ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/359765817

A Generic Trace Ordering Framework for Incremental Process Discovery

Chapter · January 2022

DOI: 10.1007/978-3-031-01333-1_21

CITATIONS
0

READS
112

4 authors, including:

Daniel Schuster
Fraunhofer Institute for Applied Information Technology FIT
16 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Sebastiaan van Zelst
Fraunhofer Institute for Applied Information Technology FIT
80 PUBLICATIONS **732** CITATIONS
SEE PROFILE

Wil Van der Aalst
RWTH Aachen University
1,438 PUBLICATIONS **85,059** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Foundations of Process Mining View project

Process Discovery View project

All content following this page was uploaded by Daniel Schuster on 15 April 2022.

The user has requested enhancement of the downloaded file.

A Generic Trace Ordering Framework for Incremental Process Discovery

Daniel Schuster$^{1,2[0000-0002-6512-9580]}$, Emanuel Domnitsch^{2}, Sebastiaan J. van Zelst$^{1,2[0000-0003-0415-1036]}$, and Wil M. P. van der Aalst$^{1,2[0000-0002-0955-6940]}$

[1] Fraunhofer Institute for Applied Information Technology FIT, Sankt Augustin, Germany
{daniel.schuster, sebastiaan.van.zelst}@fit.fraunhofer.de
[2] RWTH Aachen University, Aachen, Germany
emanuel.domnitsch@rwth-aachen.de, wvdaalst@pads.rwth-aachen.de

Abstract. Executing operational processes generates valuable event data in organizations' information systems. Process discovery describes the learning of process models from such event data. Incremental process discovery algorithms allow learning a process model from event data gradually. In this context, process behavior recorded in event data is incrementally fed into the discovery algorithm that integrates the added behavior to a process model under construction. In this paper, we investigate the open research question of the impact of the ordering of incrementally selected process behavior on the quality, i.e., recall and precision, of the learned process models. We propose a framework for defining ordering strategies for traces, i.e., observed process behavior, for incremental process discovery. Further, we provide concrete instantiations of this framework. We evaluate different trace-ordering strategies on real-life event data. The results show that trace-ordering strategies can significantly improve the quality of the learned process models.

Keywords: Process mining · Process discovery · Ordering effects

1 Introduction

Process mining [17] offers tools and methods to systematically analyze data generated during the execution of operational processes, e.g., business and production processes. These data are referred to as *event data*, which can be extracted from organizations' information systems. Process mining aims to generate valuable insights into the processes under investigation to optimize them ultimately.

Process discovery, a key discipline within process mining, comprises algorithms that learn process models from event data. Most process model formalisms focus on describing the control flow of process activities. Note that process model formalisms like BPMN [7] allow modeling, e.g., resource information and data flows, besides the control flow of process activities. In short, process models are an essential artifact within process mining.

Conventional process discovery algorithms [2] are fully automated. Other than configuring parameter settings, they do not provide any form of interaction. Thus, they function as a black box from a user's perspective. Since event data often have quality issues, e.g., wrongly captured, missing, and incomplete process behavior, process discovery can be considered an unsupervised learning task. Many conventional process discovery algorithms yield low-quality models on real-life event data. Automated filtering techniques, such as [4], attempt to solve such data quality problems but often remove too much process behavior. In addition, they cannot add missing process behavior to the event data.

Domain-knowledge-utilizing process discovery aims to overcome the limitations of conventional process discovery by using additional knowledge about the process under consideration besides event data and by incorporating user feedback into the discovery, respectively, learning phase [16]. *Incremental* process discovery is a subclass of domain-knowledge-utilizing process discovery where the user gradually selects process behavior that is added to a process model under construction by the discovery algorithm. With incremental process discovery, a user can, for example, examine the process model after each incremental execution and, if necessary, jump back to a previous version of the model and add other observed process behavior. In this way, the user can steer and influence the discovery phase compared to conventional process discovery.

In previous work [14], we introduced an incremental process discovery algorithm that allows to gradually add process instances, i.e., individual process executions, which are also referred to as *traces*, to a process model under construction. An open research question is the influence of the order in which the process behavior is gradually inserted into the process model under construction on the quality of the eventual process model discovered. In this paper, we address this research question by exploring strategies to recommend a trace order. From a practical perspective, these strategies are helpful in situations where, for example, a user selects several traces at once to be added next but does not have any preferences about the exact order in which they are added to the model by the incremental discovery algorithm.

This paper contains two main contributions. First, we define a general framework for trace-ordering strategies within the context of incremental process discovery. The framework can be applied for any incremental process discovery algorithm that gradually adds traces to a process model. Second, we provide instantiations of this framework, i.e., various trace-ordering strategies for an existing incremental process discovery algorithm [14]. Finally, we present an evaluation of the proposed strategies. Our experiments show that using trace-ordering strategies results in significantly better models than random trace selection, cf. [15].

