Scalable Process Discovery with Guarantees

Sander J.J. Leemans, Dirk Fahland, and Wil M.P. van der Aalst

Eindhoven University of Technology, the Netherlands
{s.j.j.leemans, d.fahland, w.m.p.v.d.aalst}@tue.nl

Abstract Considerable amounts of data, including process event data, are collected and stored by organisations nowadays. Discovering a process model from recorded process event data is the aim of process discovery algorithms. Many techniques have been proposed, but none combines scalability with quality guarantees, e.g. can handle billions of events or thousands of activities, and produces sound models (without deadlocks and other anomalies), and guarantees to rediscover the underlying process in some cases. In this paper, we introduce a framework for process discovery that computes a directly-follows graph by passing over the log once, and applying a divide-and-conquer strategy. Moreover, we introduce three algorithms using the framework. We experimentally show that it sacrifices little compared to algorithms that use the full event log, while it gains the ability to cope with event logs of 100,000,000 traces and processes of 10,000 activities.

Keywords: big data, scalable process mining, block-structured process discovery, directly-follows graphs, rediscoverability

1 Introduction

Considerable amounts of data are collected and stored by organisations nowadays. For instance, ERP systems log business transaction events; high tech systems such as X-ray machines record software and hardware events; and web servers log page visits. These event logs are often very large [5], i.e. contain billions of events. From these event logs, process mining aims to extract information, such as compliance to rules and regulations; and performance information, e.g. bottlenecks in the process [3].

Figure 1 shows a typical process mining workflow. The system executes and produces an event log. From this log, a *process discovery technique* obtains a model of the process. As a next step, the log is filtered: the behaviour in the log that does not match with the model (typically 20%) is analysed manually, while the behaviour that matches with the log is analysed with respect to the model, for instance by selecting interesting parts and drilling down on it using filtering, after which a new model is discovered.

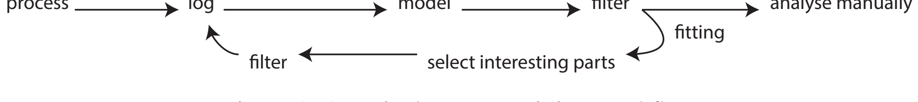


Figure 1: A typical process mining workflow.

Process mining is usually studied in small settings, for instance on processes containing a few activities (i.e. process steps) and a few thousand traces (i.e. cases; customers), e.g., the BPI challenge log of 2012 [14] contains 36 activities and around 13,000 traces. In this paper, we explore a first step towards process mining on bigger scales: we experimented using logs containing 100,000,000 traces and processes containing 10,000 activities. While such numbers seem large for a complaint-handling process in an airline, even much larger processes exist, for instance in control systems of the Large Hadron Collider [18]. Current process discovery techniques *do not scale well* when facing big event logs (i.e. containing billions of events), when facing event logs of complex systems (i.e. containing thousands of activities), or *do not provide basic, essential guarantees*, such as deadlock-freedom or the ability to rediscover the original process.

Given a big or complex log, several strategies could be of help: 1) sampling or drilling down beforehand using process cubes might reduce logs to sizes manageable for existing discovery techniques; 2) restrictions or guarantees might be dropped; 3) taylor-made algorithms could be used. In this paper, we focus on situations in which the first two options are not possible, such as if the process consists of 10,000 activities, (because of which sampling to manageable sizes would throw away too much information); or if the use case prefers taking the full log into account, such as in auditing [2]. In this paper, we push the boundary of how far we can get while taking all recorded behaviour into account.

Event logs with billions of events challenge current discovery algorithms, as most techniques require that the event log is present in main memory. If an event log is too big to fit into main memory of a single machine, few algorithms can be used. Ideally, such a discovery algorithm should require only a *single pass through the event log*, and its run time after this pass should be constant and not depend on the number of events and traces in the event log. The single-pass property would cancel the need for the event log to be in main memory; the independence of run time to the number of events and traces would ensure that logs containing billions of events can still be handled. Independence of the number of activities cannot be achieved, as activities are part of the output of any technique.

Single-pass algorithms should produce models that are as close as possible to other techniques in terms of quality criteria. In the context of process mining, several criteria exist that describe the quality of a process model, such as fitness, precision, generalisation and simplicity, of which the first three are measured with respect to an event log. Fitness describes what part of the event log is represented by the model, precision describes what part of the model is present in the log, generalisation expresses a confidence that future behaviour of the process will be representable by the model, and simplicity expresses whether a process model requires few constructs to express its behaviour. Any discovery algorithm needs to balance these sometimes conflicting four criteria [10]. However, a more basic quality criteria is whether the model is *sound*, which means it contains neither deadlocks nor other anomalies. While an unsound model can be useful for human interpretation, it is often unsuitable for further (automated) analysis, including the computation of fitness, precision and generalisation [20].

A desirable property of discovery techniques is *rediscoverability*: Rediscoverability measures the ability of a technique to find the actual process that produced the partial observations in the event log. Typically, rediscoverability is proven assuming that the log contains enough information (is *complete*), and assuming that the process adheres to some restrictions [19,8].

In this paper, we investigate how current techniques perform in big-data settings and show what happens to several discovery algorithms if they are adapted to the single-pass property. We adapt the Inductive Miner framework (IM) [19] to recurse on directly-follows graphs rather than on logs. Directly-follows graphs can be computed in a single pass over the event log, and their computation can even be parallelised, for instance using highly-scalable map-reduce techniques [15]. Moreover, the size of a directly-follows graph only depends quadratically on the number of activities (i.e, types of process steps) in a log, and is independent of the number of traces or events. We show that this adaptation combines the scalability of directly-follows graphs, as used by the Heuristics Miner (HM) [25] and the α-algorithm (α) [6], with guarantees such as soundness and rediscoverability as provided by the IM framework. The adapted framework is called the *Inductive Miner - directly-follows based* (IMD framework).

