Finding Complex Process-Structures by Exploiting the Token-Game
(Regular Paper)

Lisa L. Mannel(✉) and Wil M. P. van der Aalst

Process and Data Science (PADS), RWTH Aachen University, Aachen, Germany,
mannel@pads.rwth-aachen.de; wvdaalst@pads.rwth-aachen.de

Abstract. In process discovery, the goal is to find, for a given event log, the model describing the underlying process. While process models can be represented in a variety of ways, in this paper we focus on the representation by Petri nets. Using an approach inspired by language-based regions, we start with a Petri net without any places, and then insert the maximal set of places considered fitting with respect to the behavior described by the log. Traversing and evaluating the whole set of all possible places is not feasible since their number is exponential in the number of activities. Therefore, we propose a strategy to drastically prune this search space to a small number of candidates, while still ensuring that all fitting places are found. This allows us to derive complex model structures that other discovery algorithms fail to discover. In contrast to traditional region-based approaches this new technique can handle infrequent behavior and therefore also noisy real-life event data. The drastic decrease of computation time achieved by our pruning strategy, as well as our noise handling capability, is demonstrated and evaluated by performing various experiments.

Keywords: Process Discovery, Petri Nets, Language-based Regions

1 Introduction

More and more processes executed in companies are supported by information systems which store each event executed in the context of a so-called *event log*. For each event, such an event log typically describes a name identifying the executed activity, the respective case specifying the execution instance of the process, the time when the event was observed, and often other data related to the activity and/or process instance. An example event log is shown in Figure 1.

In the context of process mining, many algorithms and software tools have been developed to utilize the data contained in event logs: in *conformance checking*, the goal is to determine whether the behaviors given by a process model and event log comply. In *process enhancement*, existing process models are improved. Finally, in *process discovery*, a process model is constructed aiming to reflect the behavior defined by the given event log: the observed events are put into relation to each other, preconditions, choices, concurrency, etc. are discovered, and brought together in a model, e.g. a Petri net.

Case ID	Activity	Time Stamp	Resource
1	▶	01.01.2019	R0
1	A	02.01.2019	R1
2	▶	07.01.2019	R0
1	C	11.03.2019	R2
2	B	01.05.2019	R4
2	C	07.07.2019	R2
2	C	08.07.2019	R2
2	C	11.07.2019	R2
1	C	26.08.2019	R2
1	D	27.09.2019	R3
1	■	29.09.2019	R0
2	E	07.12.2019	R4
2	■	08.12.2019	R0
		...	

Real-life processes usually have a start and end, and therefore it is reasonable to assume a designated start activity ▶ to be executed at the beginning of each process instance, as well as a corresponding end activity ■. For a (simplified) process of package delivery by the postal service, this could, for example, be *package registration* and *confirmation of delivery*. Each delivery process would be corresponding to a case, with possible activities like *attempt delivery* or *relocate package*. Possible resources could be the car used for delivery or the employee processing the package.

Fig. 1. Excerpt of an example event log. The two visible cases correspond to the activity sequences $\langle ▶, A, C, C, D, ■\rangle$ and $\langle ▶, B, C, C, C, E, ■\rangle$.

Process discovery is non-trivial for a variety of reasons. The behavior recorded in an event log cannot be assumed to be complete, since behavior allowed by the process specification might simply not have happened yet. Additionally, real-life event logs often contain rare patterns, either due to infrequent behavior or due to logging errors. Especially the latter type should not be taken into account when discovering the process model, but finding a balance between filtering out noise and at the same time keeping all desired information is often a non-trivial task. Ideally, a discovered model should be able to produce the behavior contained within the event log, not allow for behavior that was not observed, represent all dependencies between the events and at the same time be uncomplicated enough to be understood by a human interpreter. It is rarely possible to fulfill all these requirements simultaneously. Based on the capabilities and focus of the used algorithm, the discovered models can vary greatly. Often, there is no one and only true model, but instead, a trade-off between the aspects noted above has to be found. In Section 2 we give an overview of work related to our paper. For a detailed introduction to process mining we refer the interested reader to [1].

In this paper we suggest an algorithm inspired by language-based regions, that guarantees to find a model defining the minimal language containing the input language ([2]). Due to our assumptions, the usually infinite set of all possible places is finite. In contrast to prominent discovery approaches based on language-based regions (see Section 2), we do not use integer linear programming to find the subset of fitting places. Instead, we replay the event log on candidate places to evaluate whether they are fitting. We achieve that by playing the token game for each trace in the log, and then utilizing the results to skip uninteresting sections of the search space as suggested in [3]. In contrast to the brute-force approach evaluating every single candidate place, our technique drastically increases the efficiency of candidate evaluation by combining this skipping of candidates with a smart candidate traversal strategy, while still providing the same guarantees. Additionally, our algorithm lends itself to apply efficient noise-filtering, as well as other user-definable constraints on the set of fitting places. As a final step, we suggest to post-process the discovered set of fitting places, thereby reducing the complexity of the resulting model and making it interpretable by humans. Altogether, our approach has the potential to combine the capability of discovering complex model structures, typical for region-based approaches, with the ability to handle noise and simplify the model according to user-definable constraints. We illustrate the capabilities of our algorithm by providing results of measuring its decrease in computation time compared to the brute-force approach, testing its noise-handling abilities, and illustrating the rediscovery of complex models.

An overview of related work is given in the next section. In the remainder of this paper we provide a detailed description, formalization and discussion of our discovery approach. In Section 3 basic notations and definitions are given. We present a detailed motivation and overview of our approach in Section 4. Section 5 provides an extensive explanation and formalization. In Section 6, we briefly discuss our implementation, including some tweaks and optimizations that can be used to further improve our approach. A comparison to existing discovery algorithms is given in Section 7 together with results and evaluation of testing. Finally, we conclude the paper with a summary and suggestion of future work in Section 8.

2 Related Work

Process discovery algorithms make use of a variety of formal and informal representations to model the behavior they extract from a given event log. However, the basic idea is similar: based on the event data given by an event log, the different event types are coordinated and ordered using some kind of connection between them. In this paper, we focus on the formal representation by Petri nets, where the event types correspond to transitions and the coordinating connections correspond to places. However, our ideas can be adapted to other representations as well. Discovery algorithms that produce a formal process model can provide desirable formal guarantees for their discovered models, for example, the ability to replay each sequence of events contained in the log, or the ability to re-discover a model based on a sufficient description of its behavior.

As noted above in process discovery there are several, often conflicting quality criteria for a discovered model. To decrease computation time and the complexity of the found Petri net, many existing discovery algorithms further abstract from a given log to another representation, containing only a fraction of the original information, based on which a formal model is created. These algorithms can rediscover only subclasses of Petri nets, and often the resulting model does not allow for the log to be fully replayed, or allows for much more than the log suggests. Examples are the Alpha Miner variants ([4]) and the Inductive Mining

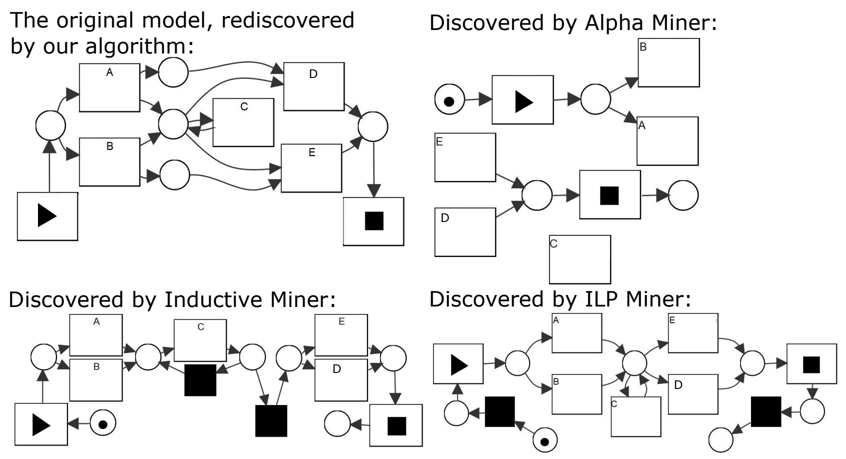
The original model, rediscovered by our algorithm:
Discovered by Alpha Miner:
Discovered by Inductive Miner:
Discovered by ILP Miner:
A
B
C
D
E

Fig. 2. There exists a variety of discovery algorithms that are able to mine a process model based on the log $[\langle \blacktriangleright, A, C, C, D\blacksquare\rangle, \langle \blacktriangleright, B, C, C, C, E\blacksquare\rangle]$ from Figure 1. As illustrated, the resulting Petri nets differ significantly between the algorithms. In particular, the places connecting A to D and B to E, which ensure that the first choice implies the second choice, are rarely discovered by existing algorithms.

family ([5]). Other miners based on heuristics, like genetic algorithms or Heuristic Miner ([6]) cannot provide guarantees at all.

