Towards Improving the Representational Bias of Process Mining

Wil van der Aalst, Joos Buijs, and Boudewijn van Dongen

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
{W.M.P.v.d.Aalst,J.C.A.M.Buijs,B.F.v.Dongen}@tue.nl

Abstract. Process mining techniques are able to extract knowledge from event logs commonly available in today's information systems. These techniques provide new means to *discover*, *monitor*, and *improve processes* in a variety of application domains. Process discovery—discovering a process model from example behavior recorded in an event log—is one of the most challenging tasks in process mining. A variety of process discovery techniques have been proposed. Most techniques suffer from the problem that often the discovered model is *internally inconsistent* (i.e., the model has deadlocks, livelocks or other behavioral anomalies). This suggests that the *search space should be limited* to sound models. In this paper, we propose a *tree representation* that ensures soundness. We evaluate the impact of the search space reduction by implementing a simple *genetic algorithm* that discovers such *process trees*. Although the result can be translated to conventional languages, we ensure the internal consistency of the resulting model while mining, thus reducing the search space and allowing for more efficient algorithms.

1 Introduction

More and more events are being recorded. Over the last decade we have witnessed an exponential growth of event data. Information systems already record lots of transactional data. Moreover, in the near future an increasing number of devices will be connected to the internet and products will be monitored using sensors and RFID tags. At the same time, organizations are required to improve their processes (reduce costs and response times) while ensuring compliance with respect to a variety of rules. *Process mining* techniques can help organizations facing such challenges by exploiting hidden knowledge in event logs. Process mining is an emerging research discipline that provides techniques to *discover, monitor and improve processes based on event data* [4].

Starting point for process mining is an *event log*. All process mining techniques assume that it is possible to *sequentially* record *events* such that each event refers to an *activity* (i.e., a well-defined step in the process) and is related to a particular *case* (i.e., a process instance). Event logs may store additional information such as the *resource* (i.e., person or device) executing or initiating an activity, the *timestamp* of an event, or *data elements* recorded with an event (e.g., the size of an order). We often distinguish three main types of process mining:

K. Aberer, E. Damiani, and T. Dillon (Eds.): SIMPDA 2011, LNBIP 116, pp. 39–54, 2012.

- *Discovery*: take an event log and produce a model without using any other a-priori information. There are dozens of techniques to extract a process model from raw event data. For example, the classical α algorithm is able to discover a Petri net by identifying basic process patterns in an event log [10]. For many organizations it is surprising to see that existing techniques are indeed able to discover real processes based on merely example executions recorded in event logs. Process discovery is often used as a starting point for other types of analysis.
- *Conformance*: an existing process model is compared with an event log of the same process. The comparison shows where the real process deviates from the modeled process. Moreover, it is possible to quantify the level of conformance and differences can be diagnosed. Conformance checking can be used to check whether reality, as recorded in the log, conforms to the model and vice versa. There are various applications for this (compliance checking, auditing, six-sigma, etc.) [30].
- *Enhancement*: take an event log and process model and extend or improve the model using the observed events. Whereas conformance checking measures the alignment between model and reality, this third type of process mining aims at changing or extending the a-priori model. For instance, by using timestamps in the event log one can extend the model to show bottlenecks, service levels, throughput times, and frequencies [4].

In this paper we focus on *process discovery*. However, we would like to stress that process discovery is only the starting point for other types of analysis. After linking events to process model elements it becomes possible to check conformance, analyze bottlenecks, predict delays, and recommend actions to minimize the expected flow time.

To illustrate the notion of process discovery see Fig. 1(a-b). Based on the event log shown in Fig. 1(a), we can discover the Petri net shown in Fig. 1(b). For simplicity we use a rather abstract description of the event log: process instances are represented by sequences of activity names (traces). For example, there are 42 cases that followed trace *abce*, 38 cases that followed trace *acbe*, and 20 cases that followed trace *ade*. This small event log consists of 380 events describing 100 process instances. There are 80 events corresponding to the execution of activity *b*. Here we abstract from additional information such as the person executing or initiating an activity, the timestamp of an event, and associated data elements. The Petri net shown in Fig. 1(b) describes a process model able to explain the observed behavior.

Process discovery can be seen as a search process, i.e., given an event log search for the model that describes the observed behavior best. When using Petri nets to describe process models, the search space consists of *all possible Petri nets*. However, even when we discard the event log, we can identify Petri nets that are clearly undesirable. Figure 1(c-d) shows two additional candidate models. Model N_2 has two potential deadlocks. After executing a and b we reach the state with just a token in place *p2*. Transition e is not enabled because there have to be tokens in both input places (*p2* and *p3*) in order for e to occur. Hence, N_2 gets

