Scalable Inductive Process Mining

Maximilian L. Franz

***Abstract*—Since the invention of the transistor, the speed of information processing at constant cost has roughly doubled every two years. With this increase in processing power, also our ability to sense, store and transfer information has increased in an exponential manner, so that it is now easier than ever to collect and store huge amounts of data. Data by itself does not provide actionable or even understandable information, however. Thus, in order to retrieve useful information from it, we ought to find meaningful and repeating patterns. In this work, *process mining* as a discipline to discover such patterns in event logs is introduced and *inductive mining* in particular is examined as a modular approach to deal with very large event logs, yielding block-structured process trees with desirable quality guarantees.**

I. Introduction

It is no surprise that the amounts of data we are able to store and process is increasing rapidly along the lines of *Moore's Law*. Not only in the information technology companies is so called Big Data an omnipresent theme nowadays. The importance of data as a source of competitive advantage and better understanding of ones own processes is becoming self-evident in basically every industrial sector [1]. Considering this bulk of readily available data the industry is collecting [2], we need reliable tools to retrieve information from this data, because data by itself is not useful. To enable the management levels of companies to make insightful decisions and inventions we have to extract actionable pieces of information. Working with data and discovering such useful patterns in large datasets or making predictions for future events based on data is the core task of *data mining* [3]. Data mining has seen big improvements by the introduction of more powerful computers and more expressive statistical models in recent years.

Process management on the other hand has been a part of business development for quite some time. It is the discipline of modelling the behaviours and procedures inside organizations in a formal and graphic manner [4]. However, traditional *process modelling* has some major drawbacks due to the nature of its approach, which is to use human cognition to either model how the process should look or to abstract a process from vaguely observed behaviour. By being based on human judgment, process modelling introduces strong biases into the model. Common errors ([5], [6]) of traditional *process modelling* include

- The model describes an idealized version of reality.
- Human behaviour cannot be captured in simple stated forms.
- The model is at the wrong abstraction level.

It is exactly here that *process mining* as a conjunction of *data mining* and *process modelling* comes in and alleviates some of the problems. Process mining can facilitate the construction of better models in less time [5]. Mined models are based on the empirical reality from event-logs and can either help to discover errors in previously manually designed process models or create new models from scratch.

Inductive process mining is a fairly new method to discover process trees from very large event logs by recursion on smaller sub-logs. Inductive Mining has the neat property to produce only sound models by construction and it provides quality guarantees for its results [7], [8].

In Section II we will introduce terminology and basic notation of the task at hand. Also, we cover some quality criteria along with the main challenges of process mining which leads us to the choice of *inductive mining*. In section III, we introduce the idea of inductive mining to discover block-structured process models, a.k.a process trees. Lastly, in IV we shortly discuss the prospect of the method in the future and other common approaches.

II. Process Mining - Task and Terminology - Preliminaries

Process minining describes a field at the intersection of process modelling and data mining. In order to speak about *inductive mining* as a particular algorithm of process mining we first introduce some basic notation of processes and the necessary data mining techniques.

A. Processes, Cases and Logs

Since we want to present an algorithm for process discovery on event logs, we ought first define formally what is meant by a process and how it relates to a log, after [5].

A process may have an arbitrary number of activities. A case belongs to one specific process and can itself have many activity instances, which manifest precisely one activity. On the event level, a case and an activity instance can have many events, which in turn have multiple event attributes. A simplified overview can be found in Figure 1. With our algorithms we want to model processes by defining the related activities. Only when we instantiate the model can we observe cases and activities which manifest themselves in events. Thus, the only source we have for our construction is a set of observed events called a log.

