Efficient Process Model Discovery Using Maximal Pattern Mining

Veronica Liesaputra[1], Sira Yongchareon[1(✉)], and Sivadon Chaisiri[2]

[1] Department of Computing and Information Technology, Unitec Institute of Technology, Auckland, New Zealand
vliesaputra@unitec.ac.nz, sira@maxsira.com

[2] Faculty of Computing and Mathematical Sciences, University of Waikato, Hamilton, New Zealand
sivadon@ieee.org

Abstract. In recent years, process mining has become one of the most important and promising areas of research in the field of business process management as it helps businesses understand, analyze, and improve their business processes. In particular, several proposed techniques and algorithms have been proposed to discover and construct process models from workflow execution logs (i.e., event logs). With the existing techniques, mined models can be built based on analyzing the relationship between any two events seen in event logs. Being restricted by that, they can only handle special cases of routing constructs and often produce unsound models that do not cover all of the traces seen in the log. In this paper, we propose a novel technique for process discovery using *Maximal Pattern Mining* (MPM) where we construct patterns based on the whole sequence of events seen on the traces—ensuring the soundness of the mined models. Our MPM technique can handle loops (of any length), duplicate tasks, non-free choice constructs, and long distance dependencies. Our evaluation shows that it consistently achieves better *precision*, *replay fitness and efficiency* than the existing techniques.

1 Introduction

Process mining has become a promising field of research that helps businesses better understand, analyze, monitor and improve their workflow processes. Process discovery in particular is a core component of process mining that focuses on constructing process models based on the analysis of processes using event log data produced from information systems, such as workflow systems and business process management systems. The discovered process models (e.g., in form of Workflow-net which is a special class of Petri-net), can then be used for conformance checking, auditing, model enhancement, configuring a WFM/BPM system, and process improvement [1].

Since the mid-nineties, several techniques have been proposed to automatically discover process models from event logs in both software processes and business process domains [2, 3, 4]. Several algorithms are variants of the α-algorithm (e.g., in [8, 9, 10, 11]), which can be seen as a well-known technique where process discovery was first studied in the field. Nevertheless, due to the fact that the α-algorithms face problems dealing with complicated routing constructs, noise, and incompletes [1],

©Springer International Publishing Switzerland 2015
H.R. Motahari-Nezhad et al. (Eds.): BPM 2015, LNCS 9253, pp. 441–456, 2015.
DOI: 10.1007/978-3-319-23063-4_29

more advanced techniques, such as region-based approaches (e.g., [21, 22, 25, 28]), heuristic mining [12], fuzzy mining [13], and genetic mining [37], have been proposed to tackle those aforementioned problems.

We argue that the existing algorithms for discovering process models are still unable to efficiently and accurately handle loops (of any length), duplicate tasks, concurrency, long dependencies and complex routing constructs. In fact, some of such algorithms may produce unsound models. To address these problems, we propose a novel process discovery technique called *Maximal Pattern Mining* (MPM). Instead of mining the relationship between two events, MPM mines a set of patterns that could cover all of the traces seen in an event log. We have implemented the algorithm and compared the results with the existing algorithms. Our evaluation shows that the MPM always produces sound process models with better *precision* and *replay fitness*. The processing time of our algorithm to mine and generate a process model is also significantly shorter than all the existing algorithms.

The remainder of the paper is organized as follows. Section 2 reviews and discusses the work that has been done in the process mining area. Section 3 proposes our MPM technique for process discovery and its algorithm. Section 4 discusses a technical evaluation and results. Finally, the conclusion and future works are given in Section 5.