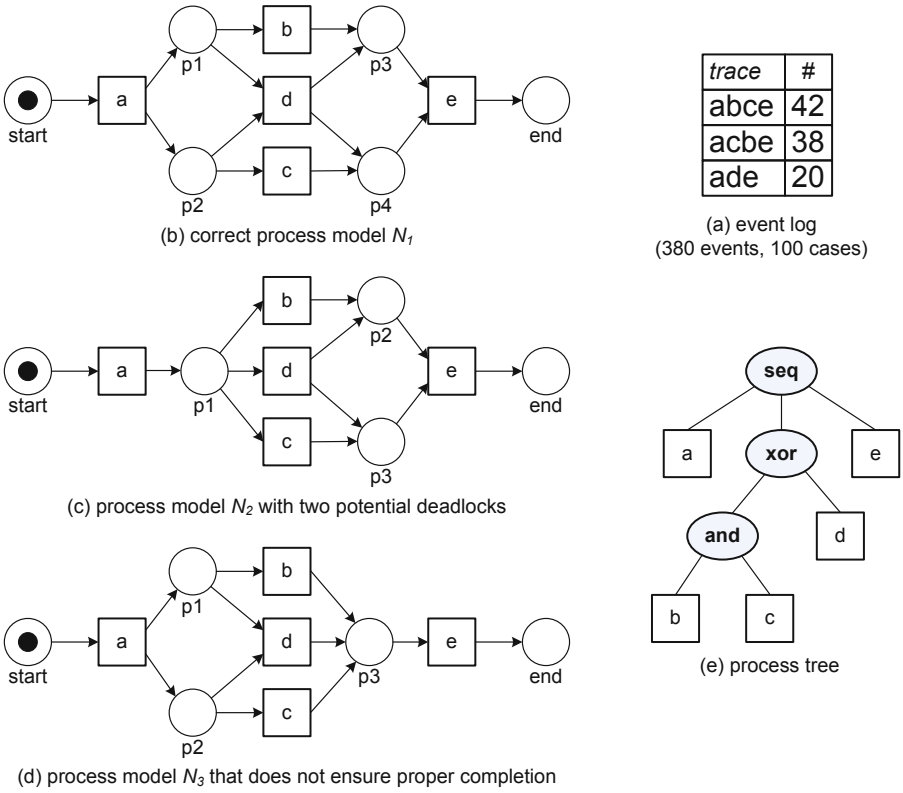


Fig. 1. A small event log (a) and four process models (b-e)

"stuck" after executing the partial trace *ab*. A similar deadlock is encountered after executing *ac*. Only after executing partial trace *ad*, transition *e* becomes enabled and the process can successfully terminate with a token in *end*.

N_3 in Fig. 1(d) has another problem. It is possible to execute trace *abe* that puts a token in place *end*. However, a token is left in *p2*. Although the process seems to have completed (token in *end*), it is still possible to execute *c*. Whereas N_2 was unable to replay the event log in Fig. 1(a), N_3 is able to replay the event log but 80 of the 100 cases do not reach the desired final state with just a token in place *end*.

The anomalies illustrated by N_2 and N_3 in Fig. 1(c-d) are *not* specific for Petri nets. Any of the main business process modeling languages (EPC, BPMN, UML, YAWL, etc.) [34] allows for deadlocks, livelocks, and improper termination. These anomalies exist independent of the event log, e.g., the event log is not needed to see that N_2 has a deadlock. Nevertheless, most process discovery techniques consider such incorrect models as possible candidates. This means that the search space is composed of both correct and incorrect models. It is not easy to limit the search space to only correct models. For common notations

such as Petri nets, EPCs, BPMN, UML activity diagrams, and YAWL models it is only possible to check correctness afterwards. Note that deadlocks and livelocks are non-local properties. Hence, simple syntactical correctness-preserving restrictions are not possible without severely crippling expressiveness.

In earlier work, we used *genetic algorithms* to discover process models [7, 28]. However, these algorithms suffered from the problem that the majority of process models considered during the search process has anomalies such as deadlocks, livelocks, and improper termination. Therefore, we propose to use *process trees* for process mining. Process trees such as the one shown in Fig. 1(e) cannot have any of the anomalies mentioned before (deadlocks, livelocks, etc.) as they are sound by design [27]. Process trees are discussed in more detail in Section 3. Using process trees as a *new representational bias*, we propose a new generation of genetic process discovery algorithms. *By limiting the search space to process trees, we can improve the quality of the discovered results and speed up the search process.*

The remainder of this paper is organized as follows. Section 2 elaborates on the importance of selecting the right representational bias. Section 3 formalizes the representational bias used in this paper and Section 5 introduces a new genetic algorithm. The first experimental results are presented in Section 6. Section 7 concludes the paper.

2 On the Representational Bias of Process Mining

In this section we discuss challenges related to process discovery and explain why an appropriate representational bias [3] needs to be selected.

2.1 Process Discovery as a Search Problem

Starting point for process mining is an event log composed of individual events. Each event refers to a case (process instance). Events corresponding to a case are ordered. Therefore, a case can be described by a trace, i.e., a sequence of activity names. Recall that in this paper we abstract from attributes such as timestamps, resources and additional data elements, and focus on activity names only. Different cases may have the same trace. Therefore, an event log can be formalized as a *multiset* of traces (rather than a set). A denotes the set of *activities* that may be recorded in the log. $\sigma \in A^*$ is a *trace*, i.e., a sequence of events. $L \in \mathbb{B}(A^*)$ is an *event log*, i.e., a multiset of traces. For example, the event log shown in Fig. 1(a) can be formalized as follows: $L = [abce^{42}, acbe^{38}, ade^{20}]$. Note that the trace *abce* appears 42 times in this event log.

