update $\text{msd}_{\mathcal{L}}$ in constant complexity:

Corollary 4.1. *Let $\mathcal{L} \subseteq \mathcal{E}^*$ be an event log over $\mathcal{E} \subseteq \mathcal{U}_{\mathcal{E}}$ and $\sigma \in \mathcal{L}$ be a trace. Suppose a new event $e' \in \mathcal{U}_{\mathcal{E}} \backslash \mathcal{E}$ arrives such that*

[1]See https://svn.win.tue.nl/repos/prom/Packages/InDatabasePreprocessing/Trunk/.

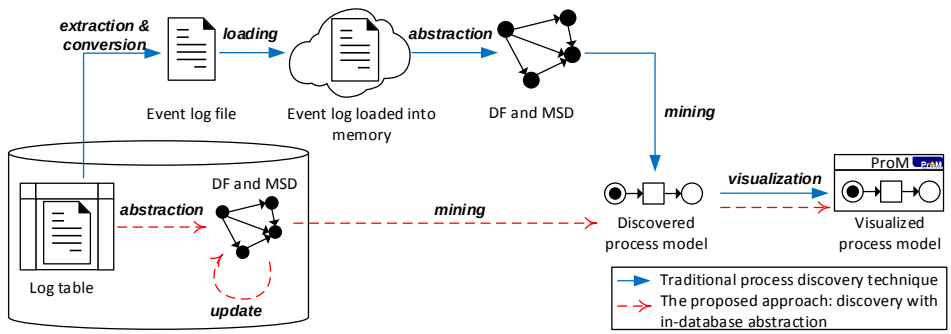


Figure 2: The traditional vs the proposed approach for process discovery.

for all $e \in \mathcal{E}$ *it holds that* $\#_{\text{time}}(e) < \#_{\text{time}}(e')$. *That is,* $\mathcal{E}' = \mathcal{E} \cup \{e'\}$, $\sigma' = \sigma \cdot \langle e' \rangle$, $\mathcal{L}' = \mathcal{L} \backslash \{\sigma\} \cup \{\sigma'\}$, *and* $\mathcal{L}' \subseteq \mathcal{E}'^*$. *Then, for all* $a \in \mathcal{U}_{\mathcal{A}}$ *it holds that:*

$$\text{msd}_{\mathcal{L}'}(a) = \begin{cases} \min(\text{msd}_{\mathcal{L}}(a),\ |\sigma'| - \gamma(\sigma, a) - 1) & \text{if } \#_{\text{act}}(e') = a \\ \text{msd}_{\mathcal{L}}(a) & \text{otherwise.} \end{cases}$$

Furthermore, γ is sufficient to update the witness activities of $\text{msd}_{\mathcal{L}}$ (α_{msd}) in $O(d)$ where d is the finite maximum value of $\text{msd}_{\mathcal{L}}$:

Theorem 4.2 (γ is sufficient to update α_{msd} in $O(d)$). *Let* $\mathcal{L} \subseteq \mathcal{E}^*$ *be an event log over* $\mathcal{E} \subseteq \mathcal{U}_{\mathcal{E}}$ *and* $\sigma \in \mathcal{L}$ *be a trace. Suppose a new event* $e' \in \mathcal{U}_{\mathcal{E}} \backslash \mathcal{E}$ *arrives such that for all* $e \in \mathcal{E}$ *it holds that* $\#_{\text{time}}(e) < \#_{\text{time}}(e')$. *That is,* $\mathcal{E}' = \mathcal{E} \cup \{e'\}$, $\sigma' = \sigma \cdot \langle e' \rangle$, $\mathcal{L}' = \mathcal{L} \backslash \{\sigma\} \cup \{\sigma'\}$, *and* $\mathcal{L}' \subseteq \mathcal{E}'^*$. *Then, for all* $a, b \in \mathcal{U}_{\mathcal{A}}$ *it holds that:*

(1) *If* $\text{msd}_{\mathcal{L}}(a) = \text{msd}_{\mathcal{L}'}(a) = \infty$ *then* $\neg \exists_{b \in \mathcal{U}_{\mathcal{A}}} \alpha'_{\text{msd}}(a, b)$.
(2) *If* $\text{msd}_{\mathcal{L}}(a) = |\sigma'| - \gamma(\sigma, a) - 1$ *then* $\forall_{b \in \mathcal{U}_{\mathcal{A}}} \alpha'_{\text{msd}}(a, b) \equiv \alpha_{\text{msd}}(a, b) \vee \exists_{i \in \{1, \ldots, \text{msd}_{\mathcal{L}'}(a)\}} \#_{\text{act}}(\sigma'(|\sigma'| - i)) = b$.
(3) *If* $\text{msd}_{\mathcal{L}}(a) > \text{msd}_{\mathcal{L}'}(a)$ *then* $\forall_{b \in \mathcal{U}_{\mathcal{A}}} \alpha'_{\text{msd}}(a, b) \equiv \exists_{i \in \{1, \ldots, \text{msd}_{\mathcal{L}'}(a)\}} \#_{\text{act}}(\sigma'(|\sigma'| - i)) = b$.