The remainder of this paper is organized as follows. In Section 2, we present related work. Section 3 introduces preliminaries. In Section 4, we introduce a framework for trace-ordering strategies and provide specific instantiations of this framework. The evaluation by use of real-life event data of these instantiations is presented in Section 5. Finally, Section 6 concludes this paper.

2 Related Work

For a general introduction to process mining, we refer to [17]. In this section, we mainly focus on process discovery. Compared to, e.g., sequential pattern mining [1], process discovery aims to return process models describing the end-to-end control-flow of activities within a process. We refer to [17] to further differentiate process mining from existing data mining techniques. Many conventional process discovery algorithms have been developed; a recent overview can be found in [2]. Regarding the field of domain-knowledge-utilizing process discovery, we refer to [16] for a recent overview. One of the first approaches to interactive process discovery was presented in [6]. The approach involves a user creating a process model gradually in an editor while being supported by the algorithm with suggestions. Regarding incremental process discovery, few approaches exist. In [14] an incremental process discovery algorithm has been introduced that produces process trees. In [10] an incremental approach has been proposed that represents the process as a set of first-order logic formulae. Techniques for *model repair* [8], a research area within process mining, can also be utilized as incremental discovery.

To the best of our knowledge, no related work focuses on trace ordering neither within incremental process discovery nor within process model repair. Outside of process mining, in the context of AI/ML, the influence of ordering data on the learning results has been addressed, for example, in [5,13].

3 Preliminaries

For an arbitrary set X, we define the set of all sequences over X as X^*, e.g., $\langle b,a,b\rangle \in \{a,b,c\}^*$. We denote a totally ordered set by $(X,\preceq)$. Given a base set X, we denote the universe of all totally ordered sets by $\mathcal{O}(X)$. A multi-set allows for multiple occurrences of the same element. We denote the set of all possible multi-sets over a base set X as $\mathcal{B}(X)$ and the power set as $\mathcal{P}(X)$.

3.1 Event Data

Event data are generated during the execution of operational processes. Table 1 shows an example of an event log. Each row corresponds to an unique event that records the execution of an activity for a specific process instance. Process instances are identified by a case-id. Events that belong to the same process instance, i.e., that have the same case ID, form a trace, i.e., a sequence of events ordered by their timestamp, for example. Consider Table 1, the trace of the process instance "A10000" is ⟨"Create Fine", "Send Fine", "Insert Fine Notification", "Add penalty", "Payment"⟩. An event log typically consists of multiple traces. Next, we formally define the concept of a trace and an event log. In the remainder of this paper, we denote the universe of activity labels by $\mathcal{A}$.

Definition 1 (Trace & Event Log). *A trace is a sequence of activity labels, i.e.,* $\sigma \in \mathcal{A}^*$. *An event log is a multi-set of traces, i.e.,* $E \in \mathcal{B}(\mathcal{A}^*)$. *We denote the universe of event logs by* $\mathcal{E} = \mathcal{B}(\mathcal{A}^*)$.

Table 1: Real-life event log from a road traffic fine management process [12]

Event-ID	Case-ID	Activity label	Timestamp	...
1	A10000	Create Fine	09.03.2007	...
2	A10000	Send Fine	17.07.2007	...
3	A10000	Insert Fine Notification	02.08.2007	...
4	A10000	Add penalty	01.10.2007	...
5	A10000	Payment	09.09.2008	...
6	A10001	Create Fine	19.03.2007	...
7	A10001	Send Fine	17.07.2007	...
8	A10001	Insert Fine Notification	25.07.2007	...
9	A10001	Insert Date Appeal to Prefecture	02.08.2007	...
10	A10001	Add penalty	23.09.2007	...
...	...	...	...	...

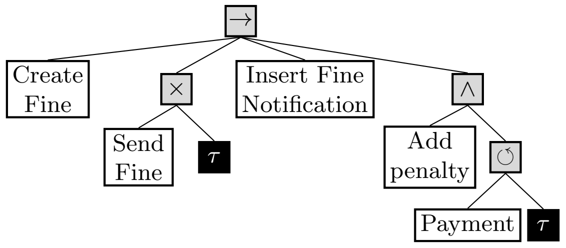


Fig. 1: Example of a process model represented as a process tree

Since we are only interested in the various sequences of executed process activities and multiple cases can have the same sequence of executed activities, we define an event log as a multi-set of traces. For an event log $E \in \mathcal{E}$, we write $\overline{E} = \{\sigma \in E\} \subseteq \mathcal{A}^*$ to denote the set of unique traces. For instance, given the event log $E = \left[\langle a,b,c\rangle^5, \langle a,b,b\rangle^3\right]$, i.e., an event log containing five times the trace $\langle a,b,c\rangle$ and three times the trace $\langle a,b,b\rangle$, $\overline{E} = \left\{\langle a,b,c\rangle, \langle a,b,b\rangle\right\}$.