We use the new framework in three algorithms: we introduce a basic algorithm, an algorithm focused on infrequent behaviour handling and an algorithm focused on handling incomplete behaviour, similar to the algorithms developed for the IM framework. These algorithms are implemented as plug-ins of the ProM framework, and are available for download at http://www.promtools.org.

To show how the use of directly-follows graphs influences the adapted framework and its algorithms, we test their suitability in a big data context where we deliberately create larger and larger logs, until reaching a log size that cannot be handled anymore, or it becomes clear it poses no restrictions on the log size (at 100 million traces). As the new algorithms have less information available than in the IM framework, we expect that they will need more information for rediscoverability and infrequent behaviour handling. Therefore, we evaluate this trade-off as well. It turns out we can get significant performance and scalability improvements, i.e. the feasibility of handling event logs with over 3.000.000.000 events (>200GB), while producing similar process models as existing non-scalable techniques.

Outline. First, related work is discussed in Section 2. Second, process trees, directly-follows graphs and cuts are introduced in Section 3. In Section 4, the framework and three algorithms using it are introduced. The algorithms using the framework are evaluated in Section 5. Section 6 concludes the paper.

2 Related Work

Process discovery and process discovery in highly scalable environments have been studied before. In this section, we discuss related process discovery techniques and their application in scalable environments.

Process Discovery. Process discovery techniques such as the Evolutionary Tree Miner (ETM) [9], the Constructs Competition Miner (CCM) [23] and Inductive Miner (IM) [19] provide several quality guarantees, in particular soundness and some offer rediscoverability, but do not manage to discover a model in a single pass. ETM applies a genetic strategy, i.e. generates an initial population, and then applies random crossover steps, selects the 'best' individuals from the population and repeats. While ETM is very flexible towards the desired quality criteria to which respect the model should be 'best' and guarantees soundness, it requires multiple passes over the event log and does not provide rediscoverability.

CCM and IM use a divide-and-conquer strategy on event logs. In the Inductive Miner IM framework, first an appropriate cut of the process activities is selected; second, that cut is used to split the event log into sub logs; third, these sub logs are recursed on, until a base case is encountered. If no appropriate cut can be found, a fall-through ('anything can happen') is returned. CCM works similarly by having several process constructs compete with one another. While both CCM and the IM framework guarantee soundness and IM guarantees rediscoverability, both require multiple passes through the event log (the event log is being split and recursed on).

Scalability. Several techniques exist that satisfy the single-pass requirement, for instance the α-algorithm (α) and its derivatives [6,26,27], and the Heuristic Miner (HM) [25]. These algorithms first obtain an abstraction from the log, which denotes what activities directly follow one another; in HM, this abstraction is filtered. Second, from this abstraction a process model is constructed. Both α and HM have been demonstrated to be applicable in highly-scalable environments: event logs of 5 million traces have been processed using map-reduce techniques [15]. Moreover, α guarantees rediscoverability, but neither α nor HM guarantees soundness. We show that our approach offers the same scalability as HM and α, but provides both soundness and rediscoverability.

Some commercial tools such as Fluxicon Disco (FD) [16] and Perceptive Process Mining (PM) offer high scalability, but have no executive semantics (FD) or do not support parallelism (PM) [22].

Other well-known discovery techniques such as the ILP miner [28] satisfy neither of the two requirements.

Streams. Another set of approaches that aims to handle even bigger (i.e. unbounded) logs assumes that the event log is an infinite stream of events. Some approaches such as [13,17] work on click-stream data, i.e. the sequence of web pages users visit, to extract for instance clusters of similar users or web pages. However, we aim to extract end-to-end process models, in particular containing parallelism. HM, α and CCM have been shown to be applicable in streaming environments [11,24]. While streaming algorithms could handle event logs containing billions of events by feeding them as streams, these algorithms assume the log can never be examined completely and, as of the unbounded stream, eventually cannot store information for an event without throwing away information about an earlier seen event. In this paper, we assume the log is bounded and we investigate how far we can get using all information in it.

3 Process Trees, Directly-Follows Graphs

Our approach combines the single-pass property of directly-follows graphs with a divide-and-conquer strategy. This section recalls these existing concepts.

An *event log* is a multiset of *traces* that denote process executions. For instance, the event log $[\langle a,b,c\rangle,\langle b,d\rangle^2]$ denotes the event log in which the trace consisting of a followed by b followed by c was executed once, and the trace consisting of b followed by d was executed twice.

A *process tree* is an abstract representation of a block-structured hierarchical process model, in which the leaves represent the *activities*, i.e. the basic process steps, and the *operators* describe how their children are to be combined [9]. τ denotes the activity which execution is not visible in the event log. We consider four operators: $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$. $\times$ describes the exclusive choice between its children, $\rightarrow$ the sequential composition and $\wedge$ the parallel composition. The first child of a loop $\circlearrowleft$ is the *body* of the loop, all other children are *redo* children. First, the body must be executed, followed by zero-or-more times a redo child and the body again. For instance, the language of the process tree $\times(\circlearrowleft(a,b),\rightarrow(\wedge(c,d),e))$ is $\{\langle a\rangle,\langle a,b,a\rangle,\langle a,b,a,b,a\rangle\ldots\langle c,d,e\rangle,\langle d,c,e\rangle\}$. Process trees are inherently sound.