Proof. Property 1: if $\text{msd}_{\mathcal{L}'}(a) = \infty$ then there are no witnesses b.
Property 2: if $\text{msd}_{\mathcal{L}}(a) = |\sigma'| - \gamma(\sigma, a) - 1$, then trace $\sigma' = \langle \ldots, e, \ldots 1, e' \rangle$ with $\#_{act}(e) = \#_{act}(e') = a$ and $|\ldots 1| = |\sigma'| - \gamma(\sigma, a) - 1 = \text{msd}_{\mathcal{L}'}(a)$, hence $\forall_{e_l \in \ldots 1, \#_{act}(e_l) = b} \alpha'_{\text{msd}}(a, b)$ and since the $\text{msd}_{\mathcal{L}}(a)$ did not change, all previous relations still hold.
Property 3: It is trivial to see that a reduction of msd implies there is no other $\sigma'' \in \mathcal{L}$ such that $\sigma'' = \langle \ldots, e_n, \ldots 1, e_m, \ldots \rangle$ with $\#_{act}(e_n) = \#_{act}(e_m) = a$ and $|\ldots 1| = \text{msd}_{\mathcal{L}'}(a)$, hence the previous relations no longer hold.

To get a set of activities which witness a minimum self distance of activity a, we loop into the corresponding trace and take the activities in between the two occurrences of a. Therefore, the complexity is $O(d)$ where d is the finite maximum value of $\text{msd}_{\mathcal{L}}$, i.e. $d = max(\{\text{msd}_{\mathcal{L}}(a) | a \in \mathcal{U}_{\mathcal{A}} \wedge \text{msd}_{\mathcal{L}}(a) \neq \infty\})$. □

5 PROCESS DISCOVERY USING THE DF AND MSD ABSTRACTIONS

In the previous section, we have shown how the DF and MSD abstractions can be obtained from database. The Inductive Miner - directly follows (IMd) framework has been shown to handle event logs of 10^4 activities and 10^8 traces [13], however this framework not use the extra information that MSD can provide. Therefore, in this section, we introduce a new process discovery framework (Inductive Miner - without log (*IMw*)) that extends the IMd framework with MSD graphs. We first explain the *IMw* framework, then we introduce an algorithm that uses the framework, after which we show an example and discuss the guarantees that it provides.

5.1 Framework (*IMw*)

The *IMw* framework takes as input a DF and an MSD abstraction, and returns a process tree.

First, the *IMw* framework identifies a *cut* of the behaviour in the abstractions: that is, a process tree operator ($\oplus$) with a partition of the activities ($\Sigma_1 \ldots \Sigma_n$), such that the cut adheres to a particular footprint corresponding to the process tree operator $\oplus$ (see e.g. Figure 3). Intuitively, a cut describes the relation between the activities in the cut's partition. Second, the DF and MSD abstractions are split: for every set of activities in the partition of the cut, a DF and an MSD sub-abstraction are constructed. Third, the framework recurses on these sub-abstractions, which yields a process tree $T_1 \ldots T_n$ for each set in the cut's partition $\Sigma_1 \ldots \Sigma_n$. Then, the framework returns the process tree $\oplus(T_1, \ldots T_n)$. Fourth, if a base case applies then that base case is returned: for instance if only a single activity remains in the abstractions, that activity is returned as a process tree leaf. Finally, if no cut can be found and no base case applies, then a generalisation (*fallthrough*) is applied.

```
procedure IMw(D, M)
    base ← findBaseCase(D, M)
    if findBaseCase successful then
        return base
    (⊕, Σ1, ... Σn) ← findCut(D, M)
    if findCut successful then
        D1, M1, ... Dn, Mn ← split(D, M, ⊕, Σ1 ... Σn)
        return ⊕(IMw(D1, M1), ... IMw(Dn, Mn))
    return fallThrough(D, M)
```

A discovery algorithm implementing the *IMw* framework thus has to provide the functions *findBaseCase*, *findCut*, *split* and *fallThrough*.

5.2 Algorithm (*IMfw*)

Second, we introduce an algorithm that implements the *IMw* framework, called Inductive Miner - infrequent - without log (*IMfw*).

As *IMfw* extends IMfd (Inductive Miner - infrequent - directly follows) [13], we discuss the changed steps here: cut detection and abstraction splitting.

Cut Detection. To perform cut detection, *IMfw* considers both the DF and MSD abstractions in a four-stage approach. In stage 1, *IMfw* searches for particular patterns (*footprints*) in both abstractions that indicate the presence of cuts in the behaviour of the event log. That is, a footprint is sought that adheres to footprints in both abstractions. Figure 3 shows the intuition of these footprints for the DF abstraction. For the MSD abstraction, we exploit the techniques of [11]:

- For a concurrent cut $(\wedge, \Sigma_1, \ldots \Sigma_n)$, no (MSD-)edges are present between the parts:
- For a loop cut $(\circlearrowleft, \Sigma_1, \ldots \Sigma_n)$: (1) each activity has an outgoing edge, (2) all redo activities that have a connection to a body activity, have connections to the same body activities, (3) all body activities that have a connection to a redo activity, have connections to the same redo activities, (4) no two activities from different redo children have a connection.

For instance, if the DF abstraction contains unconnected groups of activities, these groups form the partition of an exclusive-choice cut, as the MSD footprints do not pose further restrictions.

If no cut is found, then *IMfw* moves to stage 2: the cut detection procedure of IMd is applied unchanged. That is, a footprint is sought in the DF abstraction only, as per Figure 3. If still no cut has been found, in the third stage, the DF abstraction is filtered for infrequent edges [13] and the first stage is applied again. Similarly, if no cut has been found, the second stage is applied again.

Abstraction Splitting. After cut detection, *IMfw* splits the DF and MSD abstractions. The DF abstraction is split according to the partition of the cut, and depending on the operator, start activities, end activities and empty traces are added as appropriate.