Due to omitting part of the information contained in the log, the miners described above are not able to discover complex model structures, most prominently non-free choice constructs. The task of creating a Petri net that precisely corresponds to a given description of its behavior is known as the synthesis problem and closely related to region theory ([7]). Traditionally, this description was given in form of a transition system, which was then transformed into a Petri net using state-based region theory ([8]). The approach has been adapted for process discovery by developing algorithms, e.g. FSM Miner, that generate a transition system based on the log, which is then transformed into a Petri net ([9,10]). Other approaches use language-based region theory, where the given description is a language, rather than a transition system ([11–13]). An event log can be directly interpreted as a language. Therefore, language-based regions can be applied directly to synthesize a Petri net from a given event log ([2]). Here the basic strategy is to start with a Petri net that has one transition for each activity type contained in the log. Then all places that allow replaying the log are added. The result is a Petri net, that defines a language which is a minimal superset of the input language. Currently such algorithms, most prominently ILP Miner, are based on integer linear programming ([14–16]). However, available implementations make use of an abstraction of the log to increase performance, thus losing their ability to find all possible places.

In contrast to most other discovery algorithms, region-based approaches guarantee that the model can replay the complete event log, and at the same time does not allow for much different behavior. In particular, complex model structures like non-free choice constructs are reliably discovered. On the downside, region-based discovery algorithms often lead to complex process models that are impossible to understand for the typically human interpreter. They are also known to expose severe issues with respect to low-frequent behavior often contained in real-life event logs. Finally, finding all the fitting places out of all possible places tends to be extremely time-consuming.

To illustrate the differences between existing discovery algorithms, in Figure 2 we show the results of selected discovery algorithms applied to the log shown in Figure 1. The original model, that produced the log, can be rediscovered by the approach suggested in this paper. Alpha Miner, Inductive Miner and ILP Miner cannot discover this model because they are restricted to mine only for certain structures and/or a subset of possible places. In particular, the implication of the second choice (C orD) by the first choice (A or B) is not discovered by any of these algorithms.

3 Basic Notation, Event Logs and Process Models

Throughout our paper we will use the following notations and definitions: A set, e.g. $\{a, b, c\}$, does not contain any element more than once, while a multiset, e.g. $[a, a, b, a] = [a^3, b]$, may contain multiples of the same element. By $\mathbb{P}(X)$ we refer to the power set of the set X, and $\mathbb{M}(X)$ is the set of all multisets over this set. In contrast to sets and multisets, where the order of elements is irrelevant, in sequences the elements are given in a certain order, e.g. $\langle a, b, a, b\rangle \neq \langle a, a, b, b\rangle$. We refer to the i'th element of a sequence σ by $\sigma(i)$. The size of a set, multiset or sequence X, that is $|X|$, is defined to be the number of elements in X.

We define activities, traces, and logs as usual, except that we require each trace to begin with a designated start activity and end with a designated end activity. Since process models, in general, have a start and end, this is a reasonable assumption. It implies, that in any discovered model all places are intermediate places that are not part of an initial or final marking. Thus, every candidate place we consider during execution of our algorithm, does not contain any initial tokens. This greatly simplifies the presentation of our work. Note, that any log can easily be transformed accordingly.

Definition 1 (Activity, Trace, Log). *Let $\mathcal{A}$ be the universe of all possible* activities *(actions, events, operations, ...), let $\blacktriangleright \in \mathcal{A}$ be a designated start activity and let $\blacksquare \in \mathcal{A}$ be a designated end activity. A* trace *is a sequence containing $\blacktriangleright$ as the first element, $\blacksquare$ as the last element and in-between elements of $\mathcal{A} \setminus \{\blacktriangleright, \blacksquare\}$. Let $\mathcal{T}$ be the set of all such traces. A* log *$L \subseteq \mathbb{M}(\mathcal{T})$ is a multiset of traces.*

In the following definition of Petri nets, note that we require the set of transitions to correspond to a subset of the universe of activities. Therefore our Petri nets are free of silent or duplicate transitions. In combination with not having to deal with markings, this results in a finite set of candidate places. Also note, that we do not use weighted arcs, and can therefore assume the arcs to be implicitly

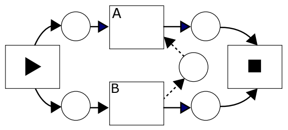

Fig. 3. Example of a Petri net $N = (A, \mathcal{P})$ with transitions $A = \{a, b, \blacktriangleright, \blacksquare\}$ and places $\mathcal{P} = \{(\{\blacktriangleright\}|\{a\}), (\{\blacktriangleright\}|\{b\}), (\{a\}|\{\blacksquare\}), (\{b\}|\{\blacksquare\})\}$. The behavior of N is the set of fitting traces $\{\langle \blacktriangleright, a, b, \blacksquare \rangle, \langle \blacktriangleright, b, a, \blacksquare \rangle\}$. A possible place $(\{b\}|\{a\})$ is underfed with respect to the trace $\langle \blacktriangleright, a, b, \blacksquare \rangle$.

defined by the sets of ingoing and outgoing transitions of the places. This definition of a subset of Petri nets, natural with respect to our discovery algorithm, removes a lot of notational overhead.

Definition 2 (Petri nets). *A Petri net is a pair $N = (A, \mathcal{P})$, where $A \subseteq \mathcal{A}$ is the set of transitions, and $\mathcal{P} \subseteq \{(I|O) \mid I \subseteq A \land I \neq \emptyset \land O \subseteq A \land O \neq \emptyset\}$ is the set of places. We call I the set of* ingoing *activities of a place and O the set of* outgoing *activities.*

Places, that are not able to perfectly replay the given log, can be unfitting in two ways: If at some point during replay there is no token at the place, but the log requires the consumption of a token anyway, we call the place *underfed*. If at the end of a replayed trace there is at least one token left at the place, we call the place *overfed*. This categorization has been extensively discussed in [3] and is the key to our efficient candidate traversal: as detailed in Section 4, by evaluating one place to be underfed (overfed) we can determine a whole set of places to be underfed (overfed), without even looking at them.

Definition 3 (Overfed/Underfed/Fitting Places, see [3]). *Let $N = (A, \mathcal{P})$ be a Petri net, let $p = (I|O) \in \mathcal{P}$ be a place, and let σ be a trace. With respect to the given trace σ, p is called*

- underfed*, denoted by $\triangledown_\sigma(p)$, if and only if $\exists k \in \{1, 2, ..., |\sigma|\}$ such that $|\{i \mid i \in \{1, 2, ...k-1\} \land \sigma(i) \in I\}| < |\{i \mid i \in \{1, 2, ...k\} \land \sigma(i) \in O\}|$,*
- overfed*, denoted by $\triangle_\sigma(p)$, if and only if $|\{i \mid i \in \{1, 2, ...|\sigma|\} \land \sigma(i) \in I\}| > |\{i \mid i \in \{1, 2, ...|\sigma|\} \land \sigma(i) \in O\}|$,*
- fitting*, denoted by $\square_\sigma(p)$, if and only if not $\triangledown_\sigma(p)$ and not $\triangle_\sigma(p)$.*

Note, that a place can be underfed and overfed at the same time.