2 Background and Related Work

In this section, we give some background and discuss related work in process mining, especially techniques for process discovery (a.k.a. control-flow discovery). Several discovery techniques have been developed based on algorithmic, machine learning, and probabilistic approaches. Very early process discovery approaches have been proposed by Cook and Wolf [3], Agrawal et al. [2], and Datta [4]. Cook and Wolf proposed RNet, Ktail and Markov software process discovery approach using event-based data based on statistical, algorithmic and probabilistic methods. Agrawal et al. and Datta studied graph-based discovery approaches in the context of workflow processes. Manilla and Meek [5] present a method for finding partial orders that describe the ordering relationships between the events in a collection of sequences and applying mixture modeling techniques to obtain a descriptive set of partial orders. However, their techniques cannot deal with concurrency, decision splits, synchronous and asynchronous joins, and other common issues found in a process mining field. Later, Schimm [6, 7] proposed a procedural approach using data mining techniques to mine a complete and minimal process algoschema from workflow logs that contains concurrent processes. However, the approach is restricted to block-structured processes. Van der Aalst et al. [8] proposed α-algorithm to learn structured workflow nets from complete event logs. However, the α-algorithm cannot cope with noise, incompleteness of workflow logs, short loops, and non-free choice constructs. Later, Alves de Medeiros et al. [9] developed α^{+}-algorithm, an improved version of the α-algorithm, which is capable of detecting short loops. Further, Wen et al. [10, 11] proposed α^{++}-algorithm to discover non-free choice constructs and β-algorithm to detect concurrency. Due to the fact that all the α-algorithms face a robustness problem, Weijters et al. [12] proposed Heuristics Miner by extending the α-algorithm to analyze the frequency of the three types of relationships between activities in a workflow log: direct dependency, concurrency, and not-directly connectedness. It is claimed

that Heuristics Miner is able to mine short loops and non-local dependencies. In contrast to the α-algorithms, Gunther and van der Aalst [13] proposed Fuzzy Miner, an adaptive technique to discover behavior models from an event log using significance and correlation measures. Their technique is capable of mining unstructured processes. Asides from these techniques, Rembert and Omokpo [26] proposed a process discovery technique using the α-algorithm with Bayesian statistics to incorporate prior knowledge supplied by a domain expert to discover control-flow model in the presence of noise and uncertainty.

Herbst and Karagiannis [23] proposed a discovery algorithm to construct a stochastic activity graph and then convert it into a structured process model. Their algorithm can discover duplicate activities but not non-local dependencies. Folino et al. [24] proposed the Enhanced WFMiner algorithm to deal with noise, duplicate tasks and non-free choice. Ferreira and Gillblad [27] proposed a technique to tackle the problem of unlabeled event logs (without a case identifier) by using the Expectation–Maximization procedure. Van der Werf et al. [25] proposed a discovery technique using Integer Linear Programming (ILP) based on the theory of regions. Van der Aalst et al. [21] proposed a Finite State Machine (FSM) Miner/Petrify two-step approach to find a balanced trade-off between generalization and precision of discovered process models. The theory of region is used in their approach as a method to bridge FSM and Petri-Net models as also proposed in [22]. Sole and Carmona [28] presented an aggressive folding region-based technique, which is based on the theory of region, to reduce the total number of states of a transition system and speed up the discovery process.

Several machine leaning techniques have been used in the process discovery domain. Maruster et al. [14] proposed to use propositional rule induction, i.e., a uni-relational classification learner, to predict dependency relationships between activities from event logs that contain noise and imbalance. Ferreira and Ferreira [15] apply a combination of Inductive Logic Programming learning and partial-order planning techniques to discover a process model from event logs. In addition, Lamma et al. [16] applied Inductive Logic Programming to process discovery by assuming negative sequences while searching. Due to the limitations of local search, these approaches were unable to detect non-free choice constructs, duplicate tasks, and hidden tasks. Therefore, in order to discover such constructs, Buijs et al. [37] proposed a genetic algorithm which performs a global search based on the use of alignment fitness function to find the best matched models. Genetic Miner can detect non-local patterns and, due to its post-pruning step, it has a reasonable robustness. Similarly, Goedertier et al. [17] proposed AGNEsMiner to deal with problems such as expressiveness, noise, incomplete event logs, and the inclusion of prior knowledge by representing process discovery as a multi-relational classification problem [18] on event logs supplemented with Artificially Generated Negative Events (AGNEs). This technique can learn the conditions that distinguish between the occurrence of either a positive or a negative event. Furthermore, Greco et al. [19] proposed DWS mining to improve precision. The technique is implemented in an iterative procedure that refines the process model in each step, based on clustering of similar behavior patterns. In [20], Greco et al. proposed an approach for producing taxonomy of workflow models to capture the process behavior at different levels of abstraction by extending traditional discovery methods and an abstraction method.