The MSD abstraction is split according to the partition of the cut: for each set of activities, a sub-abstraction is constructed containing only these activities. The edges are filtered as follows: the edge is kept in the sub-abstraction if and only if its both endpoints are in the set of activities; any other edge is removed.

5.3 Example

To illustrate the *IMw* framework and the *IMfw* algorithm, we walk through its steps using an example event log: $[\langle a, b, a, c\rangle, \langle c, a\rangle, \langle a, b, c, a\rangle, \langle a, c, b, a, b, a\rangle]$. The DF and MSD graphs of this log are shown in figures 4a and 4b. In the DF graph, several cuts apply: $(\circlearrowleft, \{a, c\}, \{b\})$, $(\wedge, \{a, b\}, \{c\})$ and $(\wedge, \{a\}, \{b, c\})$. Hence, the IMd framework would not have enough information to distinguish these three cuts and has to choose arbitrarily. The *IMw* framework uses the MSD graph, which allows the *IMfw* algorithm to use more information: there is an edge between a and b, which excludes $(\wedge, \{a\}, \{b, c\})$. Furthermore, c does not contain an outgoing edge, which excludes $(\circlearrowleft, \{a, c\}, \{b\})$, which leaves *IMfw* to select the cut $(\wedge, \{a, b\}, \{c\})$. Consequently, *IMfw* splits the graphs in the subgraphs shown in figures 4c to 4f.

Next, *IMfw* recurses on the graphs shown in figures 4c and 4d. In these graphs, the cut $(\circlearrowleft, \{a\}, \{b\})$ is identified and the graph is split into the graphs shown in figures 4g to 4j. After this step, *IMfw* recurses on the graphs of figures 4e and 4f, returning a base case c. After recursion on the remaining graphs, the process tree $\wedge(\circlearrowleft(a, b), c)$ is returned.

5.4 Guarantees

First, by its use of process trees, the *IMw* framework guarantees to return a *sound* model, that is, the models that are discovered are bounded, free of deadlocks and free of livelocks.

Second, discovery techniques might guarantee *rediscoverability*: let S be a business process being executed in practice and let L be an event log derived from S. Then, a process discovery technique that discovers a model that is language equivalent to S provides rediscoverability. As S is unknown, rediscoverability is a formal property that is typically proven using asssumptions on S and L.

As shown in our example, as the *IMfw* algorithm uses more information from the event log (the MSD abstraction), it can handle process trees with direct nestings of $\wedge$ and $\circlearrowleft$ operators, whereas IMfd does not provide guarantees on such models. We conjecture that *IMfw* provides rediscoverability on a larger class of processes (S) than IMfd, that is, on arbitrarily nested $\wedge$ and $\circlearrowleft$ operators. For a detailed description of this class and a skeleton proof of its rediscoverability, please refer to [11], however, a full proof is outside the scope of this paper.

6 EVALUATION

We performed an evaluation in order to investigate the following three experimental goals: (1) to compare the time-performance of *IMfw* with traditional settings (*Trad*) versus *IMfw* with in-database abstraction (*DB*), (2) to explore the time-performance of the DF and MSD update function, and (3) to evaluate the quality of models discovered by *IMfw* compared to other process discovery techniques.

6.1 Traditional *IMfw* vs *IMfw* with In-database Abstraction

In the first experiment, we compare the run time of the proposed technique (*DB*) with the traditional discovery technique (*Trad*)– both techniques utilize *IMfw* as their discovery algorithm, however the former performs an abstraction phase in databases while the latter requires event log files to be loaded into memory before the abstraction. Note that *IMfw* is our proposed algorithm that implements our proposed framework (*IMw*). To this end, we applied both techniques to 21 real-life logs[2], measured the run time of the different stages of discovery, and averaged the time over ten runs.

Figure 5 shows the results in absolute time while Figure 6 shows the results in relative time: for each log, the run time of *Trad* has been taken as 1, while scaling the time of *DB*. It is clear that *DB* outperforms *Trad* for all logs. Moreover, *Trad* required more memory to load the two Xerox logs. While 2GB of main memory is sufficient for the other logs, the Xerox(jun) log required 4GB and the Xerox(nov) log required 14GB of memory. These logs were obtained from processes occurred at the Xerox company in June and November 2015 and contain 1.2M and 15M events. Obviously, these logs do not fit into 2GB of memory. However, the Xerox(jun) log contains only 54 activities which corresponds to 282 edges of DF and MSD, while the Xerox(nov) log contains 49 activities and

[2] All logs are publicly available from https://data.4tu.nl/repository/collection:event_logs_real, except the two Xerox logs.

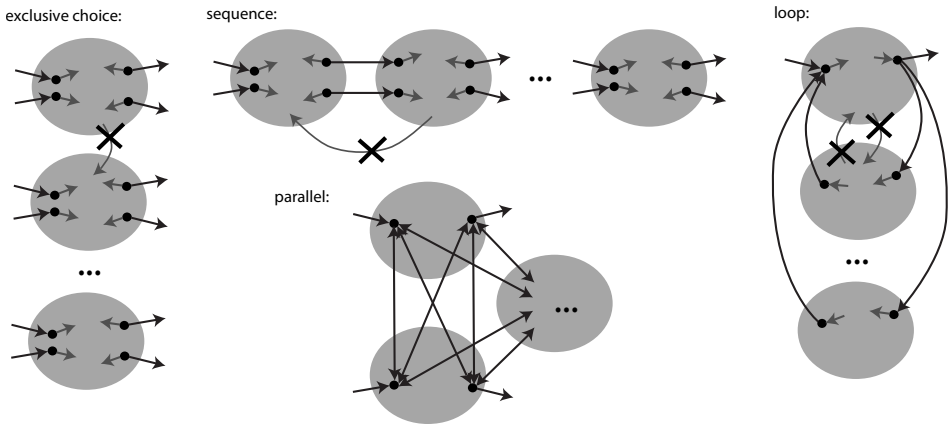


Figure 3: Cut footprints, used for the identification of cuts [source: [13]].