A *directly-follows graph* can be derived from a log and describes what activities follow one another directly, and with which activities a trace starts or ends. In a directly-follows graph, there is an edge from an activity a to an activity b if a is followed directly by b. The weight of an edge denotes how often that happened. For instance, the directly-follows graph of our example log $[\langle a,b,c\rangle,\langle b,d\rangle^2]$ is shown in figure 2. Notice that the multiset of start activities is $[a,b^2]$ and the multiset of end activities is $[c,d^2]$. A directly-follows graph can be obtained in a single pass over the event log with minimal memory requirements [15].

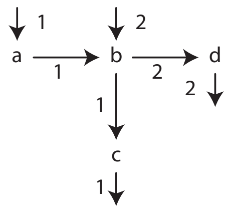


Figure 2: Example of a directly-follows graph.

A *partition* is a non-overlapping division of the activities of a directly-follows graph. For instance, $(\{a,b\},\{c,d\})$ is a binary partition of the directly-follows graph in Figure 2. A *cut* is a partition combined with a process tree operator, for instance $(\rightarrow,\{a,b\},\{c,d\})$. In the IM framework, finding a cut is an essential step: its operator becomes the root of the process tree, and its partition determines how the log is split.

Suppose that the log is produced by a process which can be represented by a process tree T. Then, the root of T leaves certain characteristics in the log and in the directly-follows graph. The most basic algorithm that uses the IM framework, i.e. IM [19], searches for a cut that matches these characteristics perfectly.

Each of the four process tree operators $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$ leaves a different characteristic footprint in the directly-follows graph. Figure 3 visualises these characteristics: for exclusive choice, each trace will be generated by one child; so we expect several unconnected clusters in the directly-follows graph. For sequential behaviour, each child generates a trace; in the directly-follows graph we expect to see a chain of clusters without edges going back. For parallelism, each child generates a trace and these traces can occur in any intertwined order; we expect all possible connections to be present between the child-clusters in the directly-follows graph. In a loop, the directly-follows graph must contain a clear set of start and end activities; all connections between clusters must go through these activities. For more details, please refer to [19].

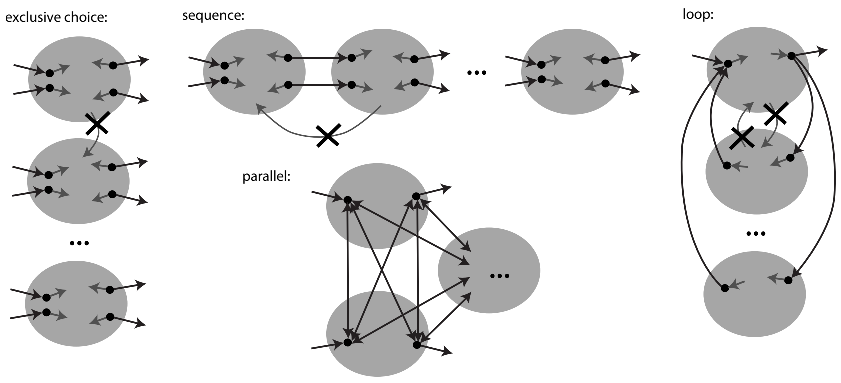


Figure 3: Cut characteristics.

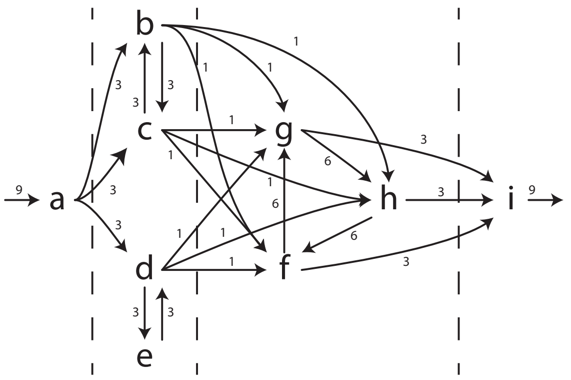


Figure 4: Directly-follows graph D_1 of L. In a next step, the partition $(\{a\}, \{b, c, d, e\}, \{f, g, h\}, \{i\})$, denoted by the dashed lines, will be used.

4 Process Discovery Using a Directly-Follows Graph

Algorithms using the IM framework guarantee soundness, and some even rediscoverability, but do not satisfy the single-pass property, as the log is traversed and even copied during each recursive step. Therefore, we introduce an adapted framework: *Inductive Miner - directly-follows based* (IM framework) that applies the recusion on the directly-follows graph directly. In this section, we first introduce the IMD framework and a basic algorithm using it. Second, we introduce two more algorithms: one to handle infrequent behaviour; another one that handles incompleteness.

The input of the IMD framework is a directly-follows graph, and it applies a divide-and-conquer strategy: first, a cut of the directly-follows graph is selected. Second, using this cut the directly-follows graph is split, which yields several smaller directly-follows graphs. Third, the framework recurses on these smaller directly-follows graphs until a base case is encountered. If no cut can be found, a fall-through, i.e. a model that allows for all behaviour, is applied. The returned process model is the hierarchical composition of operators, base cases and fall-throughs.