Based on the above discussions, we have observed that only Genetic Miner [37] can tackle all the typical problems in process mining, i.e., noise, duplicate tasks, hidden tasks, non-free choice constructs, and loops. However, because of the nature of the genetic algorithm, it consumes more processing time and space in order to learn and construct a model. Mining efficiency is considered a major drawback of this approach in which it is undesirable, especially when it is applied to a complicated real-life log. To overcome such issues, we need to develop a better technique that can not only solve all of the typical process mining problems but also takes much less time to process.

3 Maximal Pattern Mining (MPM)

As discussed earlier, the well-known α-algorithm and its variants can be considered the most substantial techniques in the field of process mining [1, 8]. The model was built based on the relationship of an event A with the event's direct predecessors and successors. However, those algorithms have problems with complex control-flow constructs, such as non-free-choice constructs (where concurrency and choice meet), arbitrary nested loops, duplicate tasks, etc.

Bose et al. [33] proposes an algorithm to discover common patterns on events in traces, especially loops. Pattern similarity is calculated by using edit distance. Although their evaluation shows promising results, it was not clear how it would handle other control-flow constructs such as long distance dependencies and duplicate tasks, or how accurate and robust their algorithm is compared to other existing process-mining algorithms.

In this paper we use a similar pattern matching technique called *Maximal Pattern Mining* (MPM) to construct a workflow model. Instead of looking at the relationship between 2 events, we consider the whole sequence of events in all of the traces and find the optimal set of "regular expression"-like patterns that will cover them. Therefore, our algorithm can handle complex constructs such as non-free choice constructs, nested loops of any length (as opposed to short one or two length loops), duplicate tasks and long distance dependencies. It also uses a stricter rule than the edit distance uses in [33] to find similar patterns.

We overview and detail the MPM technique in Sections 3.1 and 3.2. Then, the assumptions and limitations of our technique are discussed in Section 3.3.

3.1 Overview

Let $T = \{t_0, t_1 \ldots t_n\}$ be the collections of all the traces in an event log in which they are ordered by the type of the events in the trace and then by the number of events in the trace. A trace t_n is an ordered sequence of events or completed tasks, $t_n = \langle z_0, z_1 \ldots z_m \rangle$. We denote $|t_n|$ as the number of events in a trace. An event z_m only contain 1 event type, i.e. $|z_m| = 1$. Possible event types include *create*, *schedule*, *assign*, *revoke*, *start*, *addFact*, *removeFact*, *updateFact*, and *complete* [17]. All the traces in T are not unique and a trace t_n may contain particular events more than once, i.e. it is possible to have $T = \{\langle a,b,c,b,b,c,d,e\rangle, \langle a,b,c,b,b,c,d,e\rangle, \langle a,b,b,c,e,d\rangle\}$.

Given an input T, our algorithm will first create a list of unique patterns $P = \{p_0, p_1 \dots p_i\}$ and then generate a graph based on P. The following sections will describe each of them.

A pattern $p_i = \langle e_0, e_1 \dots e_j\rangle$ is an ordered sequence of elements, $|p_i|$ is the number of elements in the pattern and $p_i.support$ is the number of traces covered by the pattern.