Definition 4 (Behavior of a Petri net). *We define the* behavior *of the Petri net $(A, \mathcal{P})$ to be the set of all fitting traces, that is $\{\sigma \in \mathcal{T} \mid \forall p \in \mathcal{P} \colon \square_\sigma(p)\}$.*

4 Algorithmic Framework

As input our algorithm takes a log L, that can be directly interpreted as a language, and a parameter $\tau \in [0, 1]$. This parameter τ in principle determines the fraction of the log that needs to be replayable by a fitting place, and is essential for our noise handling strategy. We provide details on this threshold at the end of this section.

Inspired by language-based regions, the basic strategy of our approach is to begin with a Petri net, whose transitions correspond to the activity types used in the given log. From the finite set of unmarked, intermediate places $\mathcal{P}_{\text{all}}$ we want to select a subset $\mathcal{P}_{\text{final}}$, such that the language defined by the resulting net defines the minimal language containing the input language, while, for human readability, using only a minimal number of places to do so. Note, that by filtering the log for noise the definition of the desired language becomes less rigorous, since the allegedly noisy parts of the input language are ignored, and thus the aforementioned property can no longer be guaranteed.

We achieve this aim by applying several steps detailed in Section 5. First of all, we observe that the set $\mathcal{P}_{\text{all}}$ contains several places that can never be part of a solution, independently of the given log. By ignoring these places we reduce our set of candidates to $\mathcal{P}_{\text{cand}} \subseteq \mathcal{P}_{\text{all}}$. Next, we apply the main step of our approach: while utilizing the token-game to skip large parts of the candidate space, we actually evaluate only a subset of candidates, $\mathcal{P}_{\text{visited}} \subseteq \mathcal{P}_{\text{cand}}$. We can guarantee that the set of fitting places is a subset of these evaluated places, that is $\mathcal{P}_{\text{fit}} \subseteq \mathcal{P}_{\text{visited}}$. Finally, aiming to achieve a model that is interpretable by human beings, we reduce this set of fitting places to a set of final places $\mathcal{P}_{\text{final}} \subseteq \mathcal{P}_{\text{fit}}$ by removing superfluous places.

The main challenge of our approach lies in the size of the candidate space: there are $|\mathcal{P}_{\text{all}}| = |\mathbb{P}(A)\backslash\emptyset \times \mathbb{P}(A)\backslash\emptyset| \approx \ (2^{|A|})^2$ possible places to be considered. Keeping all of them in memory and even more replaying the log for this exponentially large number of candidates will quickly become infeasible, even for comparably small numbers of activities. Reducing the set $\mathcal{P}_{\text{all}}$ to $\mathcal{P}_{\text{cand}}$ is by far not a sufficient improvement.

Towards a solution to this performance issue, we propose an idea allowing us to drastically reduce the amount of traversed candidate places, while still providing the complete set $\mathcal{P}_{\text{fit}}$ as outcome. The monotonicity results on Petri net places introduced in [3] form the basis of our approach. Intuitively, if a candidate place $p_1 = (I_1|O_1)$ is underfed with respect to some trace σ, then at some point during the replay of σ there are not enough tokens in p_1. By adding another outgoing arc to p connecting it to some transition $a \notin O_1$ we certainly do not increase the number of tokens in the place and therefore the resulting place $p_2 = (I_1|O_1 \cup \{a\})$ must be underfed as well. Thus, by evaluating p_1 to be underfed we can infer that all candidates $(I_1|O_2)$ with $O_1 \subseteq O_2$ are underfed as well, without having to evaluate them. A similar reasoning can be applied to overfed places. This is formalized in Lemma 1. For more details, we refer the reader to the original paper ([3]).

Lemma 1 (Monotonicity Results (see [3])). *Let $p_1 = (I_1|O_1)$ be a place and let σ be a trace. If $\triangledown_\sigma(p_1)$, then $\triangledown_\sigma(p_2)$ for all $p_2 = (I_2|O_2)$ with $I_1 \supseteq I_2$ and $O_1 \subseteq O_2$. If $\triangle_\sigma(p_1)$, then $\triangle_\sigma(p_2)$ for all $p_2 = (I_2|O_2)$ with $I_1 \subseteq I_2$ and $O_1 \supseteq O_2$.*

As detailed in [3], these monotonicity results allow us to determine a whole set of places to be unfitting by evaluating a single unfitting candidate. Combining this idea with the candidate traversal strategy presented in Section 5 allows us to skip most unfitting places when traversing the candidate space, without missing any other, possibly interesting place. It is important to note that we do not simply skip the replay of the log, but actually do not traverse these places at all. This greatly increases the performance of our algorithm.

Setting our algorithm apart from other region-based approaches is its ability to directly integrate noise filtering. When evaluating a visited place, we refer to a user-definable parameter τ as detailed in the following definition:

Definition 5 (Fitness with Respect to a Threshold). *With respect to a given log L and threshold τ, we consider a place $p = (I|O)$ to be*

- *fitting, that is* $\square_L^\tau(p)$, *if and only if* $\frac{|\{\sigma \in L \mid \square_\sigma(p) \wedge \sigma \cap (I \cup O) \neq \emptyset\}|}{|\{\sigma \in L \mid \exists a \in \sigma \colon a \in (I \cup O)\}|} \geq \tau$,
- *underfed, that is* $\triangledown_L^\tau(p)$, *if and only if* $\frac{|\{\sigma \in L \mid \triangledown_\sigma(p)\}|}{|\{\sigma \in L \mid \exists a \in \sigma \colon a \in (I \cup O)\}|} > (1 - \tau)$,
- *overfed, that is* $\vartriangle_L^\tau(p)$, *if and only if* $\frac{|\{\sigma \in L \mid \vartriangle_\sigma(p)\}|}{|\{\sigma \in L \mid \exists a \in \sigma \colon a \in (I \cup O)\}|} > (1 - \tau)$.

Intuitively, a place p is fitting/underfed/overfed/ with respect to L and τ, if it is underfed/overfed/fitting with respect to a certain fraction of traces in L that involve the activities of p. This fraction is determined by the threshold τ. By defining the value of τ, the user of our algorithm can choose to ignore a fraction of traces when evaluating the places, making the result much more robust with respect to infrequent behavior. If $\tau = 1$, then all places are required to be perfectly fitting. In [3] it is shown that Lemma 1 can be extended to the use of such a threshold. Despite our slightly modified definition, their proof remains valid. The impact of different values of τ will be investigated in Section 7.

5 Computing a Desirable Subset of Places

As input, our algorithm expects a log L, and a user-definable parameter $\tau \in [0, 1]$. The activities contained in L, $A \subseteq \mathcal{A}$, define the set of transitions of the Petri net we want to discover. These define a finite set of unmarked, intermediate places $\mathcal{P}_{\text{all}}$, that we could insert, as the starting point of our algorithm.

5.1 Pre-pruning of Useless Places

Within the set $\mathcal{P}_{\text{all}}$, there are many candidates that are guaranteed to be unfitting, independently of the given log. These are all places $(I|O)$, with $\blacktriangleright \in O$ or with $\blacksquare \in I$ for designated start and end activities $\blacktriangleright$, $\blacksquare$. By completely excluding such places from our search space, we remain with a much smaller set of candidates $\mathcal{P}_{\text{cand}} \subseteq \mathcal{P}_{\text{all}}$: For a set of activities A the number of candidates is bounded by $2^{|A|} \times 2^{|A|}$. By removing all places with $\blacktriangleright \in O$ or $\blacksquare \in I$, we effectively decrease the size of the activity set A by one for each, incoming and outgoing activities. The new bound on the number of candidates is $2^{|A|} \times 2^{|A|}$, thus reducing its size by 25%.