For practicality we only consider *simple event logs* over a set of activity names $\mathcal{A}$. The *simple event log* L can be constructed out of a complete event log $\hat{L}$ by considering only the activity classifiers of the events without their attributes. More formally:

Definition 1. (Simple Event Log, simple trace) [8] Let $\mathcal{A}$ be a set of activity names like above. A *simple* trace σ is a sequence of activities ($\sigma \in \mathcal{A}^*$) and a simple event log is then a multi-set of traces over $\mathcal{A}$, i.e., $L \in \mathbb{B}(\mathcal{A}^*)$, where $\mathbb{B}(\mathcal{A}^*)$ is the

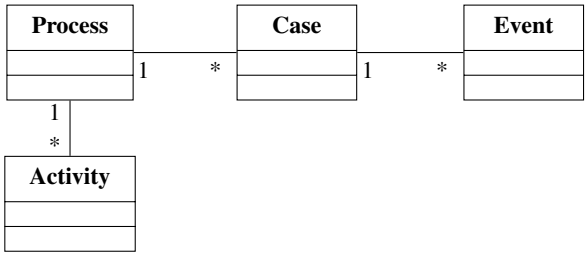

Figure 1: Relation of Processes, Cases and Events. Simplified representation after [5] to illustrate the dependencies.

multi-set over the domain $\mathcal{A}^*$ and $\mathcal{A}^*$ describes the set of all finite sequences over $\mathcal{A}$.

To illustrate, let $A = \{a, b, c\}$, then $\sigma_1 = \langle a, c, b\rangle, \sigma_2 = \langle c, b\rangle \in A^*$ are traces and $L_1 = \{\sigma_1, \sigma_2\}$ is a log. In the following we denote by L simple event logs over generic activities indetified by $\mathcal{A} = \{a, b, c, \dots\}$, by ϵ the empty trace and by $\|L\| = \sum_{\sigma \in L} |\sigma|$ the size of the log. For example $\|L_1\| = 5$.

B. Process Trees

A *process tree* after [7] is a compact and abstract representation of a block-structured workflow net and thus a subclass of Petri Nets, which can be transformed into most process modelling notations. Process trees are sound process models by construction and they describe a regular language. A process model is informally called *sound* if (i), any execution terminates in one of the pre-defined end-states and (ii), for all activities in the model there exists at least one execution which includes it [9].

Definition 2. (Process Trees) [8] A *process tree* is a graph $Q = (V, E)$ with the structure of a rooted tree. Vertices $v \in V$ are either nodes or leafs. Leafs are labeled with activites $a \in \mathcal{A}$ and nodes are labeled with operators in $\bigoplus = \{\times, \circlearrowleft, \rightarrow, \wedge\}$, which denote different causal relationships between the targeted nodes. Edges $e \in E$ connect nodes with sub-trees or activities, never activities with activities, because trees are *directed acyclic graphs* [10].

To formally describe the semantics of a process tree, we look at the recursive monotonic function $\mathcal{L}(Q)$, which maps to the language (i.e. event-log), that can be constructed by the tree. I.e.

$$\begin{aligned} \mathcal{L}(a) &= \{\langle a\rangle\} \text{ for } a \in \mathcal{A}, \\ \mathcal{L}(\tau) &= \{\epsilon\} \text{ and} \\ \mathcal{L}(\oplus(Q_1, \dots, Q_n)) &= \oplus_l(\mathcal{L}(Q_1) \dots, \mathcal{L}(Q_n))). \end{aligned}$$

Where the Q_i for $i = 1, \dots, n$ are sound subtrees (or single activities) and τ is the silent activity, which cannot be observed in the log. Thus, process trees are defined recursively. Formally, the different language join functions $\oplus_l$ function are

$$\begin{aligned} \times_l(L_1, \dots, L_n) &= \bigcup_i L_i, \\ \rightarrow_l (L_1, \dots, L_n) &= \{t_1 \cdot \dots \cdot t_n | \forall i : t_i \in L_i\}, \\ \wedge_l(L_1, \dots, L_n) &= L_1 \diamond L_2 \diamond \dots \diamond L_n, \\ \circlearrowleft_l (L_1, \dots, L_n) &= \\ &\{t_1 \cdot t'_1 \dots t_m | \forall i : t_i \in L_i \wedge t'_i \in \times(L_2, \dots, L_n)\} \end{aligned}$$

where $i = 1, \dots, n$, $L_i = \mathcal{L}(Q_i)$ and t_i are traces in the sub-logs L_i.