4.1 Inductive Miner - directly-follows based

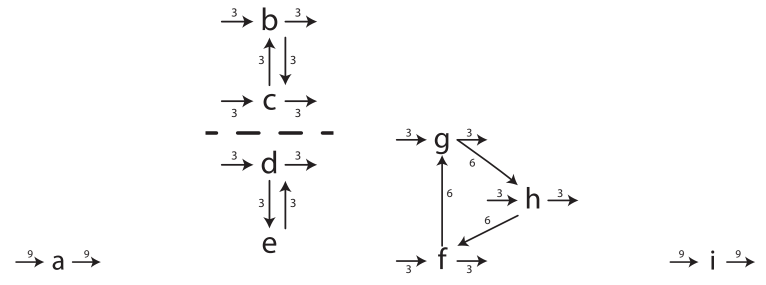
As a first algorithm that uses the framework, we introduce Inductive Miner - directly-follows based (IMD). We explain the stages of IMD in more detail by means of an example: Let L be $[\langle a, b, c, f, g, h, i\rangle, \langle a, b, c, g, h, f, i\rangle, \langle a, b, c, h, f, g, i\rangle,$ $\langle a, c, b, f, g, h, i\rangle, \langle a, c, b, g, h, f, i\rangle, \langle a, c, b, h, f, g, i\rangle, \langle a, d, f, g, h, i\rangle,$ $\langle a, d, e, d, g, h, f, i\rangle, \langle a, d, e, d, e, d, h, f, g, i\rangle]$. The directly-follows graph D_1 of L is shown in Figure 4.

Cut Detection. IMD searches for a cut that perfectly matches the characteristics mentioned in Section 3. Cut detection has been implemented using standard graph algorithms (connected components, strongly connected components) [19], which run in linear time, given the number of activities and directly-follows edges in the graph.

In our example, the cut $(\rightarrow, \{a\}, \{b, c, d, e\}, \{f, g, h\}, \{i\})$ is selected: as shown in Figure 3, every edge crosses the cut lines from left to right. Therefore, it perfectly matches the sequence cut characteristic. Using this cut, the sequence is recorded and the directly-follows graph can be split.

Directly-Follows Graph Splitting. Given a cut, the IMD framework splits the directly-follows-graph in disjoint subgraphs. The idea is to keep the internal structure of each of the clusters of the cut by simply projecting a graph on the cluster. Figure 5 gives an example of how D_1 (Figure 4) is split using the sequence cut that was discovered in our example. If the operator of the cut is $\rightarrow$ or $\circlearrowleft$, the start and end activities of child might be different from the start and end activities of its parent. Therefore, every edge that enters a cluster is counted as a start activity, and edges leaving a cluster are counted as end activities. In our example, the cluster $\{f, g, h\}$ gets as start activities all edges that are entering the cluster $\{f, g, h\}$ in D_1; similar for all end activities. The result is shown in Figure 5a. In case the operator of the cut is $\times$ or $\wedge$, traces that are crossing cluster boundaries do not cause a child to start or end.

The choice for a sequence cut and the split directly-follows graphs are recorded in an intermediate tree: $\rightarrow((D_2), (D_3), (D_4), (D_5))$, denoting a sequence operator with 4 unknown sub-trees that are to be derived from 4 directly-follows graphs.



(a) D_2 of $\{a\}$ (b) D_3 of $\{b, c, d, e\}$ (c) D_4 of $\{f, g, h\}$ (d) D_5 of $\{i\}$

Figure 5: Split directly-follows graphs of D_1. The dashed line is used in a next step and denotes another partition.

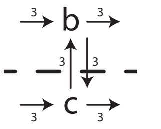
Recursion. Next, IMD recurses on each of the new directly-follows graphs (find cut, split, ...) until a base case (see below) is reached or no perfectly matching cut can be found. Each of these recursions returns a process tree that is inserted as a child.

Base Case. Directly-follows graphs D_2 (Figure 5a) and D_5 (Figure 5d) contain base cases: in both graphs, only a single activity is left. The algorithm turns these into leaves of the process tree and inserts them at the respective spot of the parent operator. In our example, detecting the base cases of D_2 and D_5 yields the intermediate tree $\rightarrow(a, (D_3), (D_4), i)$, in which D_3 and D_4 indicate directly-follows graphs that are not base cases and will be recursed on later.

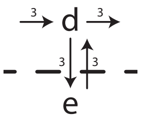
Fall-Through. Consider D_4 as, shown in Figure 5c. D_4 does not contain unconnected parts, so does not contain an exclusive choice cut. There is no sequence cut possible, as f, g and h form a strongly connected component. There is no parallel cut as there are no dually connected parts, and no loop cut as all activities are start and end activities. Hence, IMD selects a fall-through, being a process tree that allows for any behaviour consisting of f, g and h. The intermediate tree of our example up till now becomes $\rightarrow(a, (D_3), \circlearrowleft(\tau, f, g, h), i)$ (remember that τ denotes the activity which execution is invisible).

Example Continued. In D_3, shown in Figure 5b, a cut is present: $(\{b, c\}, \{d, e\})$. No edge in D_3 crosses this cut, hence this is an exclusive choice cut. The directly-follows graphs D_6 and D_7, shown in Figures 6a and 6b, result after splitting D_3. The tree of our example up till now becomes $\rightarrow(a, \times((D_6), (D_7)), \circlearrowleft(\tau, f, g, h), i)$.

In D_6, shown in Figure 6a, a parallel cut is present, as all possible edges cross the cut, i.e. the dashed line, in both ways. The dashed line in D_7 (Figure 6b) denotes a loop cut, as all connections between $\{d\}$ and $\{e\}$ go via the set of start and end activities $\{d\}$. Four more base cases give us the complete process tree $\rightarrow(a, \times(\wedge(b, c), \circlearrowleft(d, e)),$ $\circlearrowleft(\tau, f, g, h), i)$.



(a) D_6 of $\{b, c\}$ in D_3



(b) D_7 of $\{d, e\}$ in D_3

Figure 6: Split directly-follows graphs. Dashed lines denote cuts, which are used in the next steps.

To summarise: IMD selects a cut, splits the directly-follows graph and recurses until a base case is encountered or a fall-through is necessary. As each recursion removes at least one activity from the graph and cut detection is $O(n^2)$, IMD runs in $O(n^3)$, in which n is the number of activities in the directly-follows graph.