An element $e_j = \{v_0, v_1 \dots v_k\}$ contains k number of unique event types (i.e. $|e_j| = k$) and $e_j.loop$ is a list of $\langle v_k: w\rangle$ tuples that indicate whether v_k is self-looping ($w = \{v_k\}$) and/or is the last element of a sequence-loop ($w = \{e_x\ e_{x+1} \dots e_{x+y}\}$ and $e_{x+y} = v_k$). The *loop* list is ordered first by event value and then by the number of elements in w ($|w|$). An element's value v_k only contains 1 event type.

All the elements inside p_i may not be unique. For instance, given the $T = \{\langle a,b,c,b,b,c,d,e\rangle, \langle a,b,c,b,b,c,d,e\rangle, \langle a,b,b,c,e,d\rangle\}$ specified above, our algorithm will only produce 1 pattern in P. $p_0 = \langle e_0, e_1, e_2, e_3\rangle$, where $e_0 = a$ and $e_0.loop = \emptyset$; $e_1 = b$ and $e_1.loop = \{<b: b>\}$; $e_2 = c$ and $e_2.loop = \{\langle c: \{bc\}\rangle\}$; and $e_3 = \{d, e\}$ and $e_3.loop = \emptyset$. Elements with more than one event type indicate a parallelization. In our example, e_3 shows that in the last 2 events of our model the values could be either *de* or *ed*. Because p_0 covers all the traces in T, $p_0.support = 3$.

Our graph algorithm will then generate the following model (Fig. 1) based on p_0. We use the operator AND to indicate the set of tasks that are running at the same time, and XOR to indicate a path selection.

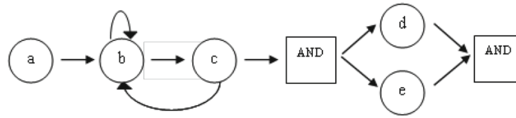


Fig. 1. The generated model for $\{\langle a,b,c,b,b,c,d,e\rangle,\langle a,b,c,b,b,c,d,e\rangle,\langle a,b,b,c,e,d\rangle\}$

The algorithm we use to construct the most optimal patterns for a given trace of events has five main components: finding self and/or sequence loops, storing the pattern in a vertical format, identifying events that should be done concurrently, investigating whether a trace is covered by a pattern in P, and pruning non-maximal patterns.

Loops. A sequence of elements $S = \langle s_0, s_1 \dots s_q\rangle$ is in a loop in the trace $t_n = \langle z_0, z_1 \dots z_m\rangle$ or in the pattern $p_i = \langle e_0, e_1 \dots e_m\rangle$ if and only if there is a sequence of elements such that for all $b \in \{0 \dots q\}$and $q \le (m - a)/2$, $z_{a+b} = s_b$ and $z_{a+q+b} = s_b$ or $e_{a+b} = s_b$ and $e_{a+q+b} = s_b$, where a is the starting index where S occurs in the trace or in the pattern ($0 \le a \le m$). The first phase of our pattern mining is to identify these loops. For every $S+$ occurring in t_n and p_i, we replace it with S and set the *loop* property of the last element in S. For instance, given a pattern *⟨a,b,b,c,d,{e,f},c,d,{e,f}c,d,{e,f}g⟩*, the pattern becomes *⟨a,b,c,d,{e,f}g⟩* where the loop property for *b* is *b*, and the loop property for *{e,f}* is *cd{e,f}*. We keep iterating on the pattern until there are no more loops in the pattern. All the loops in the pattern *⟨a,b,d,d,c,b,b,b,d,c,b,d,c,e⟩* will be identified in 2 iterations: 1) *⟨a,b,d*,c,b*,d,c,b,d,c,e⟩*, 2) *⟨a,(b,d*,c)*,e⟩*. By identifying loops first, MPM will be able to deduce that traces *⟨a,b,d,d,c,b,b,b,d,c,b,d,c,e⟩* and *⟨a,b,d,c,b,d,d,c,e⟩* are the same and are both covered by the pattern *⟨a,b,d,c,e⟩*.