3.2 Process Models

Process models describe process behavior, especially the control flow of process activities. For example, consider Figure 1, showing a process tree, i.e., an important process model formalism within process mining [17]. The process tree specifies that the activity 'Create Fine' is executed first. Next, 'Send Fine' is optionally executed, followed by 'Insert Fine Notification'. Finally, 'Add Penalty' is executed parallel to potentially multiple executions of 'Payment'. A formal definition of process trees is outside the scope of this paper; we refer to [17].

This paper abstracts from a specific process model formalism, e.g., Petri nets or process trees. Thus, we generally define the universe of process models by $\mathcal{M}$. Each process model M defines a language, i.e., a set of accepted traces. As such, we denote the language of a process model $M \in \mathcal{M}$ by $L(M) \subseteq \mathcal{A}^*$.

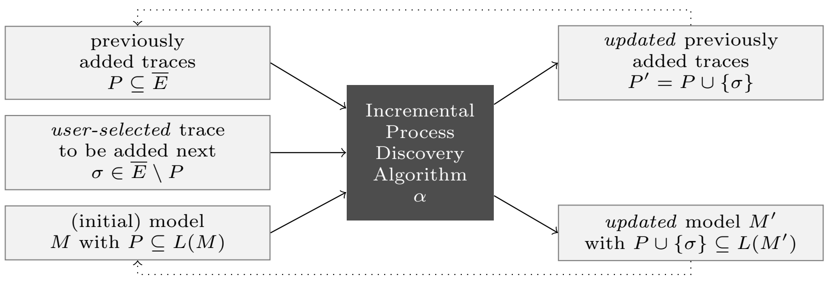

previously added traces $P \subseteq \overline{E}$
user-selected trace to be added next $\sigma \in \overline{E} \setminus P$
(initial) model M with $P \subseteq L(M)$
Incremental Process Discovery Algorithm α
updated previously added traces $P' = P \cup \{\sigma\}$
updated model M' with $P \cup \{\sigma\} \subseteq L(M')$

Fig. 2: Overview of the procedure of an incremental process discovery algorithm

3.3 Incremental Process Discovery

Conventional process discovery algorithms can be seen as a function $d : \mathcal{E} \to \mathcal{M}$. Incremental process discovery algorithms form a specific class of process discovery algorithms that gradually learn a process model. Figure 2 shows an overview of the procedure used by incremental process discovery algorithms. Given an event log $E \in \mathcal{E}$ and an (initial) process model $M \in \mathcal{M}$, a trace $\sigma \in E$ that is added by the incremental process discovery algorithm to the process model M. The resulting process model M', which describes the previously added traces P and the trace σ, is then used as an input in the next iteration. Next, we formally define an incremental process discovery algorithm.

Definition 2 (Incremental Process Discovery Algorithm). *The function $\alpha : \mathcal{M} \times \mathcal{P}(\mathcal{A}^*) \times \mathcal{A}^* \to \mathcal{M}$ is an incremental process discovery algorithm if for any process model $M \in \mathcal{M}$, set of previously added traces $P \in \mathcal{P}(\mathcal{A}^*)$ with $P \subseteq L(M)$, and trace to be added $\sigma \in \mathcal{A}^*$ it holds that $P \cup \{\sigma\} \subseteq L(\alpha(M, P, \sigma))$.*

4 Dynamic Trace-Ordering Strategies

In this section, we present the proposed approach to order trace candidates. First, we present a general framework on how to define trace-ordering strategies for incremental process discovery. Finally, we present concrete strategies.

4.1 General Framework

In this section, we present the proposed framework for defining Dynamic Trace-Ordering Strategies (DTOS) for incremental process discovery. In Figure 3, we depict the proposed framework. A DTOS consists of $n \geq 1$ sequentially applied *strategy components (sc)*. Each strategy component sc_i orders all input trace candidates C_{i-1} according to some internal logic. The calculated ordering of the trace candidates represents a ranking which trace should be added next to the process model M. A 'minimal' trace in $(C_{i-1}, \preceq)$ represents the most suitable candidate to be added next according to the current strategy component sc_i. Next, we formally define a strategy component.