A process discovery algorithm can be seen as a function f that, given an event log L, produces a model M, i.e., $f \in \mathbb{B}(A^*) \rightarrow \mathcal{M}$ where $\mathcal{M}$ is the class of process models considered. $\mathcal{M}$ is the *representational bias*, i.e., the set of possible candidate models. For example, the α algorithm can be seen a function that produces Petri net N_1 shown in Fig. 1(a) based on event log $L = [abce^{42}, acbe^{38}, ade^{20}]$. In this example, the representational bias $\mathcal{M}$ is the class of Petri nets where all transitions have unique labels (there cannot be two transitions with the same label).

Since the mid-nineties several groups have been working on techniques for process mining [8, 10, 12, 15, 17, 19, 20, 32], i.e., discovering process models based on observed events. In [5] an overview is given of the early work in this domain. The idea to apply process mining in the context of workflow management systems was introduced in [12]. In parallel, Datta [17] looked at the discovery of business process models. Cook et al. investigated similar issues in the context of software engineering processes [15]. Herbst [26] was one of the first to tackle more complicated processes, e.g., processes containing duplicate tasks.

Most of the classical approaches have problems dealing with concurrency. The α-algorithm [10] is an example of a simple technique that takes concurrency as a starting point. However, this simple algorithm has problems dealing with complicated routing constructs and noise (like most of the other approaches described in literature). In [19, 20] a more robust but less precise approach is presented.

Region-based approaches are able to express more complex control-flow structures without underfitting. State-based regions were introduced by Ehrenfeucht and Rozenberg [22] in 1989 and generalized by Cortadella et al. [16]. In [9, 21] it is shown how these state-based regions can be applied to process mining. In parallel, several authors applied language-based regions to process mining [13, 33]. In [14] a related approach based on convex polyhedra is presented.

For practical applications of process discovery it is essential that *noise* and *incompleteness* are handled well. Surprisingly, only few discovery algorithms focus on addressing these issues. Notable exceptions are heuristic mining [32], fuzzy mining [24], and genetic process mining [7, 28].

See [4] for a more elaborate introduction to the various process discovery approaches described in literature.

2.2 Balancing between Quality Criteria Such as Fitness, Simplicity, Precision, and Generalization

Generally, we use four quality dimensions for judging the quality of the discovered process model: *fitness*, *simplicity*, *precision*, and *generalization*. As illustrated by Fig. 2(a), the different criteria may be competing.

A model with good *fitness* allows for the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. There are various ways of defining fitness [4]. It can be defined at the case level, e.g., the fraction of traces in the log that can be fully replayed. It can also be defined at the event level, e.g., the fraction of events in the log that are indeed possible according to the model.

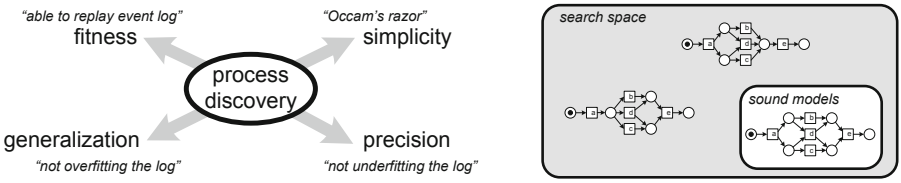
Fitness is not sufficient as it is easy to construct process models that allow for all imaginable behavior ("underfitting") or simply encode the example behaviors stored in the event log ("overfitting").

A model is *precise* if it does not allow for "too much" behavior. A model that is not precise is "underfitting". Underfitting is the problem that the model overgeneralizes the example behavior in the log, i.e., the model allows for behaviors very different from what was seen in the log. For example, a Petri net without

places and just transactions $\{a, b, c, d, e\}$ is able to replay the example event log, but also any other event log referring to the same set of activities.

A model should *generalize* and not restrict behavior to the examples seen in the log. A model that does not generalize is "overfitting". Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior, i.e., the model explains the particular sample log, but a next sample log of the same process may produce a completely different process model.

The *simplicity* dimension refers to *Occam's Razor*; the simplest model that can explain the behavior seen in the log, is the best model. The complexity of the model can be defined by the number of nodes and arcs in the underlying graph. Also more sophisticated metrics can be used, e.g., metrics that take the "structuredness" or "entropy" of the model into account.



(a) balancing between different quality dimensions

(b) the search space often allows for unsound models

Fig. 2. Process discovery can be viewed as a search problem with possibly competing quality criteria in an enormous search space mostly populated by incorrect models

2.3 Choosing the Right Representation Bias

The four main quality dimensions mentioned in Fig. 2(a) illustrate that process discovery is a non-trivial problem. For an event log there may be a simple model with a fitness of 80% and a more complex model with a fitness of 95%. Both models can be useful. Therefore, most process discovery algorithms provide parameters to guide the discovery process. However, the search process is bounded by the representational bias $\mathcal{M}$ [3].