By the nature of process trees, the returned model is sound. By reasoning similar to IM [19], IMD guarantees rediscoverability on the same class of models, i.e. assuming that the model is representable by a process tree without using duplicate activities, and it is not possible to start loops with an activity they can also end with [19]. This makes IMD the first single-pass algorithm to offer these guarantees.

4.2 Handling Infrequency and Incompleteness

The basic algorithm IMD guarantees rediscoverability, but, as will be shown in this section, is sensitive to both infrequent and incomplete behaviour. To solve this, we introduce two more algorithms using the IMD framework.

Infrequent Behaviour. Infrequent behaviour in an event log is behaviour that occurs less frequent than 'normal' behaviour, i.e. the exceptional cases. For instance, most complaints sent to an airline are handled according to a model, but a few complaints are so complicated that they require ad-hoc solutions. This behaviour could be of interest or not, which depends on the use case.

Consider again directly-follows graph D_3, shown in Figure 5b, and suppose that there is a single directly-follows edge added, from c to d. Then, $(\times, \{b, c\}, \{d, e\})$ is not a perfectly matching cut, as with the addition of this edge the two parts $\{b, c\}$ and $\{d, e\}$ became connected. Nevertheless, as 9 traces showed exclusive-choice behaviour and only one did not, this single trace is probably an outlier and in most cases, a model ignoring this trace would be preferable.

To handle these infrequent cases, we apply a strategy similar to IMI [20] and introduce an algorithm using the IMD framework: Inductive Miner - infrequent - directly-follows based (IMID). Infrequent behaviour introduces edges in the directly-follows graph that violate cut requirements. As a result, a single edge makes it impossible to detect an otherwise very strong cut. To handle this, IMID first searches for existing cuts as described in Section 4.1. If none is found (when IMD would select a fall through), the graph is filtered by removing edges which are infrequent with respect to their neighbours. Technically, for a parameter $0 \leqslant h \leqslant 1$, we keep the edges (a, b) that occur more than $h \times \max_c(|(a, c)|)$, i.e. occur frequently enough compared to the most occurring outgoing edge of a. Start and end activities are filtered similarly.

Incompleteness. A log in a "big-data setting" can be assumed to contain lots of behaviour. However, we only see example behaviour and we cannot assume to have seen all possible traces, even if we use the rather weak notion of directly-follows completeness [21] as we do here. Moreover, sometimes smaller subsets of the log are considered, for instance when performing slicing and dicing in the context of process cubes [4]. For instance, an airline might be interested in comparing the complaint handling process for several groups of customers, to gain insight in how the process relates to age, city and frequent-flyer level of the customer. Then, there might be combinations of age, city and frequent-flyer level that rarely occur and the log for these customers might contain too little information.

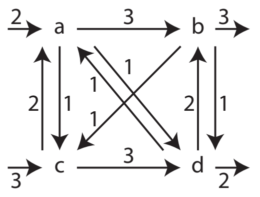


Figure 7: An incomplete directly-follows graph.

If the log contains little information, edges might be missing from the directly-follows graph and the underlying real process might not be rediscovered. Figure 7 shows an example: the cut $(\{a, b\}, \{c, d\})$ is not a parallel cut as the edge (c, b) is missing. As the event log only provides example behaviour, it could be that this edge is possible in the process, but has not been seen yet. Given this directly-follows graph, IMD can only give up and return a fall-through flower model, which yields a very imprecise model. However, choosing the parallel cut $(\{a, b\}, \{c, d\})$ would obviously be a better choice here, providing a better precision.

To handle incompleteness, we introduce Inductive Miner - incompleteness - directly-follows based (IMIND), which adopts ideas of IMIN [21] into the IMD framework. IMIND first applies the cut detection of IMD and searches for a cut that perfectly matches a characteristic. If that fails, instead of a perfectly matching cut, IMIND searches for the most probable cut of the directly-follows graph at hand.

IMIND does so by first estimating the most probable behavioural relation between any two activities in the directly-follows graph. In Figure 7, a and b are most likely in a sequential relation as there is an edge from a to b. a and c are most likely in parallel as there is a forth and a back edge. Loops and choices have similar local characteristics. For each pair of activities x and y the probability $P_r(x, y)$ that x and y are in relation R is determined. The best cut is then a partition into sets of activities X and Y such that the average probabilities that $x \in X$ and $y \in Y$ are in relation R is maximal. For more details, please refer to [21].

In our example, the probability of cut $(\wedge, \{a, b\}, \{c, d\})$ is the average probability that (a, c), (a, d), (b, c) and (b, d) are parallel. IMIND chooses the cut with highest probability, using optimisation techniques. This approach gives IMIND a run time exponential in the number of activities, but still requires a single pass over the event log.

5 Evaluation

In this section, we illustrate handling of big event logs and complex systems, and we study the possible losses of quality of the resulting model. The new algorithms were compared to various existing discovery algorithms in two experiments: First, we test their scalability in handling big event logs and complex systems, and second we test their ability to handle infrequent behaviour. The new algorithms were implemented as plug-ins of the ProM framework, taking as input a directly-follows graph. In order to obtain these, we used an ad-hoc script that builds a directly-follows graph from an event log incrementally.

5.1 Scalability

First, we compare the IMD algorithms with several process discovery tools and algorithms in their ability to handle big event logs and complex systems using limited main memory.