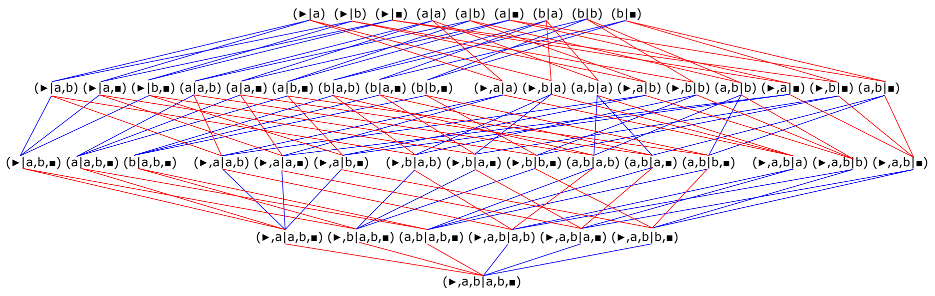
(▶|a) (▶|b) (▶|■) (a|a) (a|b) (a|■) (b|a) (b|b) (b|■)
(▶|a,b) (▶|a,■) (▶|b,■) (a|a,b) (a|a,■) (a|b,■) (b|a,b) (b|a,■) (b|b,■) (▶,a|a) (▶,b|a) (a,b|a) (▶,a|b) (▶,b|b) (a,b|b) (▶,a|■) (▶,b|■) (a,b|■)
(▶|a,b,■) (a|a,b,■) (b|a,b,■) (▶,a|a,b) (▶,a|a,■) (▶,a|b,■) (▶,b|a,b) (▶,b|a,■) (▶,b|b,■) (a,b|a,b) (a,b|a,■) (a,b|b,■) (▶,a,b|a) (▶,a,b|b) (▶,a,b|■)
(▶,a|a,b,■) (▶,b|a,b,■) (a,b|a,b,■) (▶,a,b|a,b) (▶,a,b|a,■) (▶,a,b|b,■)
(▶,a,b|a,b,■)

Fig. 4. Example of the candidate space based on four activities $\blacktriangleright, a, b$ and $\blacksquare$. The horizontal levels and colored edges indicate the relations between the activity sets: each candidate place $(I_1|O_1)$ is connected to a place $(I_2|O_2)$ from the level above by a blue line if $I_1 = I_2$ and $O_1 \supsetneq O_2$, and by a red line if $I_1 \supsetneq I_2$ and $O_1 = O_2$.

5.2 Developing an Efficient Candidate Traversal Strategy

We illustrate our idea with the help of a running example, where $A = \{\blacktriangleright, a, b, \blacksquare\}$ is the given set of activities. The corresponding pre-pruned candidate space is visualized in Figure 4. The organization and coloring are chosen to clarify the relations between the candidates, which we are going to utilize to further prune the candidate space.

Our strategy for traversing the set of candidates is the key to the effective utilization of the described monotonicity results. Since we cannot efficiently store and retrieve an exponential amount of candidate places, we need a scheme to deterministically compute the next place we want to evaluate based on a limited set of information we keep in storage. This scheme should at the same time guarantee that all fitting places are visited, each place is visited only a limited number of times (preferably at most once), and we are able to prune the search space by employing the results obtained by previous place evaluations. In the following, we are going to develop such a candidate traversal scheme.

5.3 Organization of the Candidate Space as Trees

We organize the candidate space in a set of trees, as described in Definition 6. An example is shown in Figure 5. Note, that there are many ways to organize the candidates in such a structure and this example shows merely one of these possibilities.

Let $A \subseteq \mathcal{A}$ be the given set of activities and let $>_i$ and $>_o$ be two total orderings on A. In the remainder, we assume all sets of ingoing activities of a place to be ordered lexicographically according to $>_i$, and sets of outgoing activities according to $>_o$. Possible strategies of computing such orderings are noted in Section 6.

Definition 6 (Complete Candidate Tree). *A* complete candidate tree *is a pair* $CT = (N, F)$ *with* $N = \{(I|O) \mid I \subseteq A \backslash \{\blacksquare\}, O \subseteq A \backslash \{\blacktriangleright\}, I \neq \emptyset, O \neq \emptyset\}$.

We have that $F = F_{red} \cup F_{blue}$, with

$$\begin{aligned} F_{red} =& \{((I_1|O_1),(I_2|O_2)) \in N \times N \mid |O_2| = 1, O_1 = O_2, \\ & \exists a \in I_1 \colon \big(I_2 \cup \{a\} = I_1 \wedge \forall a' \in I_2 \colon a' <_i a\big)\} \;\; \textit{(\textbf{red edges})} \\ F_{blue} =& \{((I_1|O_1),(I_2|O_2)) \in N \times N \mid I_1 = I_2, \\ & \exists a \in O_1 \colon \big(O_2 \cup \{a\} = O_1 \wedge \forall a' \in O_2 \colon a' <_o a\big)\} \;\; \textit{(\textbf{blue edges})}. \end{aligned}$$

If $((I_1|O_1),(I_2|O_2)) \in F$, we call the candidate $(I_1|O_1)$ the child *of its* parent *$(I_2|O_2)$. A candidate $(I|O)$ with $|I| = |O| = 1$ is called a* base root.

Note, that there is a certain asymmetry in the definition of F_{red} and F_{blue}: red edges connect only places which have exactly one outgoing transition, while for blue edges the number of outgoing transitions is unlimited. This is necessary to obtain the collection of trees we are aiming for, and which is further investigated below: if we did not restrict one of the edge types in this way, the resulting structure would contain cycles. If we restricted both types of edges, many candidates would not be connected to a base root at all. However, the choice of restricted edge type, red or blue, is arbitrary.

In the following we show that each candidate is organized into exactly one tree, that can be identified by its unique base root. Therefore, the number of connected trees contained in one structure CT as described in Definition 6 is exactly the number of base roots. In our running example (Figure 5) there are 9 such trees.

Lemma 2. *The structure CT described by Definition 6 organizes the candidate space into a set of trees rooted in the base roots, where every candidate is connected to exactly one base root.*

Proof (Lemma 2). Let $CT = (N, F)$ be the structure defined in Definition 6. We show that every candidate $(I|O) \in N$ has a unique parent, and, by induction on the number of activities of a candidate, that each candidate is the descendant of exactly one of the base roots. This implies that there are no cycles and the structure is indeed a set of connected trees rooted in the base roots.

If $|I \cup O| = 2$ then p is a base root that cannot have any parents and the claim holds. Now assume that the claim holds for all candidates with $|I \cup O| = n$.

Consider a candidate $p_1 = (I_1|O_1)$ with $|I_1 \cup O_1| = n + 1$. Let $p_2 = (I_2|O_2)$ be any potential parent of of p_1. We distinguish two cases:

Case $|O_1| = 1$: This implies $I_1 = \{a_1, a_2, ..., a_{n-1}, a_n\}$. We have that $(p_1, p_2) \notin F_{\text{blue}}$, because, otherwise, we would have $O_2 = \emptyset$. If $(p_1, p_2) \in F_{\text{red}}$, then by definition $O_1 = O_2$ and $I_1 = I_1 \backslash \{a_n\}$.

Case $|O_1| \geq 2$: This implies $O_1 = \{a_1, a_2, ..., a_{k-1}, a_k\}$, for some $k \in [2, n-1]$. We have that $(p_1, p_2) \notin F_{\text{red}}$, because red edges require $|O_1| = 1$.If $(p_1, p_2) \in F_{\text{blue}}$, then by definition we have that $I_1 = I_2$ and $O_2 = O_1 \backslash \{a_k\}$.