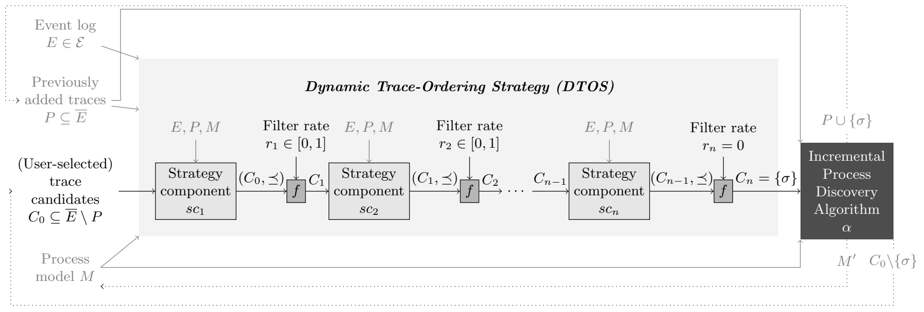

Event log $E \in \mathcal{E}$
Previously added traces $P \subseteq \overline{E}$
(User-selected) trace candidates $C_0 \subseteq \overline{E} \setminus P$
Process model M
Dynamic Trace-Ordering Strategy (DTOS)
E, P, M
Filter rate $r_1 \in [0,1]$
Strategy component sc_1
$(C_0, \preceq)$
f
C_1
Strategy component sc_2
Filter rate $r_2 \in [0,1]$
$(C_1, \preceq)$
C_2
C_{n-1}
Strategy component sc_n
Filter rate $r_n = 0$
$(C_{n-1}, \preceq)$
$C_n = \{\sigma\}$
$P \cup \{\sigma\}$
Incremental Process Discovery Algorithm α
M'
$C_0 \setminus \{\sigma\}$

Fig. 3: Proposed framework for dynamic trace-ordering strategies that consist of sequentially aligned strategy components, sc_1 until sc_n, which order the trace candidates from best to worst suitability. After each strategy component, a filter function f removes the worst suited trace candidates. After the last strategy component sc_n, the function f filters such that a single trace candidate σ remains that is then fed into the incremental process discovery algorithm.

Definition 3 (Strategy component). *A strategy component is a function* $sc : \mathcal{E} \times \mathcal{P}(\mathcal{A}^*) \times \mathcal{P}(\mathcal{A}^*) \times M \rightarrow \mathcal{O}(\mathcal{A}^*)$ *that maps an event log* $E \in \mathcal{E}$*, a set of previously added traces* $P \subseteq \overline{E}$*, a set of trace candidates* $C \subseteq P \setminus \overline{E}$*, and a process model* $M \in \mathcal{M}$ *to an ordered set of trace candidates* $(C, \preceq) \in \mathcal{O}(C)$*. We denote the universe of strategy components by* $\mathcal{SC}$*.*

After each strategy component, a filter function f filters out the worst suited trace candidates. Each call of the filter function f can be configured via a filter rate $r_i \in [0, 1]$. A filter rate of 1 results in no trace candidate being filtered. A filter rate of 0 results in only one trace candidate remaining. Thus, $C_n \subseteq C_{n-1} \subseteq ... \subseteq C_1 \subseteq C_0$ holds (cf. Figure 3). Each strategy component together with the subsequent filtering can be viewed as a knock-out step that reduces the number of trace candidates that are potentially to be added next to the process model M. Next, we formally define a filter function and a DTOS.

Definition 4 (Filter function). *A filter function* $f : \mathcal{O}(A^*) \times [0, 1] \rightarrow \mathcal{P}(\mathcal{A}^*)$ *maps an ordered set of traces* $(C, \preceq) \in \mathcal{O}(\mathcal{A}^*)$ *and a filter rate* $r \in [0, 1]$ *to a set of traces* $C' \in \mathcal{P}(\mathcal{A}^*)$ *such that* $C' \subseteq C$*,* $|C'| = max\{1, \lceil r * |C| \rceil\}$*, and* $\forall c' \in C' \forall c \in C \setminus C' (c' \leq c)$*.*

Definition 5 (Dynamic trace-ordering strategy (DTOS)). *A DTOS is a non-empty sequence of strategy components and corresponding filter rates, i.e.,* $\langle (sc_1, r_1), \ldots, (sc_n, r_n) \rangle \in (\mathcal{SC} \times [0, 1])^*$ *with* $r_n = 0$ *for* $n \geq 1$*.*

We consciously decided to design a DTOS as a sequence of strategy components and filters that work in a knock-out fashion, i.e., every strategy component orders the trace candidates, and subsequently, a function filters out the worst trace candidates. This decision was taken to keep the computational effort low because it is crucial to compute recommendations fast in an interactive process discovery setting. The use of multiple strategy components within a DTOS allows combining different aspects when evaluating which trace candidate should be added next. The general intention of the framework is to initially perform evaluations that are fast to calculate within the first strategy components. More complex evaluations should be performed in the later strategy components. These components will receive fewer trace candidates since the previously executed strategy components have already filtered out some trace candidates.

4.2 Instantiations

Here, we present specific strategy components, i.e., instantiations of Definition 3. We provide general applicable strategy components that are independent of a specific incremental process discovery algorithm and strategy components which are specifically tailored for the incremental process discovery algorithm introduced in previous work [14]. Next, we briefly present six strategy components.