It is important to separate the visualization of process mining results from the representation used during the actual discovery process. The selection of the representational bias $\mathcal{M}$ should be a conscious choice and should not be (only) driven by the preferred graphical representation. To illustrate the importance of this choice, we discuss implications related to *correctness* and *expressiveness*.

The three Petri nets shown in Fig. 1(b-d) are so-called *WF-nets* (workflow nets) [1]. A WF-net is a Petri net with a designated source place (*start*) and sink place (*end*). All nodes in the net need to be on a path from *start* to *end*. A commonly used correctness notion for WF-nets is *soundness* [6]. A WF-net is sound if from any reachable state it is possible to reach the desired final state with just a token in *end*. Moreover, there should be no dead parts that

can never be executed. WF-net N_1 in Fig. 1 is sound. WF-net N_2 is not sound because of the potential deadlocks: after executing b or c it is no longer possible to reach the desired final state. WF-net N_3 is not sound because only the trace ade results in the desired final state. The notion of soundness is not specific for WF-nets and similar notions can be defined for all mainstream process modeling languages [1, 6]. Similar anomalies can be encountered in EPCs, BPMN models and the like [34].

Figure 2(b) shows the implications of having a representational bias $\mathcal{M}$ that allows for unsound models. Consider for example a genetic process mining algorithm. Initially, process models are generated randomly. Obviously, most of such models will not be sound. In each generation of a genetic process mining algorithm new models (called individuals) are created using mutation and crossover. When using conventional languages, such genetic operators are likely to create models with anomalies such as deadlocks and livelocks. As a result, the search process may take unnecessarily long because irrelevant models are considered (cf. Fig. 2(b)).

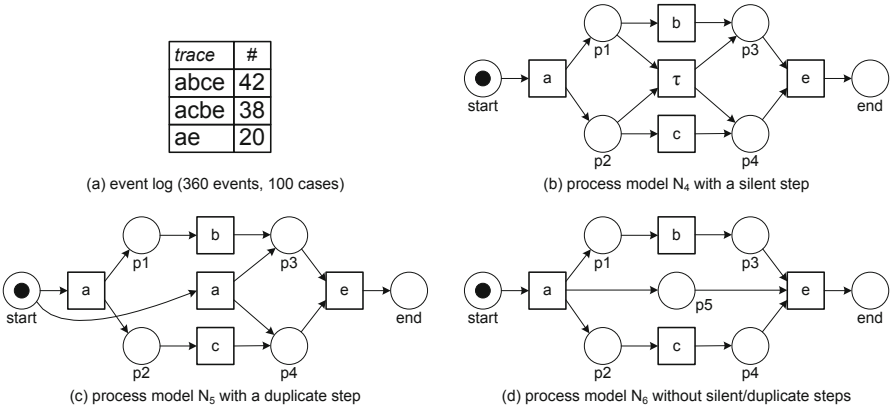


Fig. 3. One event log (a) and three process models (b-d)

The representational bias $\mathcal{M}$ also has implications on the *expressiveness* of the resulting model. To illustrate this, consider event log $L' = [abce^{42}, acbe^{38}, ae^{20}]$ shown in Fig. 3(a). Note that this is original event log where activity d is filtered out. WF-net N_4 in Fig. 3(b) has a silent step τ to model that after executing a it is possible to immediately enable e, i.e., the execution of τ is invisible and not recorded in the event log. WF-net N_5 has two a labeled transitions to model the three observed scenarios. Both N_4 and N_5 are able to reproduce the behavior seen in event log L'. However, there is no WF-net with unique visible labels having the visible behavior reflected in L'. Hence, any process discovery algorithm that has a representational bias limited to WF-nets with unique visible labels is destined to fail. The α algorithm [10] uses such a representational bias. Therefore, it is unable to discover the underlying process properly. The α algorithm produces WF-net N_6 shown in Fig. 3(d). This model does not allow for trace ae.

The example shows that the representational bias $\mathcal{M}$ may exclude desirable models. It is important that $\mathcal{M}$ supports at least the basic workflow patterns.

3 Process Trees

In this paper, we choose to use a representational bias called *process trees* that satisfy two important properties: (i) all process trees correspond to sound models, (ii) even the most basic process trees support the basic control flow patterns.

A process tree is a directed connected graph without cycles. A node V in the graph is either a branch node or a leaf node. Each leaf node represents an activity from the collection of activities $\mathcal{A}$. Each branch node, or operator node, has two children. These children can be either another operator node or a leaf. The labeling function ℓ assigns each operator node an operator from $\mathcal{O}$ and each leaf node an activity from $\mathcal{A}$. Currently, we have defined basic operators for the sequence ($\rightarrow$), exclusive choice ($\times$) and parallel ($\wedge$) constructs. Furthermore, operators for loop ($\circlearrowleft$) and OR ($\vee$) are available. The order of the children matters for the operators sequence and loop. The order of the children of a sequence operator specify the order in which they are executed (from left to right). For a loop, the left child is the 'do' part of the loop. After the execution of this 'do' part the right child, the 'redo' part, might be executed. After this execution the 'do' part is again enabled. The loop in Fig. 4 for instance is able to produce the traces $\langle A \rangle$, $\langle A, B, A \rangle$, $\langle A, B, A, B, A \rangle$ and so on. Therefore, the children of an operator node are ordered using a sorting function s. All operator nodes represent both the split and the join construction in other process modeling languages. The sequence, exclusive choice and parallel operators together cover all five basic Control Flow Patterns [2] which is one of the requirements as discussed in Section 2.3. Furthermore, by adding the loop and or operators, process trees are able to express any event log.