Vertical Representation. Existing process mining algorithms require several scans of the event log or need to maintain large amounts of intermediate candidates in the main memory to generate a process model [10, 11, 37, 18]. To alleviate this problem, MPM stores all patterns in the vertical format as an IdList in bitset representation [31] where each entry represents an element, the id of the trace where the element appears (*id*) and the positions (*pos*) where it appears. The support of a pattern is calculated by making joint operations with IdLists of smaller patterns. Thus, MPM would only need to perform a single scan through the log to generate IdLists of patterns containing single elements (see [31] for details). To make it more verbose, MPM uses the symbol $ to indicate the end of a trace. Given $T = \{\langle a,b,c,b,b,c,d,e,a\rangle,\langle a,b,b,c,e,d,a\rangle,\langle e,d,a\rangle\}$, the vertical representation (V_T) of it is represented as follow:

A	
id	pos
0	0, 5
1	0, 5
2	2

b	
id	pos
0	1
1	1

c	
id	pos
0	2
1	2

D	
id	pos
0	3
1	4
2	1

e	
id	pos
0	4
1	3
2	0

$	
id	pos
0	6
1	6
2	3

Concurrency. A set of events $V = \{v_0, v_1 \dots v_q\}$ are performed at the same time (or parallel) if and only if there are at least q number of unique traces with the following sequence $\langle z_0, z_1 \dots z_{a-1}\ z_a, z_{a+1} \dots z_{a+q}\ z_{a+q+1}, z_{a+q+2} \dots z_m\rangle$, where the sequence $\langle z_0, z_1 \dots z_{a-1}\rangle$ and $\langle z_{a+q+1}, z_{a+q+2} \dots z_m\rangle$ have the same pattern across those traces, there are no events mentioned more than once in $\langle z_a, z_{a+1} \dots z_{a+q}\rangle$, and for all $b \in \{0\dots q\}$and $q \leq (m - a)$, $z_{a+b} \subseteq V$, where a is the starting index where a combination of all the events in V occur ($0 \leq a \leq m$). Sequence $\langle z_0, z_1 \dots z_{a-1}\rangle$ and $\langle z_{a+q+1}, z_{a+q+2} \dots z_m\rangle$ may be $\emptyset$. Instead of $z_{a+b} = V$, we relax the criteria to $z_{a+b} \subseteq V$ with the assumption that if we see almost all of V possible events combination in T, it must just be the case that the trace log is incomplete. For example, given a set of traces *{⟨a,b,c,d,e⟩, ⟨a,b,d,c,e⟩,⟨a,c,d,b,e⟩}*. We first look at the first two traces where we get *⟨a,b,{c,d},e⟩* as it is possible to switch the position of task *c* and *d* around. We then compare it with the last trace where we get *⟨a,{b,c,d},e⟩* as we can switch the position of task *c* and *d* around with *b*. In the future, we may use the trace frequency to help us decide when we should use the strict or relaxed criteria.

Coverage. A pattern $p_i = \langle e_0, e_1 \dots e_n\rangle$ specifies the sequence of patterns that covers some of the traces in T and it can be represented as a deterministic finite automata DFA_i with (a) well-defined start state, (b) one or more accepted states and (c) deterministic transitions across states on symbols of the event values. A trace $t_n = \langle z_0, z_1 \dots z_m\rangle$ is covered by the pattern p_i if and only if the sequence of transitions for the elements of t_n from the start state results in an accept state. Fig. 2 illustrates the deterministic finite automaton for the pattern *⟨a,b,{c,d},e⟩* with the *loop* property for *b* to be *b*. We use > to indicate the start state and double circles for the accept state. The diagram shows that the pattern covers the following set of traces *{⟨a, b, c, d, e⟩, ⟨a, b, d, c, e⟩, ⟨a, b, b, c, d, e⟩, ⟨a, b,…,b, c, d, e⟩, ⟨a, b,…,b, d, c, e⟩}*. However, it will reject the following set of traces *{⟨a,b⟩, ⟨e⟩, ⟨a,b,h⟩, ⟨a,b,c,d⟩, ⟨a,b,b,d,c⟩, ⟨a,b,a,d,c,e⟩}*.