Alignment Costs Alignments [18] are a state-of-the-art conformance checking technique that quantifies to which extent a trace can be replayed on a process model. They further provide diagnostic information on missing and unexpected behavior when comparing a trace with a process model. The costs of an optimal alignment reflect the conformance degree of the trace and the closest process model execution. Given a process model, we can assign costs, i.e., alignment costs, to each trace candidate. These costs are then used to rank/sort the trace candidates, i.e., the trace candidate with the lowest costs first and the trace candidate with the highest costs last. The intention is to first add trace candidates to the process model that are close to the specified behavior by the current process model. Note that the computation of alignments has an exponential time complexity, i.e., also called the *state space explosion problem* [3].

Missing Activities When starting to discover a process model incrementally, it is likely that the first process models obtained do not describe all process activities that have been recorded in the event data. This results from the fact that in many real-life event logs not every trace contains all possible executable process activities of a process. The 'missing activities strategy' ranks the trace candidates according to their number of activity labels already present in the process model M. Trace candidates that contain process activities present in the current process model M get costs 0. For the other trace candidates, costs correspond to the number of unique activity labels within the trace that are not yet part of the process model M.

Levenshtein Distance This strategy compares the trace candidates among each other by calculating the Levenshtein distance, i.e., a metric to compare the distance between two sequences based on edit operations: insertion, deletion, and substitution. The idea behind this strategy is to favor traces that are more similar to all other traces that still need to be potentially added. For example, consider the three trace candidates with corresponding frequency values in Table 2. We compare all traces and weigh the different Levenshtein distances according to the trace frequency. In the example, we would choose the trace $\langle a, b\rangle$ as the best trace candidate to be added next.

Table 2: Example of the weighted Levenshtein distance for trace candidates

Trace candidates	Frequency in E	Weighted Levenshtein distance	Rank
$\langle a, b\rangle$	100	$50 * lev(\langle a, b\rangle, \langle a, b, b\rangle) + 20 * lev(\langle a, b\rangle, \langle a, c\rangle) = 70$	1
$\langle a, b, b\rangle$	50	$100 * lev(\langle a, b, b\rangle, \langle a, b\rangle) + 20 * lev(\langle a, b, b\rangle, \langle a, c\rangle) = 140$	2
$\langle a, c\rangle$	20	$100 * lev(\langle a, c\rangle, \langle a, b\rangle) + 50 * lev(\langle a, c\rangle, \langle a, b, b\rangle) = 200$	3

Brute-Force Assume a process model M, an event log $E \in \mathcal{E}$, the set of previously added traces $P \subseteq \overline{E}$, and a set of trace candidates $C \subseteq \overline{E} \setminus P$ (cf. Figure 3). The strategy separately applies the incremental process discovery (cf. Figure 2) to all trace candidates in C and the model M. As a result, $|C|$ different process models are obtained. A quality metric, i.e., the F-measure representing the harmonic mean of recall and precision, is calculated on the given event log E for each obtained model. The trace candidate that yields a process model with the highest F-measure is ranked first.

LCA Height This strategy is tailored to the incremental process discovery algorithm introduced in our earlier work [14]. The incremental process discovery algorithm uses process trees (cf. Figure 1) as a process model formalism. When incrementally adding a new trace to the model, the central idea of the algorithm is to identify subtrees that need to be modified so that the new trace fits the language of the model. These deviating subtrees are called LCAs in [14]. Depending on which trace is added, the LCAs that must be altered change.

The key idea of this strategy is to avoid changing large parts of the already learned process model upon adding a new trace. Thus, the strategy prefers trace candidates that lead to only minor changes in the process model. Therefore, the strategy computes for each trace candidate in $C \subseteq \overline{E}$ the height of the first LCA, i.e., the first subtree in the process model M, that must be altered[3]. The height of an LCA is defined by the path length from the LCA's root node to the root node of the entire tree, i.e., the entire process model M. Trace candidates are then descending ordered based on the first LCA's height.

Duplicates This strategy is tailored to the incremental process discovery algorithm introduced in our earlier work [14]. These LCAs, i.e., subtrees of the

[3] Note that per trace that is incrementally added, various LCAs might be changed. However, without fully executing the incremental process discovery approach for a trace, we only can compute the first LCA that must be changed. Therefore, there is a risk that the first LCA will be rated as good based on the strategy, but that further LCAs will have to be changed, which the strategy would rate as bad.

Table 3: Overview of the strategy components

Abbreviation	Strategy component (Section 4.2)	Algorithmic specific or general
C	Alignment Costs	general
M	Missing Activities	general
L	Levenshtein Distance	general
B	Brute Force	general
D	Duplicates	specific
H	LCA Height	specific

process tree, may have multiple leaf nodes with the same activity label, i.e., *duplicate labels*. In general, duplicate labels can increase the precision of a process model and are therefore desirable. When altering an LCA, the incremental process discovery algorithm [14] rediscovers the LCA using a conventional process discovery algorithm [11]. The downside of this rediscovery is that the used conventional process discovery algorithm [11] is not able to discover process trees with duplicate labels. Thus, the rediscovery would remove the potentially desirable duplicate labels in the LCA that have been learned so far. Thus, this strategy, called *Duplicates*, favors trace candidates whose first LCA does not contain leaf nodes with duplicate labels. Trace candidates are ascending ordered based on the number of duplicate leaf nodes.