In contrast to Petri nets, process trees always represent sound models [27] and a straightforward translation from process trees to Petri nets exists. Figure 4 shows how each of the operators can be translated to a Petri net construct. The children of the sequential operator are ordered from left to right. In order to correctly translate the parallelism operator to a Petri net, new transitions need to be added. Since these transitions do not represent an observable activity these are marked as invisible or τ-transitions (as is for example the case in the Petri net of Figure 3(b)).

Unlike frequent pattern mining and episode mining [25], process tree discovery aims to discover end-to-end processes rather than frequent patterns. The goal is to find "complete process models" and not just process fragments that are executed frequently. Moreover, traditional data mining techniques are not considering all four quality dimensions and tend to focus on just two dimensions (e.g., fitness and simplicity).

To determine the quality of a process tree, we need metrics to measure the four dimensions. While plenty of metrics exist [18,31] to measure the quality of a Petri net, we are not aware of any metrics for process trees. Therefore, to measure the

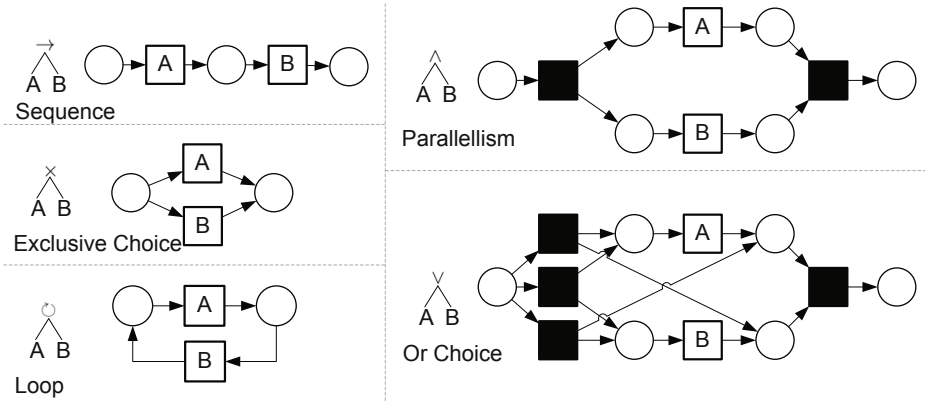


Fig. 4. Translation of Tree Operators to Petri net constructs

quality of process trees, we measure the quality of the corresponding Petri net translation, where we focus on fitness and precision. For the fitness dimension we use the cost-based fitness as defined by [11] and precision is covered by behavioral appropriateness metric which is currently under development by the same authors.

The overall quality of a process tree is computed by taking the harmonic mean of the fitness and precision metric. For two inputs (x_1 and x_2) the harmonic mean H is defined as $H = \frac{2x_1x_2}{x_1+x_2}$. The harmonic mean ensures a 'pessimistic' fitness value when the two quality metrics are more apart. This ensures that if one metric scores very high but the other very low, the overall quality is relatively low since the two metrics should be more balanced.

4 Searching for Process Trees

In the introduction, we stated that the goal of selecting the right representational bias was to limit the size of the search space. Therefore, when comparing process trees to Petri nets, we should compare the size of the search space for a given number of activities (i.e. a given number of transitions in a Petri net, or a given number of leaf nodes in a process tree).

Since a place in a Petri net can have any transition as input or as output, or is not connected to that transition, there are 3^n different places and a Petri net contains any number of places (where we neglect the initial marking).

Definition 1 (Number of Petri nets on n activities). *The number of different Petri nets on n activities is defined as:*

$$\#PNets(n) = 2^{3^n}$$

The number of possible binary process trees having 5 types of operations (AND,XOR,SEQ,OR and $LOOP$) and n leafs for the activities depends on

the number of operator nodes $(n-1)$, the selection of the operator type of each node (5^{n-1}) and the possible orderings of the leaf nodes $(n!)$, as well as the number of ordered rooted trees with n leaves:

Definition 2 (Number of binary process trees). *The number of binary process trees with three types of operator nodes can be defined as a function of the number of leaf nodes n as follows:*

$$\begin{aligned} \#Trees(n) &= \#StructuralCombinations \cdot \#OperatorChoices \cdot \#LeafOrders \\ &= C(n-1) \cdot 5^{(n-1)} \cdot n! \\ &= \frac{(2(n-1))!}{(n)!\,(n-1)!} \cdot 5^{(n-1)} \cdot n! \end{aligned}$$

where $C(n)$: the Catalan number sequence [29] which specifies the number of ordered rooted trees with n operator nodes.