In order to understand the $\wedge$ join, we introduce a shuffle operator $\diamond$, which takes two sequences σ_1, σ_2 and generates the set of all interleaved sequences, where the ordering of the original sequences is preserved. E.g.,

$$\begin{aligned} \langle p, r\rangle \diamond \langle o, m\rangle = \{&\langle p, r, o, m\rangle, \langle p, o, r, m\rangle, \langle p, o, m, r\rangle, \\ &\langle o, m, p, r\rangle \langle o, p, m, r\rangle, ...\}. \end{aligned}$$

This operator can be generalized to sets of sequences by $S_1 \diamond S_2 = \{\sigma_1 \diamond \sigma_2 | \sigma_1 \in S_1 \wedge \sigma_2 \in S_2\}$ and - since it is associative - also over multiple sets $S_1, \dots, S_n$.

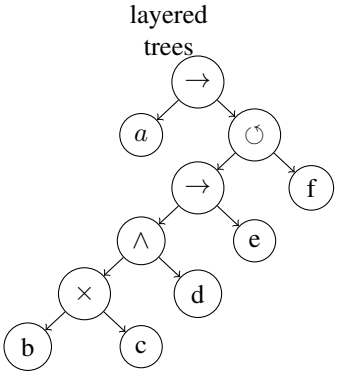

Figure 2: An example process tree Q_e

See Figure 2 for an example process tree Q_e. We can also represent this tree textually starting from the root:

$$Q_e \cong \ \rightarrow \big(a, \circlearrowleft (\rightarrow (\wedge(\times(b, c), d), e), f)\big).$$

And possible traces $\sigma_1, \sigma_2 \in \mathcal{L}(Q_e)$ that can be replayed from this model are $\sigma_1 = \langle a, c, d, e, f\rangle$ or $\sigma_2 = \langle a, c, d, e, f, d, b, e, f\rangle$ where σ_2 makes use of the loop construct and the shuffle operator for the second loop execution $(.., f, d, b, e, f)$. Each process tree operator has a formal translation to a sound, block-structured workflow Petri net (WF-net). Thus any process tree can be translated into a petri-net which in turn can be translated into almost any workflow notation (e.g. YAWL, BPMN, EPCs). For details, see [11].

Given these notations we can now more or less formally define, what we mean by *process discovery*.

Definition 3. (Process Discovery Algorithm) [5] A *process discovery algorithm* can be understood as a function γ which maps a simple event log L onto a marked Petri net, which is ideally sound.

In our case the function maps to a *process tree*, which is sound by construction and equivalent up to isomorphism to a Petri net.

C. Directly-Follows Graph

Within the *Inductive Mining* we will work with *directly-follows relations* and corresponding *directly-follows graphs*, which we quickly introduce here.

Definition 4. (Directly follows graph (DFG)) [5] Let L be a simple event log. The *directly-follows graph* of L is $G(L) = (A_L, \mapsto_L, A_L^{start}, A_L^{end})$ where

$$A_L = \{a \in \sigma | \sigma \in L\}, \text{ the set of acitivites,}$$
$$\mapsto_L = \{(a,b) \in A \times A | a >_L b\}, \text{the follows relation,}$$
$$A_L^{start} = \{a \in A | \exists_{\sigma \in L} a = first(\sigma)\}, \text{ start acitivites and}$$
$$A_L^{end} = \{a \in A | \exists_{\sigma \in L} a = last(\sigma)\}, \text{ end acitivites.}$$

For $a, b \in A, a >_l b$ means that a is directly followed by b somewhere in the log. In the following we use the short form DFG for a directly-follows graph.

Using standard graph algorithms, DFGs can be used to perfom meaningful cuts.

Definition 5. (Cut) [5] Let $G(L)$ be a DFG. A *n-ary cut* of $G(L)$ is a partition of A_L into pairwise disjoint sets $A_1, \ldots, A_n$. The notation based on the semantic relation of the partitions is $(\oplus, A_1, \ldots, A_n)$. An *n-ary cut* is *maximal* if there exists no cut $(\oplus, (A_1, \ldots, A_m))$ of $G(L)$ with $m > n$. A Cut is *nontrivial* if $n > 1$.

A Log L is called *directly-follows complete* with respect to process tree Q if (i) all activities in Q appear in the log, (ii) for every start- or end-activity in Q there is a trace starting or ending with that activity and (iii) $a \mapsto_L b$ if b can directly follow a in Q.