In both cases, p_2 is fully defined based on p_1 and therefore the unique parent. Since $|I_2 \cup O_2| = n$, the claim holds for p_2 and thus for p_1 as well. By induction, the claim holds for all candidates in N. □

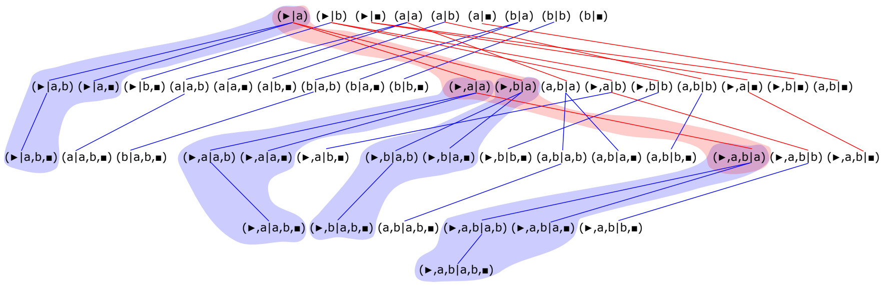
(▸|a) (▸|b) (▸|■) (a|a) (a|b) (a|■) (b|a) (b|b) (b|■)
(▸|a,b) (▸|a,■) (▸|b,■) (a|a,b) (a|a,■) (a|b,■) (b|a,b) (b|a,■) (b|b,■) (▸,a|a) (▸,b|a) (a,b|a) (▸,a|b) (▸,b|b) (a,b|b) (▸,a|■) (▸,b|■) (a,b|■)
(▸|a,b,■) (a|a,b,■) (b|a,b,■) (▸,a|a,b) (▸,a|a,■) (▸,a|b,■) (▸,b|a,b) (▸,b|a,■) (▸,b|b,■) (a,b|a,b) (a,b|a,■) (a,b|b,■) (▸,a,b|a) (▸,a,b|b) (▸,a,b|■)
(▸,a|a,b,■) (▸,b|a,b,■) (a,b|a,b,■) (▸,a,b|a,b) (▸,a,b|a,■) (▸,a,b|b,■)
(▸,a,b|a,b,■)

Fig. 5. Example of a complete candidate tree based on $A = \{\blacktriangleright, a, b, \blacksquare\}$. In reference to Lemma 3, an example subtree of red edges is marked by a red background, while all subtrees of blue edges attached to it are marked by a blue background. Together they form the whole tree rooted in the base root $(\{\blacktriangleright\}|\{a\})$.

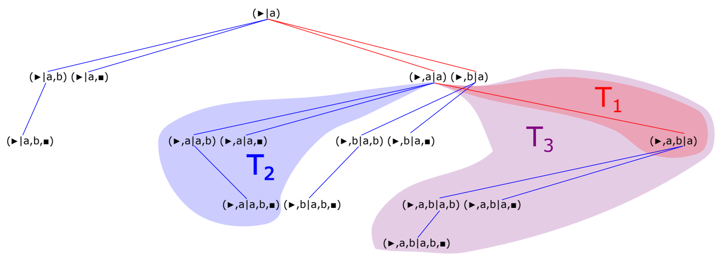

Fig. 6. We illustrate our tree cutting strategy using the tree rooted in the base root $(\{\blacktriangleright\}|\{a\})$ from our running example (Figure 5). For the example place $(\{\blacktriangleright, a\}|\{a\})$ we highlight the blue-edged subtree (T_2) and red-edged subtree (T_1) referred to in Lemma 3. In reference to Lemma 4 we indicate the subtrees attached by blue edges, which coincides with the blue-edged subtree (T_2), and the subtrees attached by red edges (T_3). We emphasize that $T_1 \neq T_3$.

5.4 Tree Traversal and Pruning of Candidate Space

Definition 7 (Tree Traversal Strategy). *Let L be a log, τ threshold, and A the set of activities contained in L. Let $CT = (N, F)$ be the complete candidate tree based on A, with $N = \mathcal{P}_{cand}$. We traverse one connected tree after the other, by using a depth-first strategy starting on each base root of CT.*

Let $p = (I|O)$ be a visited place. If p is fitting, i.e. $\square^{\tau}_{L}(p)$, we add it to $\mathcal{P}_{fit}$. If p is underfed $(\triangledown^{\tau}_{L}(p))$, we skip all subtrees attached to p by a blue edge. If p is overfed $(\triangle^{\tau}_{L}(p))$ and $O = \{a\}$ with $\nexists a' \in A\colon a' >_o a$, we skip all subtrees attached to p by a red edge. After skipping a subtree this way, we proceed with our traversal as if we had already visited those places.

Formalized in Theorem 1, we show that this strategy allows us to skip sets of unfitting places and at the same time guarantee, that no place in $\mathcal{P}_{\text{fit}}$ is missed.

Lemma 3. *Let L, τ be a given log and threshold, $CT = (N, F)$ a complete candidate tree and $p_1 \in N$ any candidate. If $\triangledown^{\tau}_{L}(p_1)$ $(\triangle^{\tau}_{L}(p_1))$, then for all* *candidates $p_2 \in N$ within the blue-edged (red-edged) subtree rooted in p_1, we have that $\triangledown^{\tau}_{L}(p_2)$ $(\triangle^{\tau}_{L}(p_2))$.*

Proof (Lemma 3). Let $p_1 = (I_1|O_1) \in N$ be a candidate with $\triangledown^{\tau}_{L}(p_1)$ $(\triangle^{\tau}_{L}(p_1))$. Consider any descendant $p_2 = (I_2|O_2)$ attached to p_1 via a path of blue (red) edges. From the definition of blue (red) edges (see Definition 6) we infer that $I_1 = I_2$ and $O_1 \subsetneq O_2$ $(O_1 = O_2$ and $I_1 \subsetneq I_2)$. Then from the monotonicity results presented Lemma 1 it follows that $\triangledown^{\tau}_{L}(p_2)$ $(\triangle^{\tau}_{L}(p_2))$. □

According to Lemma 3, if a place p is underfed, then blue-edged subtree rooted in p contains only underfed places, and if p is overfed, then the red-edged subtree rooted in p contains only overfed places. In the following, we try to extend these results to subtrees rooted in p in general. We will show that for subtrees attached to p by blue edges this is possible in a straight-forward way. Unfortunately, due to the asymmetry of the definition of the edges, the same argument does not always apply to subtrees attached by red edges: here it is entirely possible that there are blue edges within the corresponding subtree, and "overfedness" of p cannot be necessarily extended to the contained places. This is illustrated in Figure 6.

Lemma 4. *Let L, τ be a given log and threshold, $CT = (N, F)$ a complete candidate tree and $p_1 = (I_1|O_1) \in N$ any candidate. If $\triangledown^{\tau}_{L}(p_1)$, then for all candidates p_2 within the subtrees attached to p_1 with a blue edge we have $\triangledown^{\tau}_{L}(p_2)$. If $\triangle^{\tau}_{L}(p_1)$ and $O_1 = \{a\}$ with $\nexists a' \in A\colon a' >_o a$, then for all candidates p_2 within the subtrees attached to p_1 we have $\triangle^{\tau}_{L}(p_2)$.*

Proof (Lemma 4). Assume $\triangledown^{\tau}_{L}(p_1)$. Due to the definition of blue edges, it holds for each child $p' = (I'|O')$ in a subtree attached to p_1 by such an edge, that $|O'| > |O_1|$, and thus in particular $|O'| \geq 2$. By definition, all descendants of p' have at least two outgoing activities. This implies that there is no red edge in the whole subtree, since they require by definition at most one outgoing activity. Thus, by Lemma 3, for every place p_2 in a subtree rooted in such a p' we have $\triangledown^{\tau}_{L}(p_2)$, and the first claim holds.