Setup. This experiment searches for the largest event log that a process discovery algorithm currently can handle. All algorithms are tested on the same set of XES event logs, which are created randomly from a process tree of 40 activities. First, each algorithm gets as input a log of t traces generated from the model. The algorithm can handle this log if it returns a model using only the allocated memory of 2GB, i.e. it terminates and does not crash. The ad-hoc pre-processing step was not exempted from this restriction. If the algorithm is successful, t is multiplied by 10 and the procedure is repeated. The maximum number of traces that an algorithm can handle is recorded. This procedure (A) is repeated for a process tree of 10,000 activities (B). For the algorithms implemented as ProM plug-ins, the logs are imported using a disk-buffered import plug-in; enough SSD disk space is available for this plug-in. Algorithms are restarted between all runs to release all memory.

Besides the new algorithms introduced in this paper, from the ProM 6.4.1 framework we included IM, IMI, IMIN, α, HM and ILP in the comparison. From the PM-LAB suite [12], we included the IMMEDIATELY_FOLLOWS_CNET_FROM_LOG (P-IF)

Table 1: Scalability results. #: max traces/events/activities an algorithm could handle.

	A: tree of 40 activities		B: tree of 10,000 activities		
	#traces	#events	#traces	#events	#activities
α	10,000	522,626	1	420	141
HM	100,000	5,218,594	1	420	141
ILP	1,000	51,983	1	420	141
P-IF	10,000	522,626	*1	420	141
P-PT	10,000	522,626	*1	420	141
IM	100,000	5,218,594	100	82689	6941
IMD	100,000,000	3,499,987,460	100,000	76,793,937	10,000
IMI	100,000	5,218,594	10	5886	2053
IMID	100,000,000	3,499,987,460	100,000	76,793,937	10,000
IMIN	100,000	5,218,594	*10	5886	2053
IMIND	100,000,000	3,499,987,460	*10	5886	2053

and the PN_FROM_TS (P-PT) functions. To obtain the directly-follows graphs, a one-pass pre-processing step is executed before applying the actual discovery.

Results. Table 1 shows the results. Some results could not be obtained; they are denoted with *; (for instance, IMIN and IMIND ran for over a week). The largest log we could generate was 217GB (100,000,000 traces) for A, limited by disk space; and 100,000 traces for B, limited by RAM needed for log-generation.

For A, many implementations (HM, IM, IMI, IMIN, P-IF and P-PT) are limited by their requirement to have the complete log in main memory: the ProM disk-caching importer (HM, IM, IMI, IMIN) could handle 100,000 traces, the PMLAB importer (P-IF, P-PT) 10,000. This shows the value of the single-pass property: for a single-pass algorithm, there is no need to import the log. In [15], a single-pass version of HM and α are described. We believe such a version of HM could be memory-restricted, but still this would not offer any of the guarantees described. Of the IM framework, single-pass versions cannot exist due to the necessary log splitting. The IMD framework algorithms are clearly not limited by log size.

For B, log importers were not a problem. Algorithms that are exponential in the number of activities (IMIN, IMIND, α, HM) clearly show their limitations here: none of them could handle our log of 100 traces / 6941 activities. This experiment clearly shows the limitiations of sampling: by sampling a log to a size manageable for other algorithms, many activities are removed, making the log incomplete. In fact, IMD and IMID were the only algorithms that could handle the 10,000 activities.

Timewise, our ad-hoc pre-processing step on the log of 100,000,000 traces (A) took a few days, after that mining was a matter of seconds; P-IF, α, HM, IM, IMI and IMIN took a few minutes; P-PT took days, ILP a few hours.

This experiment clearly shows the scalability of the IMD framework. A manual inspection revealed that IMD and IM algorithms always returned the same model for A once the log was complete.

5.2 Infrequent Behaviour

The goal of the second experiment is investigate the loss of quality faced by the algorithms of the IMD framework compared to the IM framework. We do this with two use cases in mind. First use case is to obtain a model describing almost all behaviour, i.e. having a fitness close to 1.0, preferably with good precision and generalisation. Second use case is, if precision and generalisation make the user consider all found 100% models to be unacceptable, to obtain a model that is accurate for 80% of the log, i.e. having a fitness of around 0.8, that represents the main flow of the process and allows for classification and separate analysis of outliers. Current evaluation techniques force us to perform this experiment on rather small event logs.

Setup. First, a random tree of 40 activities is generated. Second, from this tree a log of 1000 random traces is generated. Third, we vary the number t of infrequent behaviour traces that are added to the log and do not fit the model. Each infrequent trace is generated by inserting errors at certain decisions points with a probability of 0.2; as a result the infrequent trace does not fit the model. The total number of deviations in the log is recorded. Fourth, a discovery algorithm is applied, and conformance checking is measured using fitness [1], precision and generalisation [7]. This process is repeated for the same algorithms as the infrequency experiments, and increasing t. In this experiment, we compared only algorithms that return process trees, as only on sound models fitness, precision and generalisation can be measured reliably [20]. Simplicity is not measured as by nature of these algorithms all trees contain each activity once.

We performed a similar experiment on a real-life log of a financial institution of the Netherlands [14]. This log, consisting of 36 activities, was split in three parts, containing the activities prefixed by respectively A, O or W. On these logs, fitness, precision and generalisation were measured.