D. Quality Criteria of Process Models

In [5] van der Aalst introduces four competing quality criteria of process models that we quickly introduce here to get a sense of the challenges posed to *process discovery* at large.

- *Fitness*: A model is called *fit* if it is able to replay the log from which it was generated
- *Simplicity*: Along the lines of *Occam's Razor*, the best model is that which is simplest in structure
- *Generalization*: Refers to the problem of *overfitting* in machine learning, where the model is heavily specialized on the traces encountered in the log. A model *generalizes* if it is able to account for behaviour generated by the original process models which is not in the log.
- *Precision*: A Model is *precise* if it does not allow for 'too much' behaviour. That is, if it properly replays structures visible in the log, but doesn't allow for random behaviour.

To illustrate, consider a FLOWERMODEL(L), which is a model that is able to replay any log with activities in L. It is thus *fit* and reasonably *simple*, because it only contains one redo loop node. However, a FLOWERMODEL(L) is obviously not *precise*, since it allows for any behaviour. An enumeration model ENUM(L) on the other hand is built by having a distinct start to end path for every trace encountered in the log. This way, ENUM(L) is perfectly *precise* but horribly *overfitted*, because no behaviour is possible that is not in the log L.

III. INDUCTIVE MINING

Inductive Mining describes a framework which aims to discover block-structured process models (i.e. *process trees*) from event logs [8].

A. Inductive Mining - General Idea

Given a set $\bigoplus$ of operators like above, the idea is to search for possible splits of L into smaller sub-logs $L_1, \ldots, L_n$, such that the sub-logs combined by the respective operator yield L again. This selection step is performed by some - currently still abstract - $select(L)$, for which some essential properties hold. The algorithm then recurses on the found sub-logs until a base case is discovered. (e.g. $L = \{\langle a \rangle\}, \{\epsilon\}$ or $\{\varnothing\}$)

The framework independent of the actual discovery of cuts is depicted in Algorithm 1. Due to space constraints we will focus on one concrete implementation of the framework provided in [8], which is specified by the selection algorithm $select_{B'}(L)$ in III-B.

The counter variable ϕ allows the $select(L)$ procedure to return a sub-log of equal size $\|L_i\| = \|L\|$, while only doing so finitely often. Otherwise, termination could not be guaranteed. The base cases in lines 1 to 6 describe the behaviour when the algorithm encounters empty logs or logs with only one activity and thus define the terminating steps of the recursion.

Algorithm 1 Recursive $\text{B}_{select}(L,\phi)$

Require: the log L, counter variable ϕ
1: **if** $L = \{\epsilon\}$ **then** ▷ handle base cases
2: base $\leftarrow \{\tau\}$
3: **else if** $\exists a \in \Sigma : L = \{\langle a \rangle\}$ **then**
4: base $\leftarrow \{a\}$
5: **else**
6: base $\leftarrow \varnothing$
7: $P \leftarrow select(L)$
8: **if** $|P| = 0$ **then**
9: **if** base $= \varnothing$ **then**
10: **return** FLOWERMODEL(L)
11: **else**
12: **return** base
13: **return** $B(P)$ ▷ Recursion occurs here

Where

$$B(P) = \Big\{ \oplus(M_1, \ldots, M_n) \quad | \quad (\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n))) \in P \wedge M_i \in B(L_i, \phi_i) \Big\} \cup \text{base}$$

and FLOWERMODEL(L) returns a loop that can replay any log containing the acitivities in L. We see that the generic procedure $select(L)$ returns tuples $(\oplus, ((L_1, \phi_1), \ldots, (L_n, \phi_n)))$, for which all of the following conditions must hold.