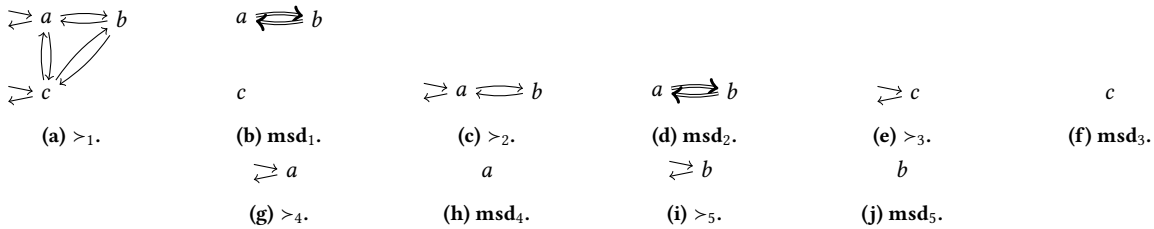


Figure 4: DF and MSD graphs to illustrate the *IMw* framework and the *IMfw* algorithm.

corresponds to 287 edges, which fits in 2GB of memory. In fact, all logs were successfully processed by *DB* using only 2GB of main memory, as *DB* only imported the DF and MSD abstractions of the logs into memory. This illustrates that the growth of event data does not immediately influence the size of DF and MSD abstractions, because of which exploring these abstractions independently from event data can save lots of time and resources.

In the following we elaborate on the results shown in Figure 6 by considering the time spent in the several phases of *DB* and *Trad*. As mentioned in Section 4, *Trad* has five discovery phases, namely extraction/conversion, loading, abstraction, mining, and visualization. *DB* has the same phases except for the first two. *DB* does not perform the event logs extraction and loading. However, *DB* creates some overhead during the database processing, which in Figure 6 is illustrated similar to the loading phase in *Trad*. Note that here we do not incorporate the time needed to extract and convert event data into event logs files. If included, the total time of process discovery in *Trad* will be even longer than reported.

As shown in Figure 6, the mining and visualization time of the two approaches are similar as they utilize the same algorithm and are performed in the same process mining tool. It is also important to note that the models generated by *DB* are exactly the same as *Trad*, which shows that the in-database abstraction produces structures precisely the same as the in-memory abstraction and illustrates its correctness. On the other hand, the abstraction phase in *DB* takes longer than in *Trad*. This is not only because they use different implementation techniques, but also because the abstraction phase in *Trad* is executed in main memory, making it faster than *DB*. Nevertheless, *DB* beats *Trad* as the gain in DB-overhead is less than the loss in loading phase.

6.2 Update Function

In Section 4, we have shown that DF and MSD abstractions can be updated incrementally in a database. In this experiment, we aim to show the transformation of process models over time using the incrementally updated DF and MSD. We utilized the application process in the BPIC 2012 and discovered a process model over three different days of recorded data. Figure 7 illustrates the discovery. On day 1, not all activities have appeared. Three days later, all activities have emerged but we spot a sequence between activities A_registered (h), A_approved (i), and A_activated (j). This sequence turns into parallel activities with some loop on day 14. This demonstrates that processes are subject to alteration and process analyst may get insight into it by inspecting the process drift.

In the implementation, we utilize database triggers to update DF and MSD. In order to show the performance of this update step, we measure the processing time for handling the first 9000 events of the BPI 2012, road fine, and sepsis logs. Figure 8a depicts the performance measurement for all events. As expected, there is no increasing trend over time. Then, in order to see the trendline clearer, we zoom-in to events whose processing time are less than 5 ms (shown in Figure 8b). At the beginning the database needs more time to process events. We expect it is due to database initialization and some query cache that have not been activated in the beginning. However, after some point the processing time stabilizes. On average, the time needed for inserting one event and updating

Event log	*DB* (ms)	*Trad* (ms)
Sepsis	122.98	257.253
Road fine	1679.814	7234.506
Hospital	1624.431	6056.864
DailyLiving	66.94	78.009
BPIC12	666.358	2500.539
BPIC15-1	1487.932	1752.467
BPIC15-2	1627.339	1856.859
BPIC15-3	1313.825	1635.915
BPIC15-4	1109.465	1371.014
BPIC15-5	1364.234	1694.923
BPIC17	567.719	3243.25
BPIC18-1	528.849	1502.241
BPIC18-2	255.149	491.158
BPIC18-3	708.057	2637.466
BPIC18-4	1384.69	5265.335
BPIC18-5	442.081	1711.061
BPIC18-6	366.36	1186.907
BPIC18-7	2298.329	9121.744
BPIC18-8	436.011	1220.775
Xerox(jun)	2646.977	29 603.72
Xerox(nov)	220 470.585	297 184.513

Figure 5: Absolute time.