Table 2: Results of infrequent behaviour on a tree of 40 activities. f: fitness, p: precision, g: generalisation. t: # deviating traces. Model most suitable for use case $\underline{1}$ and $\utilde{2}$.

t \log size deviations		0 \1000 0	10 \1010 30	100 \1100 450	1000 \2000 4472
IM		$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ 1.00\ p\ 0.62\ g\ 0.94$	$f\ 1.00\ p\ 0.39\ g\ 0.95$	$f\ \underline{1.00}\ p\ \underline{0.16}\ g\ \underline{0.95}$
IMD		$f\ 1.00\ p\ 0.81\ g\ 0.93$	$f\ \underline{0.94}\ p\ \underline{0.53}\ g\ \underline{0.93}$	$f\ 1.00\ p\ 0.15\ g\ 0.93$	$f\ 1.00\ p\ 0.15\ g\ 0.95$
IMI	0.01	$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ 1.00\ p\ 0.79\ g\ 0.93$	$f\ 0.99\ p\ 0.44\ g\ 0.93$	
IMID	0.01	$f\ 1.00\ p\ 0.81\ g\ 0.93$	$f\ 0.93\ p\ 0.58\ g\ 0.93$	$f\ 1.00\ p\ 0.35\ g\ 0.94$	$f\ 1.00\ p\ 0.15\ g\ 0.95$
IMI	0.05	$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ \underline{1.00}\ p\ \underline{0.83}\ g\ \underline{0.92}$	$f\ \underline{0.96}\ p\ \underline{0.83}\ g\ \underline{0.93}$	
IMID	0.05	$f\ 1.00\ p\ 0.81\ g\ 0.93$	$f\ \utilde{0.93}\ p\ \utilde{0.72}\ g\ \utilde{0.93}$	$f\ \underline{0.96}\ p\ \underline{0.57}\ g\ \underline{0.94}$	
IMI	0.20	$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ 0.98\ p\ 0.81\ g\ 0.92$	$f\ 0.94\ p\ 0.82\ g\ 0.93$	
IMID	0.20	$f\ 0.98\ p\ 0.90\ g\ 0.92$	$f\ 0.89\ p\ 0.62\ g\ 0.93$	$f\ 0.95\ p\ 0.51\ g\ 0.94$	$f\ \underline{0.94}\ p\ \underline{0.46}\ g\ \underline{0.95}$
IMI	0.80	$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ \utilde{0.56}\ p\ \utilde{0.88}\ g\ \utilde{0.86}$	$f\ \utilde{0.61}\ p\ \utilde{0.91}\ g\ \utilde{0.87}$	$f\ \utilde{0.55}\ p\ \utilde{0.69}\ g\ \utilde{0.93}$
IMID	0.80	$f\ \utilde{0.97}\ p\ \utilde{0.91}\ g\ \utilde{0.92}$	$f\ 0.70\ p\ 0.82\ g\ 0.91$	$f\ \utilde{0.66}\ p\ \utilde{0.68}\ g\ \utilde{0.90}$	$f\ \utilde{0.67}\ p\ \utilde{0.62}\ g\ \utilde{0.90}$
IMIN		$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ 1.00\ p\ 0.55\ g\ 0.94$	$f\ 1.00\ p\ 0.29\ g\ 0.96$	
IMIND		$f\ \underline{1.00}\ p\ \underline{0.90}\ g\ \underline{0.92}$	$f\ 0.86\ p\ 0.74\ g\ 0.93$		

Results. The results of the infrequency experiments are shown in Table 2. For some measurements, mining finished but computation of fitness, precision and generalisation failed; these measurements are denoted by empty cells. (As shown in the scalability experiment, discovery algorithms easily handle much larger logs.) For each deviation level, there is an algorithm in both groups that gives a high fitness, i.e. use case 1. As to be expected, IMI and IMID with higher parameter settings are needed when the log has more deviations. IMD usually scores worse in precision than the IM counterparts: IM clearly benefits from the full information in the log. However, IMD sometimes scores slightly better in generalisation.

Using parameter setting .8, IMiD returns models with fairly high precision and generalisation, even reaching levels similar to the best IMi parameter settings, although not achieving similar fitness. Therefore, IMiD at .8 seems to be a good candidate default algorithm to get an 80% model from a log with infrequent behaviour. In a practical use case, this 80% model can be used to separate outliers from main flow, i.e. traces that fit the log and traces that do not. This classification can be used to investigate main flow and outliers using separate techniques, possibly using IMinD to achieve robust results on small logs.

We illustrate the results of the experiments on the real-life logs using two models: Figure 8 shows the Petri net returned by IMi 0.2; Figure 9 the model by IMiD 0.2. This illustrates the different trade-offs made between IM and IMD: these models are very similar, except that for IMi, three activities can be skipped. Without the log, IMiD could not decide to make these activities skippable, which lowers fitness a bit (0.93 vs 1) but increases precision (1 vs 0.89).

To summarise, these experiments show that the IMD family is able to adequately handle logs with infrequent behaviour in different use cases, and can achieve results similar to IM with only minor losses in quality (except for the 100 deviations log).

6 Conclusion

Process discovery aims to obtain process models from event logs. Currently, there is no process discovery technique that works on logs containing billions of events or thousands of activities, and that guarantees both soundness and rediscoverability. In this paper, we pushed the boundary on what can be done with logs containing billions of events.

We introduced the *Inductive Miner - directly-follows based* (IMD) framework and three algorithms using it. The input of the framework is a directly-follows graph, which has been shown to be obtainable in highly-scalable environments, for instance using Map-Reduce style log analyses. The framework uses a divide-and-conquer strategy that splits this graph and recurses on the sub-graphs, until a base case is encountered.

We showed that the memory usage of all algorithms of the IMD framework is independent of the number of traces in the event log considered. In our experiments, the scalability was only limited by the logs we could generate. The IMD framework managed to handle over five billion events, while using 2GB of RAM; some other techniques required the event log to be in main memory and therefore could handle up to 10,000-100,000 traces. Besides performance, we also investigated how the new algorithms compare qualitatively to existing techniques that use more knowledge, but also have higher memory requirements, in a number of practical process mining use cases. We showed that the IMiD algorithms allow to handle infrequent behaviour adequately. In particular, parameter setting .8 allows to obtain an "80% model" of a process which

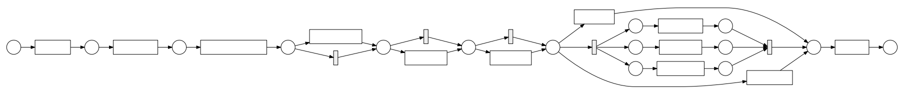


Figure 8: Result without activity names of IMi 0.2 applied to BPIC12_A. $f1.00$, $p0.89$, $g0.99$.