5 Evaluation

In this section, we present the experimental evaluation. First, we present the experimental setup. Subsequently, we present and discuss the results.

5.1 Experimental Setup

To keep the experimental setup independent of a particular user selecting trace candidates to be added next (cf. Section 1), we assumed the following: given an event log and an initial process model, all traces are eventually added to the model incrementally. Thus, the set of trace candidates represents in the beginning the entire event log. After one incremental discovery step—a trace selected by an ordering strategy is added to the model by the incremental discovery algorithm—the added trace is removed from the trace candidate set.

Given the strategy components' abbreviations in Table 3, we created all potential orderings by shuffling the order of C, M, L, D, and H. Finally, strategy component B is added to each strategy. Note that the brute force (B) strategy component is computationally expensive, and therefore we decided to always add this strategy component at the end. This procedure leads to 5! = 120 different strategy component orderings. To avoid further expansion of the parameter space, we used one filter rate for each strategy component within a strategy, except the last one (cf. Figure 3). For instance, the strategy L-H-C-M-D-B F-Rate 10 (cf. Figure 4) represents the strategy where first the Levenshtein distance component is applied and finally the brute force component. All components

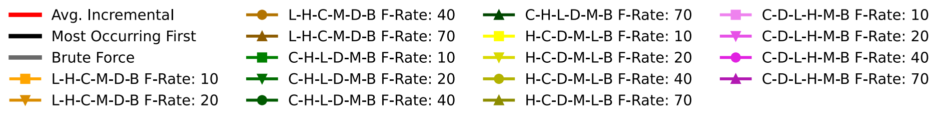


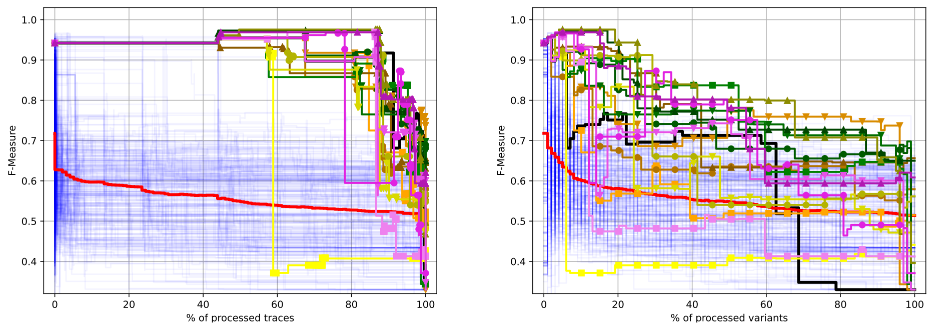
Fig. 4: Legend for the results shown in Figure 5 and Figure 6

within this specific strategy use a filter rate of $0.1 \widehat{=} 10\%$. We applied the different strategies on real-life event logs using the incremental process discovery algorithm presented in [14]. Further, we measured the F-measure, i.e., the harmonic mean of recall and precision, of each incrementally discovered process model using the given event log. We used four publicly available real-life event logs [19,12,9].

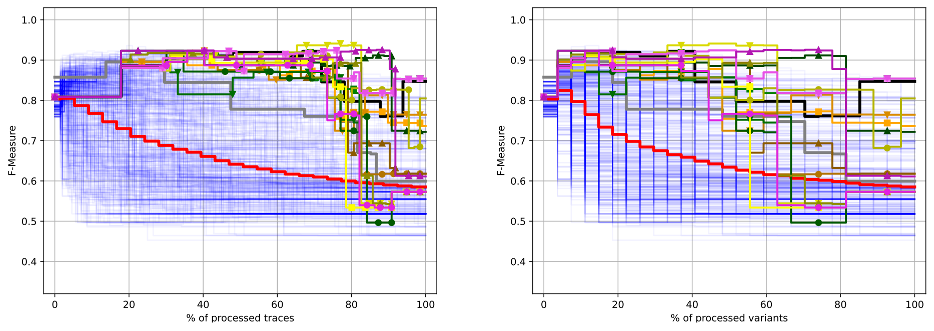
5.2 Results & Discussion

In Figure 5, we depict the results of 16 dynamic strategies, a static strategy, i.e., most occurring trace variant first (black line), the brute force component as a stand-alone strategy (gray line), random trace orderings (blue lines), and the average of the random trace orderings (red line). Note that we only show a selection of the strategies evaluated. Per log, we provide two x-axis scales: percentage of processed traces and percentage of uniquely processed trace variants.