- The operator applied on the sublogs L_i must yield a super-set of L
- $\forall i : \|L_i\| + \phi_i < \|L\| + \phi$, so that the recursion ends
- $\forall i : L_i$ less than or equal to L
- $\forall i : \phi_i$ less than or equal to ϕ

- Activities in L_i also occur in L, so that it is a sound sub-log. (i.e $A_i \subset A_L$)
- For the number of sub-logs $n \leq \|L\| + \phi$ must hold

Given this, the termination and fitness of the method can be proven for a generic $select(L)$ [8] (Theorem 2 and 3)

B. Efficient Select

Let us now consider a concrete selection procedure called $select_{B'}(L)$ in [8]:

Using the DFG of a log L, we try to find the dominant operator that orders the behaviour in the graph. A *directed-acylclic graph* for example can always be split meaningfully by a sequence cut. Given the cut and the corresponding operator, we split the log and perform the same procedure on the sub-logs $L_1, \ldots, L_n$, as outlined above. The different cuts explained below closely resemble existing graph algorithms for (strongly) connected components. Given the sub-graphs (i.e. subsets of activities A_i), each cut has its corresponding SPLIT$(L, (A_1, \ldots, A_n))$ function, which maps the log L onto the sub-logs L_i according to the cut specified by $A_1, \ldots, A_n$

We now define the four cuts (defined in [5], [8]) related to the operators defined in II-B. Let G be a DFG and let $A_1, \ldots, A_n$ be sets of activities $A_i \subset A_L$.

Definition 6. (Exclusive Choice Cut) An *exclusive choice cut* is a cut $(\times, (A_1, \ldots, A_n))$ of G such that

$$\forall i, j = 1, \ldots, n \wedge a \in A_i, b \in A_j : i \neq j \Rightarrow a \not\mapsto_L b.$$

Definition 7. (Sequence Cut) A *sequence cut* is an ordered cut $(\rightarrow, (A_1, \ldots, A_n))$ of G such that

$$\forall i, j = 1, \ldots, n \wedge a \in A_i, b \in A_j : i < j \Rightarrow (a \mapsto_L^+ b \not\mapsto_L^+ a),$$

where $a \mapsto_L^+ b$ means that a is followed by b in the log, but not only directly.

Definition 8. (Parallel Cut) A *parallel cut* is a cut $(\wedge, (A_1, \ldots, A_n))$ of G such that

$$\forall i = 1, \ldots, n \wedge A_i \cap A_L^{start} \neq \varnothing, A_i \cap A_L^{end} \neq \varnothing \text{ and}$$
$$\forall i, j = 1, \ldots, n \wedge a \in A_i, b \in A_j : i \neq j \Rightarrow a \mapsto_L^+ b.$$

Definition 9. (Loop Cut) A *loop cut* is a partially ordered cut $(\circlearrowleft, (A_1, \ldots, A_n))$ of G such that

1) $n \geq 2$ and $A_L^{start} \cup A_L^{end} \subseteq A_1$
2) $\{a \in A_1 | \exists i \exists b \in A_i : a \mapsto_L b\}$ $\subseteq A_L^{end}$ similar for A_L^{start}
3) $\forall a \in A_i, b \in A_j : i \neq j \Rightarrow a \not\mapsto_L b$
4) $\forall i = 1, \ldots, n \forall a \in A_i, b \in A_L^{end} : a \mapsto_L b$ $\Rightarrow \forall a' \in A_L^{end} : a' \mapsto_L b$.

Thus, in the concrete B'_{select} (Algorithm 2), we take a log, construct a directly follows graph, search for cuts and create the sub-trees. Using the trees, we can project the original log onto the sub-log that is allowed by the respective sub-tree. This projection is called SPLIT. In this regard, a process tree Q is called *language-rediscoverable* by an inductive miner B, if for any directly-follows complete event log L generated from Q it holds that $\mathcal{L}(B(L)) = \mathcal{L}(Q)$ [5].