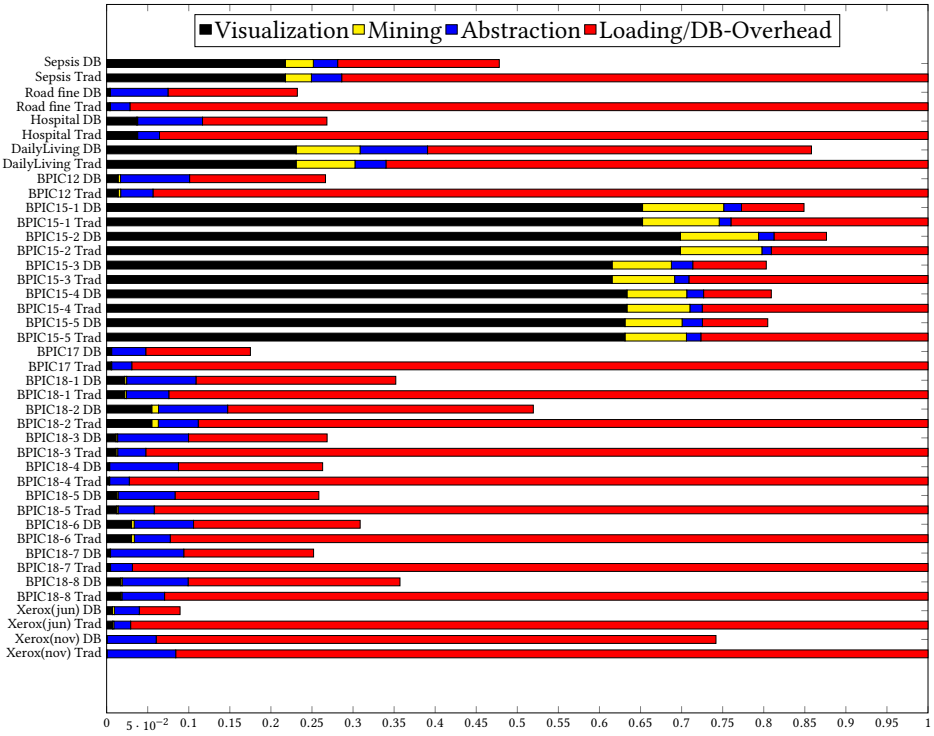


Figure 6: Relative time and discovery phases in *DB* vs *Trad*.

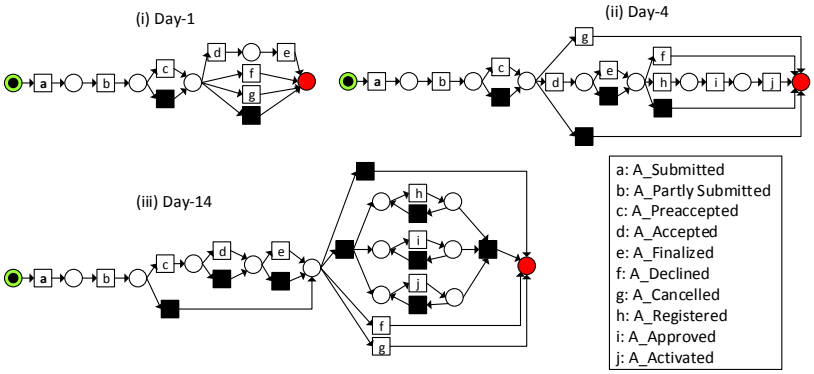


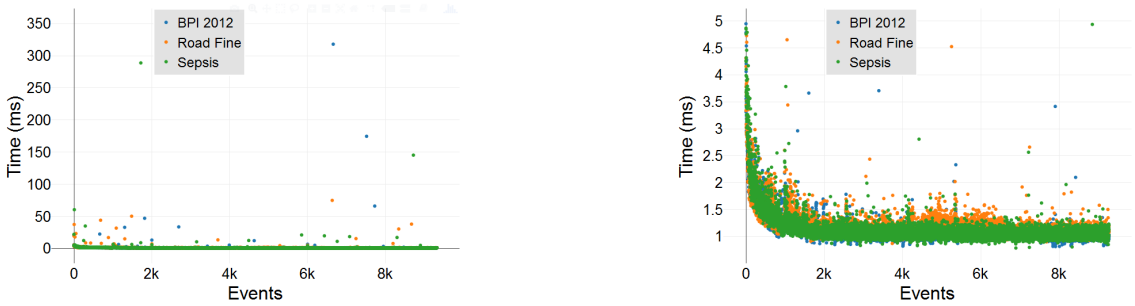
Figure 7: Process models are gradually changed over time as more data are inserted.

the corresponding abstractions was 1,2 ms. In other words, in one second the database can typically handle around 833 events.

6.3 Model Quality

In our third experiment, we evaluate the process models discovered by the *IMfw* algorithm. In particular, we investigate (Q1) whether *IMfw* improves over IMfd, and (Q2) whether its models are of a quality that is comparable to models discovered by techniques that use the full event log: (1) Inductive Miner - Infrequent (IMf) [12], (2) Indulpet Miner (IN) [14] (our experiment extends the evaluation in [14]), (3) Evolutionary Tree Miner (ETM) [3], (4) Split Miner (SM) [2], and (5) the baseline flower model (F) that allows for any behaviour. Therefore, we apply 7 process discovery techniques: for each log, we perform 3-fold cross validation: of these three parts, two parts are used for discovery while the remaining part is used for evaluation (measuring fitness [20], ETC-precision [15], simplicity). As some of the discovery techniques are not deterministic, this procedure is repeated 10 times. That is, for each combination of event log and technique, discovery is, in total, applied 30 times. Finally, as the ETC-precision computation is not deterministic, its computation is repeated 5 times for each of the 30 times.