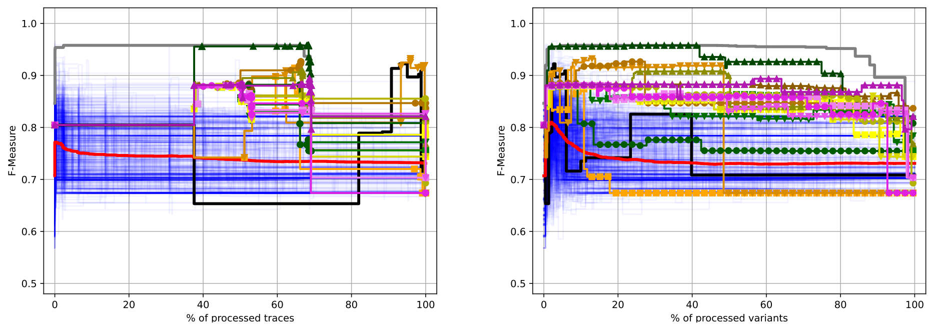
We observe that for all four event logs, the trace candidate order has a significant impact on the F-measure, cf. the large area covered by the blue lines in Figure 5. The solid red line represents the average of the blue lines. Thus, the red line can be seen as a baseline as it represents the quality of the models if a random trace order is applied. We see that *most strategies are clearly above the red line*. Thus, applying a strategy is often better than randomly selecting trace candidates. Note that with incremental process discovery, the goal is often *not* to include all traces from the event log, as event logs often have data quality issues. We observe that the brute force approach as a stand-alone strategy (gray line) often performs better than the other strategies, although the brute force approach can be considered as a *greedy* algorithm. For the *domestic* event log, the brute force as a stand-alone strategy could not be used as the calculation was still not completed after several days. In Figure 6 we depict the computation time per strategy. In general, we observe that an increasing filter rate per strategy component ordering leads to an increasing computation time. This observation can be explained because each strategy includes the brute force strategy component as the last step. We also find that the brute force approach as a single strategy (gray bar) has a significantly longer computation time than the other strategies. In short, it can be stated that many of the presented strategies lead to better process models, i.e., outperforming randomly selecting a trace to be added. Nevertheless, no clear strategy can be identified that always works best on all tested event logs.



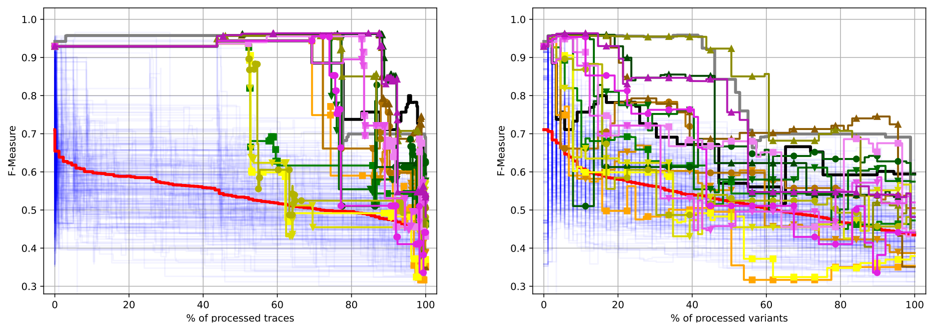
(a) Domestic declarations log [19]



(b) Sepsis log (sampled) [9]

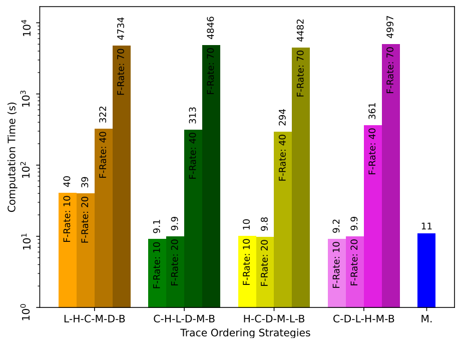


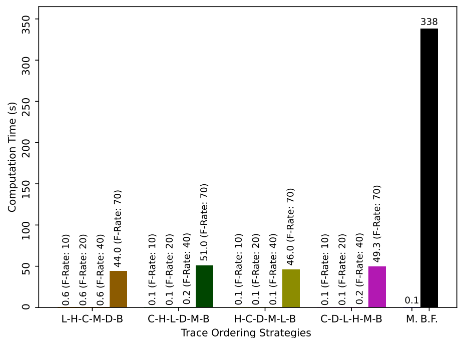
(c) Road traffic fine management log [12]

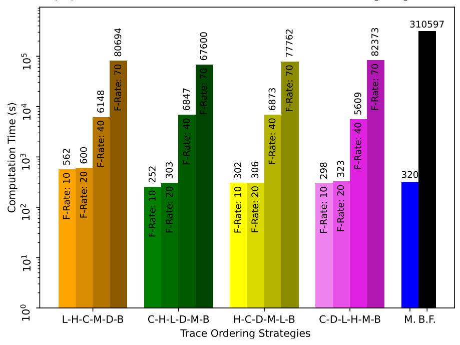


(d) Request for payment log [19]

Fig. 5: F-measure values of the incrementally discovered process models. Most evaluated strategies (cf. Figure 4) perform better than the baseline (red line). Blue lines indicate the solution space (not complete, as not every possible trace ordering can be evaluated due to a large number of trace variants per event log).