Algorithm 2 $select_{B'}(L)$ from [8]

if $\epsilon \in L \vee L = \{\langle a \rangle\}$ for $a \in A_L$ **then**
  return $\varnothing$
if $c \leftarrow$ n.m exclusive-choice cut of $G(L)$ **then**
  $L_1, \ldots, L_n \leftarrow$ ECSPLIT$(L, (A_1, \ldots, A_n))$
  return $\{\times, ((L_1, 0), \ldots, (L_n, 0))\}$
if $c \leftarrow$ n.m sequence cut of $G(L)$ **then**
  $L_1, \ldots, L_n \leftarrow$ SEQUENCESPLIT$(L, (A_1, \ldots, A_n))$
  return $\{\rightarrow, ((L_1, 0), \ldots, (L_n, 0))\}$
if $c \leftarrow$ n.m parallel cut of $G(L)$ **then**
  $L_1, \ldots, L_n \leftarrow$ PARALLELSPLIT$(L, (A_1, \ldots, A_n))$
  return $\{\wedge, ((L_1, 0), \ldots, (L_n, 0))\}$
if $c \leftarrow$ n.m sequence cut of $G(L)$ **then**
  $L_1, \ldots, L_n \leftarrow$ LOOPSPLIT$(L, (A_1, \ldots, A_n))$
  return $\{\circlearrowleft, ((L_1, 0), \ldots, (L_n, 0))\}$
return $\varnothing$

Using these cuts and there respective SPLIT functions we can construct the $select_{B'}(L)$ in Algorithm 2. Write *n.m. cut* as short for nontrivial maximal cut.

Note that the order in which the different cuts in $\bigoplus$ are considered is irrelevant ([8] Lemma 16), but we will see it is handy to consider them in the order depicted above. We can then prove that the properties in List III-A hold, by considering them for a fixed $\phi = 0$ [8]. Note also, that this is has polynomial time complexity, although the generation of the DFG and the cut detection run in linear time [12], [13]. Polynomial complexity comes from recursing on each sub-log returned. Efficiency of inductive mining can be further improved by recursing directly on the DFG rather than on the sub-logs.

C. Finding Cuts in the Directly-Follows Graph

To finish our discussion of *inductive mining*, we consider the details of how one can find the cuts outlined above in a *directly-follows graph* G of a log L using existing graph algorithms for *connected components* (simple breadth search) as well as *strongly connected components* (see [13] for details). For clarity, we do this given an example log $L_2 = \{\langle a, b, c, d \rangle, \langle a, c, b, d \rangle, \langle a, e, d \rangle\}$. Constructing the DFG by simply considering which activity follows which, yields $G_L = G(L_2)$ as shown in Figure 3.

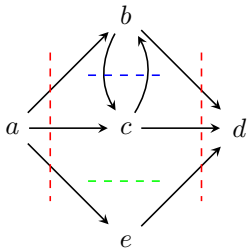

Figure 3: DFG $G_L = G(L_2)$ constructed from L_2

Now we look for cuts in the order $(\times, \rightarrow, \wedge, \circlearrowleft)$. We note that in G_L no *exclusive-choice cut* can be found, because all

nodes are connected. Thus we consider a *sequence cut* and find that the red dashed lines provide a sequence cut. To compute this, one would collapse all strongly connected components and pairwaise unreachable nodes into single nodes, which then represent the sub-graphs of a sequence cut. We now continue on the subgraphs G_{La}, G_{Lbce}, G_{Ld} that are created by spliting G_L at the dashed red lines. Note that G_{La} and G_{Ld} already represent base cases (only one activity) and the recursion finishes there (the sub-tree that $select_{B'}(Ld)$ would return is simply one node $Q_{Ld} = (d)$). On G_{Lbce} we find an *exlusive-choice* cut represented by the green dashed line in Figure 3, yielding sub-graphs G_{Lbc}, G_{Le} of which we again only consider G_{Lbc}. There we finally find a parallel cut, depicted by the blue dashed line. All in all, we found in top-down order the cuts $(\rightarrow, \times, \wedge)$ and no *loop cut*. Thus the inductive miner yields

$$Q_L = B_{select}(L_2, 0) = \rightarrow (a, \times(\wedge(b, c), e), d),$$

or more illustrative in Figure 4.

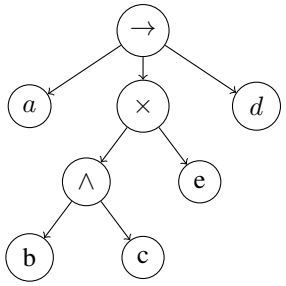

Figure 4: Process tree Q_L mined from L_2 with inductive mining after [8]

Considering again the original log L_2, we see that all traces in the log can be replayed, thus the model is *fit* and per construction also *sound*. That is, execution of Q_L always concludes in the end state d and there are no dead-ends. If the log L_2 was *directly-follows complete* with respect to the original model from which it was drawn, then our tree Q_L is language-equivalent to that model. Mind however, that such an *original model* does not usually exist, since we are mining for the purpose of extracting a new model.