Results. Table 1 shows the results. For the BPIC15 logs, ETC was unable to produce results within our memory and time constraints, so these results were obtained using the Projected Conformance



(a) (b)

Figure 8: Performance measurement: with all events (a), with removing outliers (b).

	f	p	s	f	p	s	f	p	s	f	p	s
	Roadfines			Sepsis			bpic12			bpic15-1		
IMf	0.98±0.00	0.79±0.04	97.97±4.78	0.91±0.02	0.41±0.06	140.23±13.85	0.97±0.01	0.59±0.03	186.57±9.59	1.00±0.00	0.66±0.04	1286.07±149.54
IN	0.98±0.00	0.79±0.02	96.50±7.00	0.91±0.02	0.43±0.05	122.43±8.66	0.36±0.07	0.88±0.05	116.30±30.52	0.78±0.01	0.97±0.01	91.50±54.38
ETM	0.89±0.08	0.68±0.27	117.93±85.12	!	!	!	!	!	!	!	!	!
SM	1.00±0.00	0.96±0.00	82.00±0.00	0.76±0.00	0.73±0.01	138.00±0.00	0.96±0.00	0.68±0.00	239.00±0.00	!	!	4715.20±8.88
IMfd	0.82±0.08	0.91±0.05	110.47±9.39	0.92±0.04	0.35±0.04	187.63±10.23	0.95±0.07	0.51±0.02	175.80±5.94	1.00±0.00	0.64±0.03	1350.17±178.09
IMfw	0.85±0.00	0.85±0.03	89.30±6.74	0.90±0.04	0.36±0.04	127.47±6.56	0.95±0.05	0.42±0.15	142.57±32.58	1.00±0.00	0.64±0.03	1289.63±168.41
F	1.00±0.00	0.38±0.01	46.00±0.00	1.00±0.00	0.21±0.00	61.00±0.00	1.00±0.00	0.11±0.00	85.00±0.00	1.00±0.00	0.64±0.01	1145.30±20.02
	bpic15-2			bpic15-3			bpic15-4			bpic15-5		
IMf	0.99±0.00	0.76±0.04	1048.10±167.84	1.00±0.00	0.71±0.03	1201.63±133.03	0.99±0.00	0.75±0.03	1082.47±98.26	0.99±0.00	0.79±0.05	1108.60±141.55
IN	0.74±0.01	0.96±0.01	114.10±23.71	0.79±0.01	0.97±0.01	118.37±80.59	0.76±0.02	0.93±0.03	272.17±63.98	0.75±0.02	0.96±0.02	244.17±120.50
ETM	!	!	!	!	!	!	!	!	!	!	!	!
SM	!	!	5077.27±10.28	!	!	3995.47±8.96	!	!	3882.73±5.00	!	!	4730.20±11.38
IMfd	1.00±0.00	0.66±0.02	1526.90±154.14	1.00±0.00	0.63±0.02	1433.27±156.39	1.00±0.00	0.67±0.03	1379.83±140.28	1.00±0.00	0.66±0.03	1519.63±160.24
IMfw	1.00±0.00	0.66±0.02	1479.50±134.67	1.00±0.00	0.65±0.03	1218.13±138.88	1.00±0.00	0.66±0.02	1287.80±74.46	1.00±0.00	0.65±0.02	1343.60±122.12
F	1.00±0.00	0.64±0.01	1172.70±24.44	1.00±0.00	0.66±0.01	1114.50±16.26	1.00±0.00	0.65±0.02	1024.30±27.16	1.00±0.00	0.65±0.02	1116.40±26.41
	bpic18-1			bpic18-2			bpic18-3			bpic18-4		
IMf	1.00±0.00	0.95±0.02	54.87±0.73	0.96±0.00	0.96±0.05	43.70±6.73	0.93±0.00	0.53±0.01	74.77±1.28	0.86±0.04	0.35±0.03	96.57±14.44
IN	1.00±0.00	0.95±0.02	54.87±0.73	0.96±0.00	0.96±0.05	43.70±6.73	0.79±0.04	0.95±0.04	55.60±17.73	0.61±0.19	0.83±0.27	71.97±50.49
ETM	0.97±0.04	0.95±0.12	77.50±86.35	0.99±0.00	0.76±0.21	183.93±91.24	!	!	!	!	!	!
SM	1.00±0.00	0.97±0.03	59.00±0.00	1.00±0.00	0.95±0.02	90.00±0.00	1.00±0.00	0.67±0.01	291.00±0.00	0.99±0.00	0.55±0.00	247.00±0.00
IMfd	1.00±0.01	0.81±0.09	65.77±1.28	1.00±0.01	0.76±0.05	83.87±6.53	0.87±0.10	0.44±0.05	247.53±45.40	!	!	218.17±22.80
IMfw	0.97±0.00	0.96±0.03	47.87±0.73	0.97±0.01	0.95±0.05	46.47±4.34	0.87±0.08	0.39±0.05	110.30±20.51	!	!	136.57±12.41
F	1.00±0.00	0.32±0.00	30.00±0.00	1.00±0.00	0.51±0.00	27.00±0.00	1.00±0.00	0.14±0.00	68.90±0.55	1.00±0.00	0.18±0.00	57.00±0.00
	bpic18-5			bpic18-6			bpic18-7			bpic18-8		
IMf	0.80±0.01	0.67±0.01	129.10±15.84	0.97±0.00	0.55±0.02	73.83±2.74	0.93±0.02	0.83±0.01	164.67±17.54	1.00±0.00	0.89±0.06	54.87±3.17
IN	0.78±0.01	0.69±0.05	164.00±43.89	0.97±0.00	0.55±0.02	73.83±2.74	0.89±0.03	0.84±0.02	220.43±70.34	1.00±0.00	0.89±0.05	54.87±3.17
ETM	!	!	!	!	!	!	!	!	!	!	!	!
SM	0.88±0.00	0.74±0.00	131.00±0.00	1.00±0.00	0.72±0.01	147.00±0.00	0.02±0.00	0.96±0.00	333.00±0.00	1.00±0.00	0.97±0.03	66.00±0.00
IMfd	0.76±0.06	0.67±0.11	118.97±23.81	0.94±0.00	0.57±0.02	121.87±0.73	0.87±0.04	0.49±0.04	296.97±42.23	0.99±0.01	0.74±0.08	75.43±10.25
IMfw	0.68±0.02	0.82±0.05	138.47±13.55	0.85±0.00	0.57±0.02	81.87±0.73	0.92±0.00	0.46±0.04	141.67±8.28	0.96±0.02	0.85±0.05	56.97±5.18
F	1.00±0.00	0.15±0.00	54.00±0.00	1.00±0.00	0.34±0.01	39.00±0.00	1.00±0.00	0.64±0.01	81.00±0.00	1.00±0.00	0.41±0.01	27.00±0.00