Table 1 shows the number of different Petri nets and process trees for a number of activities ranging from 1 to 6. It shows the number of process trees that are possible using three operators (SEQ,XOR and AND) and when using all 5 operators. The table clearly shows that there are far less possible trees than there are Petri nets, even when using all 5 operator types. Furthermore all of the process trees represent sound models, while of the Petri nets, only a fraction is actually sound.

Table 1. Size of the search space for varying number of activities

number of activities	number of Petri nets	number of process trees	
		3 operators	5 operators
1	8	1	1
2	512	6	10
3	134,217,728	108	300
4	$2{,}41785 \cdot 10^{24}$	3,240	15,000
5	$1{,}41347 \cdot 10^{73}$	136,080	1,050,000
6	$2{,}82401 \cdot 10^{219}$	7,348,320	94,500,000

Since any process tree that can be generated represents a sound model, a naive process mining algorithm could simply generate random trees, test their quality and if the quality is not good enough, try again. To see how many of such experiments would have to be conducted, we look at a simple example with 6 activities and three different logs. Furthermore, we only consider the SEQ, XOR and AND operators. The first event log contains only one trace which describes a sequential process model, $L_{SEQ} = [abcdef]$. The next event log describes 6 activities in an exclusive choice, $L_{XOR} = [a, b, c, d, e, f]$. The third event log contains all 720 possible permutations of the 6 activities $A - F$ and thus describes the process model where these 6 activities are executed in parallel.

For the sequential log containing 6 activities we iterated over all $7,348,320$ possible trees and found there are 42 trees describing this behavior with an overall quality of 1. This can be also be calculated using Definition 2 by observing that there is only one correct order of the leaf nodes (namely A to F from left to right) and that all operator nodes should be of the type sequence. For both the exclusive choice and parallel event logs there are $30,240$ trees that describe this behavior. This implies that for the sequential case one out of $174,960$ trees is correct where for the exclusive choice and parallel case this is one out of 243 trees.

Generating random trees until the first tree is found with quality 1 can be seen as the repetition of a Bernoulli experiment until the first success, and hence this process follows a geometric distribution. Table 2 shows how many trees we expect to generate, when doing such an experiment 100 times, i.e. in order to randomly find a process tree with perfect quality for our sequential log, the number of trees that we expect to need to generate is $174,960$ with a 99% confidence interval of 4506.66.

Table 2. Results for Process Trees with 6 Activities

	Sequential	**Exclusive Choice**	**Parallel**
Trees considered	174,960	243	243
Variance	306,108,266	5,881	5,881
99% Confidence Interval	4506.66	19.75	19.75

Clearly, just randomly searching for a process tree is not a good idea when mining a process model. Therefore, in Section 5 we investigated the use of genetic algorithms to more efficiently search for a process tree.

5 Genetic Mining for Process Trees

Genetic process mining is a technique to discover process models from an event log of observed behavior. Instead of using a deterministic approach, as most discovery algorithms do, an evolutionary algorithm is applied [23]. This approach has first been applied to process mining in [7, 28].

The main idea of evolutionary algorithms is that one of the following is unknown [23]: the input to the problem, the model or algorithm, or the desired output. In the case of genetic mining the input, the event log, is known and provided. The desired output given the input is not known exactly, but we can determine which solutions are better or worse than others by providing a fitness value for each candidate model. Therefore the goal of the genetic mining algorithm is to provide a (process) model that describes the observed behavior in the event log 'best' given a certain fitness. The fitness function can be used to emphasize desired characteristics of the resulting process model.

A genetic algorithm is a search heuristic where a suitable solution needs to be found among possible candidates. The search space is searched by (semi-) randomly creating and changing candidates until certain stop criteria are fulfilled. In general a genetic algorithm follows the flow as shown in Figure 5. The initial population, or set of candidate solutions, can be created completely random or using some simple heuristics. In the next step the fitness of each candidate is calculated. If the algorithm should continue then the candidates are changed. In general there are two types of change operations: *crossover* mixes elements of two candidates creating two children; *mutation* changes one or more details of a candidate. These change operations are applied to a selection of the fitter candidates. Furthermore, for a small group with the highest fitness values a copy is created, which is not changed, to make sure that a copy of the fittest candidates survive. Each cycle of fitness calculation and population changes is called a generation. When a specified minimum fitness value is achieved, or when a specified maximum number of generations or execution time has been reached, the algorithm stops. The fittest candidate is now returned as the output of the algorithm.

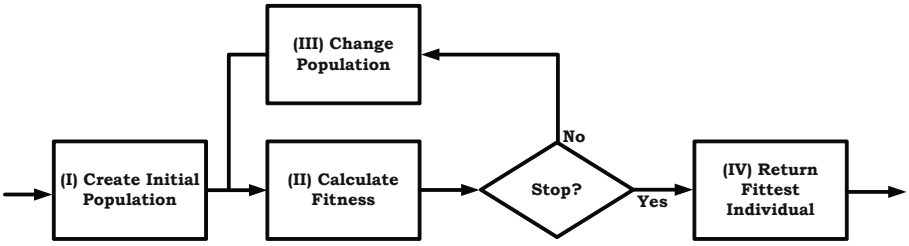


Fig. 5. Genetic Algorithm flow

In [7, 28], a genetic algorithm is presented to mine Petri nets. This genetic algorithm uses an internal matrix representation that can easily be translated to a Petri net and vice versa. In essence the authors represent a Petri net as a matrix which makes it easier to define correct mutation and crossover functions. Since the matrix representation is almost as expressive as Petri nets and is able to describe unsound Petri nets, the search space is equivalent in size as the search space of Petri nets, shown in Table 1.