Now assume $\triangle^{\tau}_{L}(p_1)$ and $O_1 = \{a\}$ with $\nexists a' \in A\colon a' >_o a$. Due to the definition of red edges, it holds for p_1 and each descendant $p' = (I'|O')$ of p_1 reachable by red edges, that $O' = \{a\}$. This implies that there is no blue edge in the whole subtree rooted in p_1, since they require the existence of an activity a' with $a <_o a'$. Thus, by Lemma 3, for every place p_2 in the subtree rooted in p_1, we have $\triangle^{\tau}_{L}(p_2)$. □

Theorem 1. *Given a log L, a threshold τ and the complete candidate tree $CT = (N, F)$ with $N = \mathcal{P}_{cand}$, let $\mathcal{P}_{fit}$ be the set of fitting places with respect to L and τ, that is $\{p \in N \mid \square^{\tau}_{L}(p)\}$. Let $\mathcal{P}_{visited}$ be the set of places visited by our tree traversal strategy described in Definition 7. It holds that $\mathcal{P}_{fit} \subseteq \mathcal{P}_{visited} \subseteq \mathcal{P}_{cand}$.*

Proof (Theorem 1). As proven in Lemma 2 every candidate is contained in exactly one tree rooted in a base root and is thus visited exactly once by standard depth-first search. Thus $\mathcal{P}_{\text{visited}} \subseteq \mathcal{P}_{\text{cand}}$.

Using depth-first search for tree traversal guarantees that for any visited candidate p, we visit the complete subtree rooted in p, before proceeding to another subtree. Thus skipping a subtree does not influence the traversal of other subtrees. If p is underfed/overfed, we can apply Lemma 4 to guarantee that all places contained in the skipped subtrees are underfed/overfed as well. Thus no fitting places are skipped, and we have that $\mathcal{P}_{\text{fit}} \subseteq \mathcal{P}_{\text{visited}} \subseteq \mathcal{P}_{\text{cand}}$. $\square$

As mentioned earlier, we cannot store the complete candidate tree due to its exponential size, and thus the challenge of the tree traversal lies in the deterministic computation of the next candidate based on limited information only. In our algorithm, this information is only the last candidate and its fitness status. We define a total ordering on the set of places easily computable based on the given orderings $>_i, >_o$. We can use this ordering to deterministically select the next subtree to traverse based on the current candidate. The fitness status is used to decide whether to actually traverse the selected subtree, or skip it and select the next one.

While the theoretical worst-case scenario still requires traversing the full candidate space, we have achieved a drastic increase in performance in practical applications. This is presented in detail in Section 7.

5.5 Evaluation of Potentially Fitting Candidates

For each place visited during candidate traversal, we need to determine its fitness status. Fitting places are added to the set of fitting places $\mathcal{P}_{\text{fit}}$, which will be the input for the post-processing step (see Section 5.6). Overfed and underfed places are not added, instead this fitness status can be used in the context of candidate traversal to skip sets of unfitting places.

To determine the fitness status of a place $p = (I|O)$, we use token-based replay. We replay every trace $\sigma \in L$ on the place p: for each activity $a \in \sigma$, from first to last, if $a \in O$ we decrement the number of tokens at the place by one. Then, if $a \in I$ we increment the number of tokens by one. If the number of tokens becomes negative we consider the place to be underfed with respect to this trace, that is $\triangledown_\sigma(p)$. Otherwise, if the trace has been fully replayed and the number of tokens is larger than zero, we consider the place to be overfed, that is $\triangle_\sigma(p)$. Note that the place can be underfed and overfed at the same time. Based on the replay results of the single traces and the user-definable threshold τ (see Definition 5), we evaluate the fitness status of the place with respect to the whole log.

5.6 Post-Processing

In the previous step, a subset of places $\mathcal{P}_{\text{fit}}$ was computed. These are exactly the places considered to be fitting with respect to the given log L and threshold τ. However, many of them can be removed without changing the behavior of the Petri net implicitly defined by $\mathcal{P}_{\text{fit}}$. Since the process model we return is likely to be interpreted by a human being, such places are undesirable. These places are called *implicit* or sometimes *redundant* and have been studied extensively in the context of Petri nets ([17–21]). In the post-processing step, we find and remove those undesirable places by solving an Integer Linear Programming Problem as suggested for example in [21]. The resulting set of places $\mathcal{P}_{\text{final}} \subseteq \mathcal{P}_{\text{fit}}$ is finally inserted into the Petri net that forms the output of our algorithm.

6 Implementation

We implemented our algorithm as a plug-in for ProM ([22]) named `eST-Miner` using Java. There are many ways in which our idea of organizing the candidate space into a tree structure can be optimized for example with respect to certain types of models, logs or towards a certain goal. Other ideas on how to improve performance can be found in [3].

In our implementation, we investigated the impact of the orderings of the ingoing and outgoing activity sets ($>_i, >_o$ in Section 5) on the fraction of cut off places. They determine the position of candidates within our tree structure. If these orderings are such that underfed/overfed places are positioned close to the root, this leads to large subtrees and thus many places being cut off due to monotonicity results. In Section 7, we present first results of testing different activity orderings and evaluate their impact.

7 Testing Results and Evaluation

In this section, we present the results of testing our algorithm on various data sets. We use a selection of artificial log-model pairs to demonstrate our ability to rediscover complex structures and deal with noise. The efficiency of our search space pruning technique and the resulting increase in speed are evaluated using artificial logs as well as real-life logs. An overview of these logs is given in Table 1.

Rediscoverability of models: In Figure 7, a simple model is shown, which includes non-free choice constructs: the places $(\{a\}|\{e\})$ and $(\{b\}|\{f\})$ are not part of any directly follows relation, and are therefore not discovered by most existing algorithms that provide formal guarantees. Well-known discovery techniques like Alpha-Miner variants ([4]) or the Inductive Mining family ([5]) fail at such tasks. Attempts to extend the capabilities of Alpha Miner to discover such places ([4]) have been only partially successful. These approaches may result in process models that cannot replay the event log. In some cases, the model may have no fitting traces due to livelocks or deadlocks. More complex structures involving non-free choice constructs, like the model depicted in Figure 8, are difficult to mine and not rediscovered by most algorithms ([4]).

In contrast to existing non-region-based algorithms, our approach guarantees that all fitting places are found, and thus we are able to rediscover every model that is free of duplicate and silent transitions, assuming that the log is complete (i.e., non-fitting places can be identified). In particular, we can rediscover both models shown in Figures 7 and 8.

Table 1. List of logs used for evaluation. The upper part lists real-life logs while the lower part shows artificial logs. Logs are referred to by their abbreviations. The `Sepsis` log has been reduced by removing all traces that occur at most twice. The log `HP2018` has not yet been published. The much smaller 2017 version can be found at [23]. We use a reduced version with all single trace variants removed. The log `Teleclaims*` results from removing the natural start and end activity activities in the log `Teleclaims` and removing the 15 % less common traces. The `Artificial1` log contains a single trace $\langle a, b, c, d, e, f\rangle$. The log `Artificial2` is a log generated based on a random model containing a loop, XOR-Split and silent transition.

Log Name	**Abbreviation**	**Activities**	**Trace Variants**	**Reference**
Sepsis-mod	`Sepsis`	11	27	[24]
HelpDesk2018SiavAnon-mod	`HD2018`	11	595	(see caption)
Road Traffic Fine Management	`RTFM`	11	231	[25]
Teleclaims	`Teleclaims`	11	12	[26]
Teleclaims-mod	`Teleclaims*`	9	12	[26]
repairexample	`Repair`	12	77	[26]
running-example	`RunEx`	8	6	[26]
MyLog1	`Artificial1`	6	1	(see caption)
MyLog2	`Artificial2`	7	78	(see caption)
N7	`a++CE`	7	3	[4]

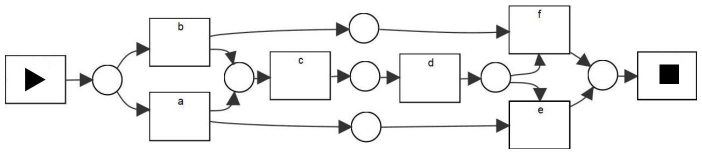

Fig. 7. The shown model can be rediscovered by our algorithm. Since (a, e) and (b, f) are not part of any directly follows relation, most other discovery algorithms fail to discover the corresponding places ([4]).