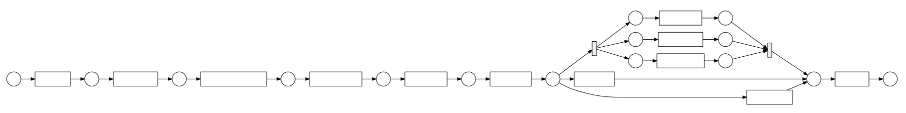


Figure 9: Result without activity names of IMiD 0.2 applied to BPIC12_A. $f0.93, p1.00, g0.99$.

helps to investigate main flow and outliers in more detail, an operation often needed for detailed analysis in process mining projects. Altogether, these results suggest that the IMD family not only handles big event logs, but that its algorithms can be used in a variety of use cases, eliminating the need to operate with different classes of algorithms.

Future Work. In the typical processs mining workflow as described in Section 1, process discovery is just the first step. Further analysis steps can be mostly manual, which makes them hard to perform on logs containing millions of traces. As we encountered during the evaluation, automatic steps are not guaranteed to work on these logs either. We envision further steps in the analysis of models, such as using alignments [7], in contexts of big data.

In streaming environments, it is assumed that events arrive in small intervals and that the log is too big to store, so the run time for each event should be short and ideally constant. The α algorithm and the Heuristics Miner have been applied in streaming environments [11] by using directly-follows graphs; the IMD framework might be applicable as well.

References

1. van der Aalst, W., Adriansyah, A., van Dongen, B.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 2(2), 182–192 (2012)
2. van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M.E.M., Verdonk, M.: Auditing 2.0: Using process mining to support tomorrow's auditor. IEEE Computer 43(3), 90–93 (2010)
3. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
4. van der Aalst, W.: Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In: Asia Pacific Business Process Management. pp. 1–22 (2013)
5. van der Aalst, W.: In: Data Scientist: Enigneer of the Future. I-ESA, vol. 7, pp. 13–26 (2014)
6. van der Aalst, W., Weijters, A., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16(9), 1128–1142 (2004)
7. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.: Alignment based precision checking. In: Business Process Management Workshops. pp. 137–149. Springer (2013)
8. Badouel, E.: On the α-reconstructibility of workflow nets. In: Petri Nets'12. LNCS, vol. 7347, pp. 128–147. Springer (2012)
9. Buijs, J., van Dongen, B., van der Aalst, W.: A genetic algorithm for discovering process trees. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2012)
10. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: OTM. LNCS, vol. 7565, pp. 305–322 (2012)
11. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: IEEE Congress on Evolutionary Computation. pp. 2420–2427 (2014)

12. Carmona, J., Solé, M.: PMLAB: an scripting environment for process mining. In: BPM Demos. CEUR-WP, vol. 1295, p. 16 (2014)
13. Datta, S., Bhaduri, K., Giannella, C., Wolff, R., Kargupta, H.: Distributed data mining in peer-to-peer networks. IEEE Internet Computing 10(4), 18–26 (2006)
14. van Dongen, B.: BPI Challenge 2012 Dataset (2012), http://dx.doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
15. Evermann, J.: Scalable process discovery using map-reduce. In: IEEE Transactions on Services Computing. vol. to appear (2014)
16. Günther, C., Rozinat, A.: Disco: Discover your processes. In: BPM (Demos). CEUR Workshop Proceedings, vol. 940, pp. 40–44. CEUR-WS.org (2012)
17. Hay, B., Wets, G., Vanhoof, K.: Mining navigation patterns using a sequence alignment method. Knowl. Inf. Syst. 6(2), 150–163 (2004)
18. Hwong, Y., Keiren, J.J.A., Kusters, V.J.J., Leemans, S.J.J., Willemse, T.A.C.: Formalising and analysing the control software of the compact muon solenoid experiment at the large hadron collider. Sci. Comput. Program. 78(12), 2435–2452 (2013)
19. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs - a constructive approach. In: Petri Nets 2013. LNCS, vol. 7927, pp. 311–329. Springer (2013)
20. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops. pp. 66–78 (2013)
21. Leemans, S., Fahland, D., van der Aalst, W.: Discovering block-structured process models from incomplete event logs. In: Petri nets 2014. vol. 8489, pp. 91–110 (2014)
22. Leemans, S., Fahland, D., van der Aalst, W.: Exploring processes and deviations. In: Business Process Management Workshops. p. to appear (2014)
23. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Constructs competition miner: Process control-flow discovery of bp-domain constructs. In: BPM 2014. LNCS, vol. 8659, pp. 134–150 (2014)
24. Redlich, D., Molka, T., Gilani, W., Blair, G.S., Rashid, A.: Scalable dynamic business process discovery with the constructs competition miner. In: SIMPDA 2014. CEUR-WP, vol. 1293, pp. 91–107 (2014)
25. Weijters, A., van der Aalst, W., de Medeiros, A.: Process mining with the heuristics miner-algorithm. BETA Working Paper series 166, Eindhoven University of Technology (2006)
26. 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)
27. Wen, L., Wang, J., Sun, J.: Mining invisible tasks from event logs. Advances in Data and Web Management pp. 358–365 (2007)
28. van der Werf, J., van Dongen, B., Hurkens, C., Serebrenik, A.: Process discovery using integer linear programming. Fundam. Inform. 94(3-4), 387–412 (2009)