Table 1: Result of the model-quality evaluation. Missing results are denoted with !; results obtained using PCC are underlined.

Checking framework (PCC) [13]. ETM could not discover models for some event logs, and for 5 logs, models discovered by SM were not bounded and could be handled by neither ETC nor PCC. For BPIC18-4, the models obtained using *IMfw* could not be measured using ETC due to their size, thus we exclude these results here.

Considering Q1, the measured quality of the models of *IMfw* and IMfd is different for all event logs, which indicates that their models are different as well. In detail, compared to IMfd, *IMfw* has a higher fitness for Roadfines and BPIC18-7, and a higher precision for Sepsis, BPIC15-3, BPIC18-1, BPIC18-2, BPIC18-5 and BPIC18-8. Furthermore, the models discovered by *IMfw* were simpler for 14 out of the 16 logs. This shows that the MSD abstraction in fact influences discovery in real-life event logs.

Considering Q2, *IMfw* obtained (on average) pareto-optimal results for 6 logs (BPIC12, BPIC15-2, BPIC15-4, BPIC18-1, BPIC18-2, BPIC18-5) of the 14 successfully tested logs. That is, for these event logs, no model discovered by any other technique is as good as the model discovered by *IMfw* on all dimensions, and better on at least one. This does not ensure that *IMfw* is the best choice for all imaginable use cases, however it shows that even though the *IMfw* algorithm uses less information from event logs, it can nevertheless discover optimal results compared to techniques that use the entire event log (IMf, IN, ETM, SM, F), while being more broadly applicable than ETM and SM.

Finally, *IMfw*'s results tend to have large variations, which, given the cross validation procedure, we believe is due to *IMfw*'s use of

abstractions, causing it to use less information from the event log than other techniques.

7 CONCLUSION

Process mining can create value for organizations using the omnipresent availability of event data nowadays. However, as event data are typically logged in legacy information systems, such as databases, process discovery techniques require these data to be transformed and extracted into main memory or into standardized environments. To avoid these transformation and extraction steps, some works [16, 18] introduced a direct approach by processing event data in databases. These works apply a separation of duties to process discovery by splitting it into two phases: first, an abstraction of an event log is constructed in a single pass over the event log, after which from the abstraction a process model is constructed. Through this strategy, the handling of event logs is decoupled from the mining activity, thus it is neither necessary to load the data into main memory, nor to transform data into a specific format, nor to copy the data to a different environment. However, these techniques only focus on the simplest abstraction called the Directly Follows (DF) abstraction.

In this paper, we exploited another abstraction to extend the class of processes that can be discovered: the Minimum Self Distance (MSD) abstraction. As with the DF abstraction, we compute the MSD abstraction in-database and keep both abstractions *live*: for each new insertion of event data, DF and MSD are updated automatically. Through this approach, we avoid the need to extract and transform event data. Furthermore, the DF and MSD abstractions are always ready for the subsequent mining phase, which avoids peak loads on production systems and saves process analysts time.

The process discovery phase of our approach consists of a new process discovery framework (*IMw*) and an algorithm implementing it (*IMfw*), that takes a DF and MSD abstraction to discover a process model by recursively searching for cuts, until a fallthrough or a base case is found. Since *IMw* only uses information from the DF and MSD abstractions (and not the full event logs), it paves the way for process discovery on large and complex event logs as the abstractions are considerably smaller than the full event logs.

We evaluated our approach on three aspects. The first experiment showed that in-database abstraction followed by *IMfw* yields performance benefits in terms of computation time and memory usage compared to a traditional in-memory abstraction approach followed by *IMfw*. The second experiment revealed how the in-database abstraction progressively builds the DF and MSD abstrations, and that the time required to process events has a constant trend after an initialization period, such that the live abstractions have a constant performance impact, independent of the size of the log. Finally, the third experiment showed that despite its use of abstractions, *IMfw* discovered pareto-optimal models (measured using fitness, precision, and simplicity values) from real-live event logs compared to existing process discovery techniques, even though it uses less information from the logs. For future work, we aim to further investigate rediscoverability of *IMw* and *IMfw*, to extend the in-database abstraction with other abstractions, and to extend discovery techniques accordingly.

REFERENCES

[1] [n. d.]. SAP Process Mining by Celonis. https://www.sap.com/developer/showcases/process-mining-by-celonis.html. Accessed: 2019-03-05.