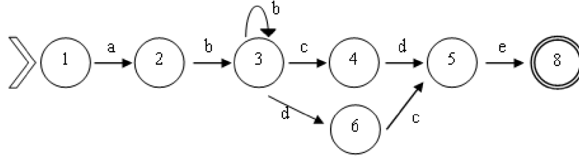


Fig. 2. The deterministic finite automata model for $p_i = \langle a,b,\{c,d\},e\rangle$

Maximal Patterns. A pattern p_i is said to be maximal if and only if there is no other pattern p_j in P that has the same start and accept states and covers the same or more traces in T. Given $P = \{\langle a,b,c,d\rangle,\langle a,\{b,c\},d\rangle,\langle a,b,c\rangle\}$, only p_1 and p_2 are maximal because p_0 is a sub-pattern of p_1.

Optionality. A sequence of elements $S_0 = \langle s_0, s_1 \ldots s_q\rangle$ in the pattern $p_j = \langle z_0 \ldots z_m, s_0, s_1 \ldots s_q, z_p \ldots z_r\rangle$ is in an XOR (optionality) relations with a sequence of elements $S_1 = \langle s'_0, s'_1 \ldots s'_t\rangle$ in the pattern $p_k = \langle z'_0 \ldots z'_n, s'_0, s'_1 \ldots s'_t, z_u \ldots z_v\rangle$ if and only if $z_m = z'_n$ and $z_p = z_u$. In some cases, z_m and z'_n could be a start state, and $z_p = z_u$ could be the accept state. For example, if $P = \{\langle a,b,c,d\rangle,\langle a,e,d\rangle,\langle g,f\rangle,\langle g,h\rangle\}$, the resulted graph will be $XOR(a \rightarrow XOR(b \rightarrow c, e) \rightarrow d, g \rightarrow XOR(f, h))$.

Noise (Frequent Patterns and Events). To further filter P from noisy data, we set a support threshold value, *thresh*, so that we will only keep frequent pattern p_i and event v_k, i.e. $p_i.support \geq thresh$ and $v_k.support \geq thresh$. All patterns and events are accepted if the threshold value is 0. To find out what the best threshold value is, we split the traces that we used for generating the pattern into 2 sets: training and validation. Our MPM algorithm generates maximal patterns only based on the traces found in the training sets. We then evaluate the performance of all the patterns generated by MPM on the traces of events in the validation set. If we are unhappy with the results, we change the threshold value of our algorithm, re-generate the pattern of the training traces based on the new threshold value and evaluate it on the validation traces. We keep doing this until we find the optimal threshold value.

3.2 Generating Maximal Patterns

The pseudo-code of the MPM algorithm is shown in Alg. 1. MPM takes as input an event log (T) and the support threshold value (*thresh*). For each trace t_n, it identifies all the loops, constructs the vertical representation of the log (V_T) and get the set of frequent events (E) as described in Section 3.1. Events in E, except for $, are ordered from the most common to the least. For each event v in E, the procedure calls the EXPAND procedure with $\langle v\rangle$, E and *thresh*.

The EXPAND procedure takes as parameters a sequential pattern (p), a set of items (S) to be appended to p to generate candidates and minimum support value (*thresh*). Each item (s_i) in the set S is appended to p as the next sequential item of p. Each of the newly generated patterns are called p_i'. Because any infrequent sequential patterns cannot be extended to form a frequent pattern, the procedure uses IdList join operation [31] to calculate the number of traces where the pattern p_i' appears. If $p_i'.support \geq thresh$, p_i' is then used in a recursive call to EXPAND to generate patterns starting with p_i'. All the frequent p_i' are passed onto the RESOLVE procedure.