Dealing with noise: By setting the threshold τ to 1 we require every fitting place, and thus the whole discovered Petri net, to be able to perfectly replay every trace of the log. However, event logs often contain some noise, be it due to exceptional behavior or due to logging errors. Often, we want to ignore such irregularities within the log when searching for the underlying model. Therefore, we suggest using the parameter τ as a noise filtering technique utilizing the internal working of our algorithm. In contrast to approaches modifying the whole event log, this allows us to filter for each place individually: based on the portion of the log relevant for the current place, we can ignore a certain fraction of irregular behavior specified by τ without losing other valuable information.

We test our implementation using different values for τ on logs modified to contain several levels of noise. We run our algorithm with different values for τ on logs with 1000 traces and added random noise of 2, 4, 10 and 20%. The resulting model is tested for precision with respect to the original log using the ETC Align Precision Metric implemented in ProM ([22]). As shown in Figure 9,

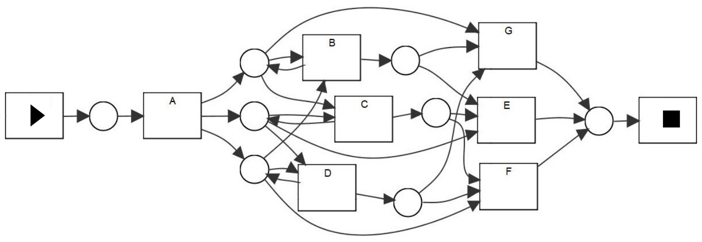

Fig. 8. The Petri net shown in this Figure is an example of a model rediscoverable by our algorithm, that existing non-region-based algorithms fail to find ([4]). The only free choice is whether to do B, C or D after doing A. The rest of the trace is implicitly determined by this choice.

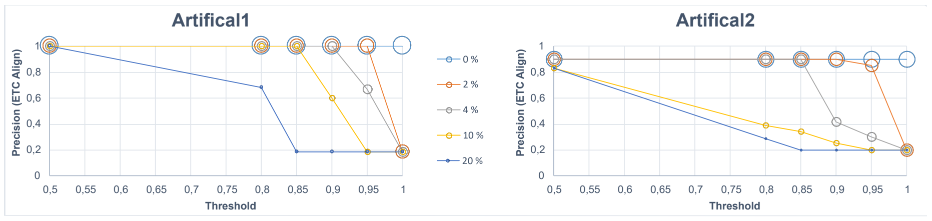

Fig. 9. For various levels of noise (0, 2, 4, 10, 20 %) introduced randomly on logs with 1000 traces each, we present the precision results (ETC Align) resulting from applying our algorithm with different values for the threshold τ. Lower values of τ have a positive impact on precision, and all tested models could be rediscovered for certain values of τ.

a lower threshold τ, in general, leads to increasing precision of the discovered model. In fact, for adequate values of τ the original models could be rediscovered. Thus, by choosing adequate values for the threshold τ, our algorithm is able to handle reasonable levels of noise within the given event log.

Measuring performance: The main contribution of our approach lies in our strategy of computing the set of fitting places $\mathcal{P}_{\text{fit}}$. The post-processing step, where implicit/redundant places are removed, has not been optimized and constitutes an application of existing results. Therefore our performance evaluation focuses on the first step. Based on several real-life logs as well as artificial logs we investigate two measures of performance: first, the absolute computation time needed to discover $\mathcal{P}_{\text{fit}}$, compared to the time needed by a brute force approach traversing the whole set of candidates ($\mathcal{P}_{\text{cand}}$), and second, the fraction of places that were cut off, that is $\mathcal{P}_{\text{skipped}}/\mathcal{P}_{\text{cand}}$. For each log, we performed 10 runs of our algorithm using two mutually independent random activity orderings for in- and outgoing activity sets to survey the influence on these performance measures.

Fraction of skipped places: The fraction of cut-off places can vary greatly between different event logs, but also within several runs of our algorithm on the same event log, depending on the chosen activity orderings for in- and outgoing

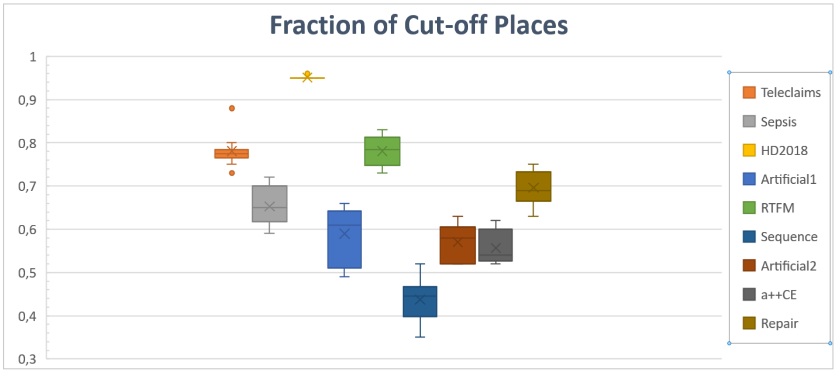

Fig. 10. Depending on the given log and activity orderings, the fraction of places skipped by our algorithm can vary greatly. In this figure, we present box plots showing the fraction of cut-off places $\mathcal{P}_{\text{skipped}}$ for 10 sample runs of our algorithm on different logs, given as the fraction of the complete candidate space ($\mathcal{P}_{\text{skipped}}/\mathcal{P}_{\text{cand}}$). The threshold τ has been set to 1 for all runs.

activities. In Figure 10, we present the results for several logs, based on 10 runs of our algorithm for each. Interestingly, the fraction of cut-off places is highest for the real-life event logs `RTFM, HD2018, Sepsis` and `Teleclaims`. For `HD2018` it goes as high as 96 % of the candidate places, that were never visited. The reason for this could be the more restrictive nature of these large and complex logs, resulting in a smaller set of fitting places $\mathcal{P}_{\text{fit}}$, and thus more possibilities to cut off branches. The lowest results are obtained for the artificial `Artificial1` log. Here we could confirm the expectation stated in Section 6, that an ordering of high average index first for ingoing activities and lo average index first for outgoing activities, leads to significantly more places being cut off than using the same ordering for both activity sets.

Comparison to the brute force approach: We evaluate the increase in performance of computing $\mathcal{P}_{\text{fit}}$ using our algorithm pruning the candidate space, in comparison to the brute force approach traversing the candidate space without any pruning. We choose three real-life event logs, `RTFM, HD2018` and `Sepsis` and perform 10 runs of our algorithm on each. In Figure 11 the results of these tests are presented: we compare the time needed by the brute force approach to the minimum, maximum and average time needed for each of the three logs. As is to be expected, the impact of applying our technique is most significant for large logs, where replaying the log on a place takes a long time, and thus cutting off places has a big effect. This is the case for the `RTFM` and `HP2018` logs. The `Sepsis` log, on the other hand, has shorter and fewer traces, reflected in a smaller difference in the time needed by our algorithm compared to the brute force approach.

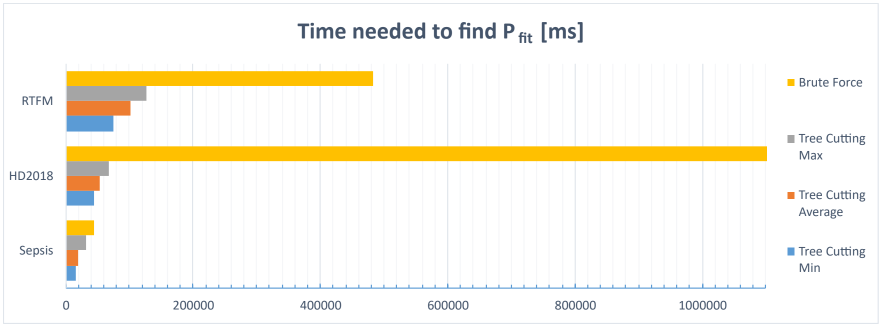

Fig. 11. Minimal, maximal and average time in milliseconds needed by 10 sample runs of our tree cutting algorithm on three real-life logs, with $\tau = 1$, compared to the time needed by the brute force approach traversing the complete candidate space.