IV. Discussion & Related Work

Like outlined in II-D, process discovery faces some of the common statistical problems that keep occurring because of the nature of the task. More challenges to the field are described in [14]. Some of which are more technical in nature like *noisy data*, *incompleteness* or *visualization of results* while others point to a more abstract epistemological problem. Mining processes, for example, always impose a *representational bias* by limiting the bandwith of models they can possibly discover (*process trees* in our case). Furthermore, process mining as presented here can by nature only consider one perspective of organizational processes, namely the *control-flow perspective*, while other views like the *information-* or *organization perspective* are neglected [14]. Also, one has to keep in mind that as of now, most discovery techniques work only with positive examples. That is, they consider only events found in the log, while neglecting possible negative examples that cannot possibly occur. Having these structural limitations in mind is an essential requirement to continue further work on the topic.

Since [15], many approaches from various fields have been tried and tested to tackle some of the more technical problems mentioned above. Genetic algorithms proved to have very high precision [16], but have impracticable run-times due to their evolutionary approach. Heuristic Mining makes use of statistics to deal with infrequent behaviour (i.e. noise) during the model construction. In [15] also *neural nets* and *Markov models* are considered besides a simpler, algorithmic approach.

In the vast collection of possibilities, *inductive mining* has shown to have desirable properties both in quality (soundness, rediscoverability, normalization) as well as in performance, due to its recursive approach. Most importantly, inductive mining provides a modular framework which can be expanded in the future. For example, normalization can be applied to simplify models post-hoc, like in [17]. Performance can be increased by working on a single DFG rather that projecting the original log back onto sub-logs [7], [12].

To close, inductive mining opens a promising path toward better understanding the bulk of process data that is collected, while doing so in a deterministic and rather straightforward manner.

References

[1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," 2011.
[2] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *science*, p. 1200970, 2011.
[3] D. J. Hand, "Principles of data mining," *Drug Safety*, vol. 30, pp. 621–622, Jul 2007.
[4] J. Jeston, *Business process management*. Routledge, 2014.
[5] Wil van den Aalst, *Process Mining*. Springer, Berlin, Heidelberg, 2016.
[6] M. Rosemann, "Potential pitfalls of process modeling: Part a," *Business Process Management Journal*, vol. 12, pp. 377–384, 05 2006.
[7] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Scalable process discovery with guarantees," in *International Conference on Enterprise, Business-Process and Information Systems Modeling*, pp. 85–101, Springer, 2015.
[8] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Discovering block-structured process models from event logs-a constructive approach," in *International conference on applications and theory of Petri nets and concurrency*, pp. 311–329, Springer, 2013.
[9] B. F. Van Dongen, J. Mendling, and W. M. van der Aalst, "Structural patterns for soundness of business process models," in *null*, pp. 116–128, IEEE, 2006.
[10] D. B. West *et al.*, *Introduction to graph theory*, vol. 2. Prentice hall Upper Saddle River, 2001.
[11] J. C. Buijs, B. F. van Dongen, and W. M. van der Aalst, "A genetic algorithm for discovering process trees," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pp. 1–8, IEEE, 2012.
[12] J. Evermann, "Scalable process discovery using map-reduce," *IEEE Transactions on Services Computing*, vol. 9, no. 3, pp. 469–481, 2016.
[13] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
[14] W. M. Van der Aalst and A. Weijters, "Process mining: a research agenda," 2004.
[15] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 3, pp. 215–249, 1998.
[16] W. van der Aalst, J. Buijs, and B. van Dongen, "Improving the representational bias of process mining using genetic tree mining," *SIMPDA 2011 Procedings*, 2011.

[17] D. Fahland and W. M. Van Der Aalst, "Simplifying discovered process models in a controlled manner," *Information Systems*, vol. 38, no. 4, pp. 585–605, 2013.