The fitness function, which specifies how 'good' a certain candidate is considering the event log, is the second most important part of a genetic algorithm. The fitness function determines the main characteristics of the output. It can for instance consider all, or a selection of, the quality dimensions shown in Figure 2. It is important however to correctly balance the different quality dimensions. If the fitness function for instance only considered the 'fitness' dimension, the resulting model is likely to have a very low 'precision' on the event log. Furthermore, if the event log contains a lot of noise, aiming for a perfect 'fitness' might not result in the desired process model. For simplicity, our fitness calculation

directly uses the quality metrics discussed in Section 3 which cover the fitness and precision dimensions.

Another important aspect of the genetic algorithm are the crossover and mutation functions specified on the internal representation. The main purpose of the crossover function is to combine two good parts from two candidates together in a single candidate. The mutation function randomly modifies a candidate to introduce possibly new behavior that might be beneficial for the fitness. Together the crossover and mutation functions need to make sure that all possible candidates *can* be discovered.

In order to change the process tree candidates we define 4 mutation functions on process trees. The single node mutation selects a single vertex and modifies the labeling function ℓ on that node. This means that an operator node gets a different operator assigned and a leaf node represent a different activity. The second mutation function adds a leaf node with a randomly selected activity to a randomly selected operator node. In a similar way the third mutation function removes a randomly selected vertex, which might be an operator node. The most complicated mutation function is the 'internal crossover' mutation which swaps subtrees within a single process tree.

Our implementation currently does not use crossover since initial experiments showed that this was not beneficial for the performance. For the future we plan to add a guided crossover.

The performance of any genetic algorithm is determined mainly by the size of the search space and the time needed to compute the fitness of an individual. By using process trees as the internal representation we drastically reduce the search space as we only consider sound models and therefore we improve performance of the genetic algorithm. In this paper, we did not try to optimize the fitness computations.

6 Experimental Results

In this section we discuss the first experimental results of the initial version of the genetic algorithm. For these experiments we ran the genetic mining algorithm 100 times and recorded how many trees it created in order to find a candidate with an overall fitness of 1. We used a small population of 10 candidates in each generation. In each next generation the 2 fittest candidates of the last generation are copied without mutation. The 8 other candidates are created by first copying one of the candidates of the previous generation and then applying one of the mutation functions. Initialization of the 10 trees in the first generation was done completely random in order to test the mutation functions.

We applied the genetic algorithm on the three event logs introduced before.

We compare the performance of our genetic algorithm to the case where trees are created completely random until a suitable candidate has been found. All of our experiments are independent and identically distributed. Therefore we can compare the random case, where we would create trees completely random, with our experiments. In the random case we would on average need to create $174,960$

process trees for the sequential or 243 process trees for the other 2 event logs. The 99% confidence interval is +/- 4507 and +/- 19.75, respectively, as shown in Table 3 (note that we copied the results of Table 2 for easy comparison).

Table 3. Results for Process Trees with 6 Activities

	Sequential		**Exclusive Choice**		**Parallel**	
	Random	Genetic	Random	Genetic	Random	Genetic
Trees considered	174,960	459.84	243	100.96	243	100.88
Variance	306,108,266	95,042	5,881	8,454	5,881	18,515
Standard dev	17,495.95	308.29	76.69	91.95	76.69	136.07
99% Confidence Interval	4506.66	79.41	19.75	23.68	19.75	35.05

The experimental results shown in Table 3 clearly show an improvement over the random case. In all cases the number of trees that needed to be generated before finding a tree with perfect fitness was far less than the random case, even when considering the 99% confidence intervals.

This shows that by first drastically reducing the search space, followed by an application of a simple genetic mining algorithm, provides perspectives for a new genetic mining algorithm. Of course, performance can be further increased by initializing the trees in a smart way. Another important future improvement is the definition of a custom fitness function directly on process trees. This could drastically reduce the time required for a fitness calculation, which currently is the main performance issue. These initial results do conform our intuition that process trees are a good candidate for genetic mining.

7 Conclusion

Most process mining discovery algorithms suffer from the problem that the discovered model is potentially unsound. Considering unsound process models in a search space suggests that improvements are possible by eliminating these. In this paper, we propose an alternative representation for process models, *process trees*. Process trees are inherently sound and can easily be translated to other process modeling languages, such as Petri nets, EPCs or BPMN.