8 Conclusion

We have introduced a process discovery algorithm inspired by language-based regions to find a model fitting with respect to a given event log and threshold τ. In contrast to non-region-based approaches our algorithm is able to discover complex structures, most prominently implicit dependencies expressed by non-free choice constructs. In particular, for $\tau = 1$, we can guarantee that the set of fitting places we discover defines a Petri net, such that the language of this net is the minimal language containing the input language (given by the event log). Our candidate traversal strategy allows us to skip large parts of the search space, giving our algorithm a huge performance boost over the brute force approach.

A well-known disadvantage of applying region theory is, that in the context of infrequent behavior within the log, the resulting models tend to be overfitting, not reflecting the core of the underlying process. We can avoid this issue, since our approach lends itself to using the threshold τ as an intuitive noise control mechanics, utilizing the internal workings of our algorithm.

We can see several possibilities for future research based on the presented ideas. The most important contribution of our work is the reduction of the search space to merely a fraction of its original size: we organize all candidate places into trees and are able to cut off subtrees, that are known to contain only unfitting places. Our approach would strongly benefit from any strategy allowing for more subtrees to be cut off or pre-pruning of the candidate space. Note, that within our work we focus on formal strategies that provide the guarantee that all fitting places are discovered. For practical applications it is important to develop heuristic techniques to increase the number of skipped places. Compared to many other approaches this is relatively easy.

New insights can be gained from further testing and evaluating different activity orderings or tree traversal schemes. By developing heuristics on how to choose orderings, based on certain characteristics of the given log, one can optimize the number of skipped candidate places without losing formal guarantees.

These guarantees are no longer given, when applying heuristic approaches that allow for fast identification and skipping of subtrees that are likely to be uninteresting. However, for practical applications, the increase in performance is likely to justify the loss of a few fitting places. Alternatively, by evaluating the more interesting subtrees first, the user can be shown a preliminary result, while the remaining candidates are evaluated in the background. User-definable or heuristically derived combinations of ingoing and outgoing activity sets can either be cut off during traversal, or excluded from the search space from the very beginning. Finally, note that our approach allows for each subtree to be evaluated independently, and thus lends itself to increase performance by implementing it in a parallelized manner.

A major issue of our algorithm is the inability to deal with silent and duplicate activities. There exist approaches to identify such activities, either as a general preprocessing step ([27]) or tailored towards a particular algorithm ([28,29]). The applicability of such strategies to our approach remains for future investigation.

We emphasize that our idea is applicable to all process mining related to Petri net definable models, and therefore we see potential not only in our discovery algorithm itself, but also in the combination with, and enhancement of, existing and future approaches.

Acknowledgments: We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research. We thank Alessandro Berti for his support with implementing the eST-Miner, and Sebastiaan J. van Zelst for reviewing this paper.

References

1. van der Aalst, W.: Process Mining: Data Science in Action, 2 edn. Springer, Heidelberg (2016)
2. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. Proc. 5th Int. Conf. on Business Process Management pp. 375–383 (2007)
3. van der Aalst, W.: Discovering the "glue" connecting activities - exploiting monotonicity to learn places faster. In: It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab, pp. 1–20 (2018)
4. Wen, L., van der Aalst, W., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery **15**(2), 145–180 (2007)
5. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs - a constructive approach. Application and Theory of Petri Nets and Concurrency pp. 311–329 (2013)
6. Weijters, A., van der Aalst, W.: Rediscovering workflow models from event-based data using little thumb. Integrated Computer-Aided Engineering **10**(2), 151–162 (2003)
7. Badouel, E., Bernardinello, L., Darondeau, P.: Petri Net Synthesis. Text in Theoretical Computer Science, an EATCS Series. Springer (2015)
8. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Acta Informatica **27**(4), 343–368 (1990)

9. Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering Petri nets from event logs. pp. 358–373 (2008)
10. van der Aalst, W., Rubin, V., Verbeek, H., van Dongen, B., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software & Systems Modeling **9**(1), 87 (2008)
11. Lorenz, R., Mauser, S., Juhás, G.: How to synthesize nets from languages: A survey. In: Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best is Yet to Come, WSC '07, pp. 637–647. IEEE Press, Piscataway, NJ, USA (2007)
12. Darondeau, P.: Deriving unbounded Petri nets from formal languages. In: CONCUR'98 Concurrency Theory, pp. 533–548. Springer, Berlin, Heidelberg (1998)
13. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri nets from finite partial languages. Fundam. Inf. **88**(4), 437–468 (2008)
14. van der Werf, J.M., van Dongen, B., Hurkens, C., Serebrenik, A.: Process discovery using integer linear programming. In: Applications and Theory of Petri Nets, pp. 368–387. Springer, Berlin, Heidelberg (2008)
15. van Zelst, S., van Dongen, B., van der Aalst, W.: Avoiding over-fitting in ilp-based process discovery. In: Business Process Management, pp. 163–171. Springer International Publishing, Cham (2015)
16. van Zelst, S., van Dongen, B., van der Aalst, W.: Ilp-based process discovery using hybrid regions. In: ATAED@Petri Nets/ACSD (2015)
17. Berthelot, G.: Checking properties of nets using transformation. In: Advances in Petri Nets 1985, Covers the 6th European Workshop on Applications and Theory in Petri Nets-selected Papers, pp. 19–40. Springer-Verlag, London, UK, UK (1986)
18. Berthelot, G.: Transformations and decompositions of nets. In: Petri Nets: Central Models and Their Properties, pp. 359–376. Springer, Berlin, Heidelberg (1987)
19. Colom, J., Silva, M.: Improving the linearly based characterization of p/t nets. In: Advances in Petri Nets 1990, pp. 113–145. Springer, Berlin, Heidelberg (1991)
20. Garcia-Valles, F., Colom, J.: Implicit places in net systems. Proceedings 8th International Workshop on Petri Nets and Performance Models pp. 104–113 (1999)
21. Berthomieu, B., Botlan, D.L., Dal-Zilio, S.: Petri net reductions for counting markings. CoRR **abs/1807.02973** (2018)
22. van Dongen, B., de Medeiros, A., Verbeek, H., Weijters, A., van der Aalst, W.: The prom framework: A new era in process mining tool support. In: Applications and Theory of Petri Nets 2005, pp. 444–454. Springer, Berlin, Heidelberg (2005)
23. Polato, M.: Dataset belonging to the help desk log of an Italian company (2017)
24. Mannhardt, F.: Sepsis cases - event log (2016)
25. De Leoni, M., Mannhardt, F.: Road traffic fine management process (2015)
26. van der Aalst, W.M.P.: Event logs and models used in Process Mining: Data Science in Action (2016). URL http://www.processmining.org/event_logs_and_models_used_in_book
27. Lu, X., Fahland, D., van den Biggelaar, F., van der Aalst, W.: Handling duplicated tasks in process discovery by refining event labels. In: Business Process Management, pp. 90–107. Springer International Publishing, Cham (2016)
28. Li, J., Liu, D., Yang, B.: Process mining: Extending α-algorithm to mine duplicate tasks in process logs. In: Advances in Web and Network Technologies, and Information Management, pp. 396–407. Springer, Berlin, Heidelberg (2007)
29. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. In: Advances in Data and Web Management, pp. 358–365. Springer, Berlin, Heidelberg (2007)