[2] A. Augusto, R. Conforti, M. Dumas, and M. La Rosa. 2017. Split Miner: Discovering Accurate and Simple Business Process Models from Event Logs. In *ICDM 2017*. 1–10. https://doi.org/10.1109/ICDM.2017.9

[3] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. 2012. A Genetic Algorithm for Discovering Process Trees. In *IEEE CEC*. 1–8. https://doi.org/10.1109/CEC.2012.6256458

[4] D. Calvanese, T.E. Kalayci, M. Montali, and S. Tinella. 2017. Ontology-based Data Access for Extracting Event Logs from Legacy Data: The onprom Tool and Methodology. In *BIS 2017*.

[5] D. Calvanese, M. Montali, A. Syamsiyah, and W.M.P. van der Aalst. 2015. Ontology-Driven Extraction of Event Logs from Relational Databases. In *BPM workshops*. 140–153. https://doi.org/10.1007/978-3-319-42887-1_12

[6] E.G.L. de Murillas, W.M.P. van der Aalst, and H.A. Reijers. 2015. Process Mining on Databases: Unearthing Historical Data from Redo Logs. In *BPM 2015*. 367–385. https://doi.org/10.1007/978-3-319-23063-4_25

[7] R.M. Dijkman, J. Gao, A. Syamsiyah, B.F. van Dongen, P. Grefen, and A. ter Hofstede. 2019. Enabling Efficient Process Mining on Large Data Sets: Realizing an In-database Process Mining Operator. *Distributed and Parallel Databases* (09 May 2019). https://doi.org/10.1007/s10619-019-07270-1

[8] E. Gonzalez. 2019. *Process Mining on Databases: Extracting Event Data from Real-life Data Sources*. Ph.D. Dissertation. TU Eindhoven.

[9] C.W. Günther. 2014. XES Standard Definition. www.xes-standard.org.

[10] S.J.J. Leemans. 2017. *Robust Process Mining with Guarantees*. Ph.D. Dissertation. TU Eindhoven.

[11] S.J.J. Leemans and D. Fahland. [n. d.]. Information-Preserving Abstractions of Event Data in Process Mining. *KAIS* accepted ([n. d.]).

[12] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. 2013. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *BPM Workshops 2013*. https://doi.org/10.1007/978-3-319-06257-0_6

[13] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. 2018. Scalable process discovery and conformance checking. *Software and System Modeling* 17, 2 (2018), 599–631. https://doi.org/10.1007/s10270-016-0545-x

[14] S.J.J. Leemans, N. Tax, and A.H.M. ter Hofstede. 2018. Indulpet Miner: Combining Discovery Algorithms. In *CoopIS*. 97–115. https://doi.org/10.1007/978-3-030-02610-3_6

[15] J. Munoz-Gama. 2016. *Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes*. LNBIP, Vol. 270. Springer. https://doi.org/10.1007/978-3-319-49451-7

[16] A. Syamsiyah, B.F. van Dongen, and R. Dijkman. 2018. A Native Operator for Process Discovery. In *DEXA 2018*. 292–300. https://doi.org/10.1007/978-3-319-98812-2_25

[17] A. Syamsiyah, B.F. van Dongen, and W.M.P. van der Aalst. 2016. DB-XES: Enabling Process Mining in the Large. In *SIMPDA 2016*. 63–77. http://ceur-ws.org/Vol-1757/paper5.pdf

[18] A. Syamsiyah, B.F. van Dongen, and W.M.P van der Aalst. 2017. Recurrent Process Mining with Live Event Data. In *BPI 2017*. 178–190. https://doi.org/10.1007/978-3-319-74030-0_13

[19] W.M.P. van der Aalst. 2016. *Process Mining: Data Science in Action*. Springer.

[20] W.M.P. van der Aalst, A. Adriansyah, and B.F. van Dongen. 2012. Replaying History on Process Models for Conformance Checking and Performance Analysis. *DMKD* 2, 2 (2012), 182–192. https://doi.org/10.1002/widm.1045

[21] W.M.P. van der Aalst, A.J.M.M. Weijter, and L. Maruster. 2003. Workflow Mining: Discovering Process Models from Event Logs. *TKDE* 16 (2003), 2004.

[22] J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. 2008. Process Discovery Using Integer Linear Programming. In *Applications and Theory of Petri Nets*. Springer Berlin Heidelberg, Berlin, Heidelberg, 368–387.

[23] B.F. van Dongen and S. Shabani. 2015. Relational XES: Data Management for Process Mining. In *CAiSE 2015*. 169–176. http://ceur-ws.org/Vol-1367/paper-22.pdf

[24] S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. 2016. Online Discovery of Cooperative Structures in Business Processes. In *OTM Conferences 2016*. 210–228. https://doi.org/10.1007/978-3-319-48472-3_12

[25] S.J. van Zelst, B.F. van Dongen, and W.M.P. van der Aalst. 2018. Event Stream-based Process Discovery Using Abstract Representations. *Knowl. Inf. Syst.* 54, 2 (2018), 407–435. https://doi.org/10.1007/s10115-017-1060-2

[26] S.K.L.M. vanden Broucke and J. De Weerdt. 2017. Fodina: A Robust and Flexible Heuristic Process Discovery Technique. *Decision Support Systems* 100 (2017), 109 – 118. https://doi.org/10.1016/j.dss.2017.04.005

[27] H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. 2010. XES, XESame, and ProM 6. In *Information Systems Evolution*, Vol. 72. 60–75.

[28] A.J.M.M. Weijters and J.T.S. Ribeiro. 2011. Flexible Heuristics Miner (FHM). In *CIDM 2011*. 310–317. https://doi.org/10.1109/CIDM.2011.5949453