(a) Domestic declarations log [19]


(b) Sepsis log [9]


(c) Road traffic fine management log [12]

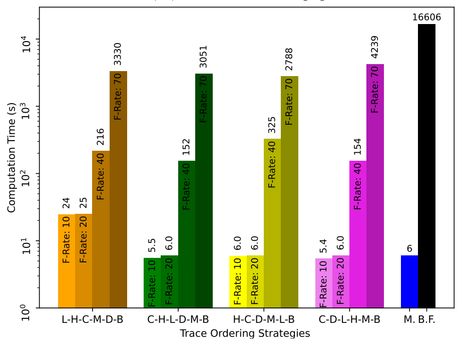

(d) Request for payment log [19]

Fig. 6: Computation time of the strategies (cf. Figure 4) per event log

6 Conclusion

We presented a framework to define trace-ordering strategies for incremental process discovery. We introduced general strategy components and evaluated different strategies on real-life event data based on the framework. The results show that the trace-ordering strategies can improve the quality of the learned process models. For future work, we are interested in non-sequential compositions of strategy components, e.g., each strategy component ranks all trace candidates, and finally, a score is determined. However, this requires more efficient computable strategy components. Finally, we plan to integrate trace-ordering strategies in our incremental process discovery tool Cortado [15].

References

1. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the Eleventh International Conference on Data Engineering. IEEE Comput. Soc. Press (1995). https://doi.org/10.1109/ICDE.1995.380415
2. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs:

Review and benchmark. IEEE Transactions on Knowledge and Data Engineering **31**(4) (2019). https://doi.org/10.1109/TKDE.2018.2841877

3. Carmona, J., van Dongen, B., Solti, A., Weidlich, M.: Conformance Checking. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-99414-7
4. Conforti, R., La Rosa, M., ter Hofstede, A.H.: Filtering out infrequent behavior from business process event logs. IEEE Transactions on Knowledge and Data Engineering **29**(2) (2017). https://doi.org/10.1109/TKDE.2016.2614680
5. Cornuéjols, A.: Getting order independence in incremental learning. In: Machine Learning: ECML-93, Lecture Notes in Computer Science, vol. 667. Springer Berlin Heidelberg (1993). https://doi.org/10.1007/3-540-56602-3_137
6. Dixit, P.M., Buijs, J.C.A.M., van der Aalst, W.M.P.: Prodigy : Human-in-the-loop process discovery. In: 12th International Conference on Research Challenges in Information Science (RCIS). IEEE (2018). https://doi.org/10.1109/RCIS.2018.8406657
7. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer Berlin Heidelberg (2018). https://doi.org/10.1007/978-3-662-56509-4
8. Fahland, D., van der Aalst, W.M.: Model repair — aligning process models to reality. Information Systems **47** (2015). https://doi.org/10.1016/j.is.2013.12.007
9. Felix Mannhardt: Sepsis cases - event log. https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460
10. Ferilli, S., Esposito, F.: A logic framework for incremental learning of process models. Fundamenta Informaticae **128** (2013). https://doi.org/10.3233/FI-2013-951
11. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In: Application and Theory of Petri Nets and Concurrency, vol. 7927. https://doi.org/10.1007/978-3-642-38697-8_17
12. M. (Massimiliano) de Leoni, Felix Mannhardt: Road traffic fine management process. https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5
13. MacGregor, J.N.: The effects of order on learning classifications by example: Heuristics for finding the optimal order. Artificial Intelligence **34**(3) (1988). https://doi.org/10.1016/0004-3702(88)90065-3
14. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Incremental discovery of hierarchical process models. In: Research Challenges in Information Science, Lecture Notes in Business Information Processing, vol. 385. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-50316-1_25
15. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Cortado—an interactive tool for data-driven process discovery and modeling. In: Application and Theory of Petri Nets and Concurrency, Lecture Notes in Computer Science, vol. 12734. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-76983-3_23
16. Schuster, D., van Zelst, S.J., van der Aalst, W.M.: Utilizing domain knowledge in data-driven process discovery: A literature review. Computers in Industry **137** (2022). https://doi.org/10.1016/j.compind.2022.103612
17. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer Berlin Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
18. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. WIREs Data Mining and Knowledge Discovery **2**(2) (2012). https://doi.org/10.1002/widm.1045
19. van Dongen, B.F.: BPI challenge 2020. https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51

View publication stats