Process mining algorithms can be seen as search algorithms. By using process trees as a representation, the search space of all possible process models is drastically reduced, compared to Petri nets. In this paper we show that genetic algorithms can be defined to search this reduced search space. One of the main benefits of genetic mining algorithms is the flexibility. The desired characteristics of the resulting process model can be defined in the fitness function. This makes the genetic algorithm a versatile discovery algorithm that can be applied in many situations. In this paper we used existing Petri net based metrics to calculate fitness. As a result, our current approach is rather slow and not very suitable for real-life processes. However, it seems possible to define quality metrics directly on trees that make use of their unique characteristics. We expect that this will dramatically speed up the discovery process.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Workflow Patterns. In: Liu, L., Tamer Özsu, M. (eds.) Encyclopedia of Database Systems, pp. 3557–3558. Springer, Berlin (2009)
3. van der Aalst, W.M.P.: On the Representational Bias in Process Mining (Keynote Paper). In: Reddy, S., Tata, S. (eds.) Proceedings of the 20th Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2011), Paris, pp. 2–7. IEEE Computer Society Press (2011)
4. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer, Berlin (2011)
5. van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G., Weijters, A.J.M.M.: Workflow Mining: A Survey of Issues and Approaches. Data and Knowledge Engineering 47(2), 237–267 (2003)
6. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. Formal Aspects of Computing 23(3), 333–363 (2011)
7. van der Aalst, W.M.P., Alves de Medeiros, A.K., Weijters, A.J.M.M.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
8. van der Aalst, W.M.P., Reijers, H.A., Weijters, A.J.M.M., van Dongen, B.F., Alves de Medeiros, A.K., Song, M., Verbeek, H.M.W.: Business Process Mining: An Industrial Application. Information Systems 32(5), 713–732 (2007)
9. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. Software and Systems Modeling 9(1), 87–111 (2010)
10. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
11. Adriansyah, A., van Dongen, B., van der Aalst, W.M.P.: Conformance Checking using Cost-Based Fitness Analysis. In: Chi, C.H., Johnson, P. (eds.) IEEE International Enterprise Computing Conference, EDOC 2011, pp. 55–64. IEEE Computer Society (2011)
12. Agrawal, R., Gunopulos, D., Leymann, F.: Mining Process Models from Workflow Logs. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 469–483. Springer, Heidelberg (1998)
13. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007)
14. Carmona, J., Cortadella, J.: Process Mining Meets Abstract Interpretation. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part I. LNCS, vol. 6321, pp. 184–199. Springer, Heidelberg (2010)
15. Cook, J.E., Wolf, A.L.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology 7(3), 215–249 (1998)
16. Cortadella, J., Kishinevsky, M., Lavagno, L., Yakovlev, A.: Deriving Petri Nets from Finite Transition Systems. IEEE Transactions on Computers 47(8), 859–882 (1998)
17. Datta, A.: Automating the Discovery of As-Is Business Process Models: Probabilistic and Algorithmic Approaches. Information Systems Research 9(3), 275–301 (1998)

18. De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A critical evaluation study of model-log metrics in process discovery. In: Business Process Management Workshops: BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13-15, vol. 66, p. 158 (2011) (Revised Selected Papers)
19. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Building Instance Graphs. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 362–376. Springer, Heidelberg (2004)
20. van Dongen, B.F., van der Aalst, W.M.P.: Multi-Phase Mining: Aggregating Instances Graphs into EPCs and Petri Nets. In: Marinescu, D. (ed.) Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, pp. 35–58. Florida International University, Miami (2005)
21. van Dongen, B.F., Busi, N., Pinna, G.M., van der Aalst, W.M.P.: An Iterative Algorithm for Applying the Theory of Regions in Process Mining. In: Reisig, W., van Hee, K., Wolf, K. (eds.) Proceedings of the Workshop on Formal Approaches to Business Processes and Web Services (FABPWS 2007), pp. 36–55. Publishing House of University of Podlasie, Siedlce (2007)
22. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures - Part 1 and Part 2. Acta Informatica 27(4), 315–368 (1989)
23. Eiben, A.E., Smith, J.E.: Introduction to Evolutionary Computing. In: Natural Computing, Springer, Berlin (2003)
24. Günther, C.W., van der Aalst, W.M.P.: Fuzzy Mining – Adaptive Process Simplification Based on Multi-perspective Metrics. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 328–343. Springer, Heidelberg (2007)
25. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Mining and Knowledge Discovery 15(1), 55–86 (2007)
26. Herbst, J.: A Machine Learning Approach to Workflow Management. In: Lopez de Mantaras, R., Plaza, E. (eds.) ECML 2000. LNCS (LNAI), vol. 1810, pp. 183–194. Springer, Heidelberg (2000)
27. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. Enterprise Modelling and Information Systems Architecture 4(1), 3–13 (2009)
28. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
29. The On-Line Encyclopedia of Integer Sequences. Sequence a000108 (October 2011), Published electronically at `http://oeis.org`
30. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
31. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J.: Process Compliance Measurement Based on Behavioural Profiles. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 499–514. Springer, Heidelberg (2010)
32. Weijters, A.J.M.M., van der Aalst, W.M.P.: Rediscovering Workflow Models from Event-Based Data using Little Thumb. Integrated Computer-Aided Engineering 10(2), 151–162 (2003)
33. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundamenta Informaticae 94, 387–412 (2010)
34. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin (2007)