Algorithm 1. The procedure of the Maximal Pattern Mining algorithm

```
MPM (T, thresh)
    V_T = ∅
    FOR each trace t_n ∈ T
        t_n = SOLVE_LOOP (t_n)
        V_T = INSERT_TRACE (V_T, t_n)
    E = GET_FREQUENT_EVENTS (V_T)
    P_T = ∅
    FOR each event v ∈ E
        P_T = P_T ∪ EXPAND (⟨v⟩, E, thresh)
    P_T = RESOLVE (P_T, 0)
    DRAW_GRAPH (P_T)
```

```
EXPAND (p, S, thresh)
    S_T = ∅
    P_T = ∅
    FOR each item s_i ∈ S
        p_i' = p ∪ s_i
        IF p_i'.support ≥ thresh  THEN
            S_T = S_T ∪ s_i
    FOR each item s_i ∈ S_T
        P_T = P_T ∪ EXPAND (p ∪ s_i, S_T, thresh)
    P_T = RESOLVE (P_T, |p|)
    Output P_T
```

```
RESOLVE (FP, idx)
    CP = Copy of FP
    FOR each item p_i ∈ FP
        FOR each item p_j ∈ FP AND i < j ≤ FP.length
            IF p_i [idx] ≠ $ AND p_j [idx] ≠ $ AND
                p_i [idx ... p_i.length] = p_j [idx ... p_j.length] THEN
                p_i = SOLVE_CONCURRENCY (p_i, p_j)
                p_i = SOLVE_LOOP (p_i)
                Remove p_j from FP
            ELSE IF p_j is sub-pattern of p_i  THEN
                Remove p_j from FP
            ELSE IF p_i is sub-pattern of p_j  THEN
                Remove p_i from FP
                Go to the next item in FP
    IF CP ≠ FP  THEN
        RESOLVE (FP, idx)
    Output FP
```

3.3 Assumptions and Limitations

An event in a transactional log usually contains information such as the event type/value (e.g. apply for a drivers licence or update patient information), the agent/performer that is doing the event, the requestor/client that initiates the whole sequence of events, timestamp and the data element being modified or accessed (e.g. the age of a patient, the driving test result). A trace of events is a sequence of events, sorted by the timestamp, done for a client. Because the goal of MPM is to find all possible orderings of the logged events in the system, only the event type/value are mined. Once we have organized the log into sets of traces, other information, such as timestamp and agent, are ignored. In this paper, the term event type and event value are used interchangeably.

In an experimental setting, we know the original model that our algorithm should strive to construct, the complete list of traces that the model could generate, and the instances in a log that are negative examples. But in real life scenarios no original models will be available; logs may contain noise such as mislabelled events and incorrectly logged sequences of events and exceptions. In fact, a particular trace of observed events does not have to be reproduced by the model. Furthermore, in a complex process with many possible paths, only a fraction of those paths may be present in the log, i.e., the log is incomplete. Thus, it is undesirable to construct a model that allows only for the observed instances in the log. Since we do not know which instance in the log is noise, we assume that every trace/event recorded in the log that appears no less than a user's specified threshold frequency is correct (positive examples). However, unobserved traces of events are not considered as negative examples. Our MPM algorithm can construct a model that can explain all the traces of events found in the log while also allowing for any unobserved behaviour.

As shown in Section 4.4., because we are always trying to solve loops before parallel tasks, just like α^{++}, AGNEs and Heuristic Miners, our algorithm is incapable of generating a model that can accurately represents duplicate tasks in a parallel process structure.

4 Experimental Result

In this paper, we evaluate the quality of the mined model produced by MPM, α^{++}, Genetic miner, Integer Linear Programming (ILP), AGNEs and heuristic miners according to logs that are mentioned in the respective publications. We did not perform the evaluation on α and α^{+} as [10, 11] have reported that α^{++} can construct a model that handles more complex control-flow constructs.

4.1 Criteria

Buijs et al. [34] reviewed all the existing criteria used by various researchers to validate their process mining techniques and found that they all shared the notions of *simplicity*, *replay fitness*, *precision* and *generalization*. Therefore, we have also used these criteria along with *time* to evaluate the performance of our algorithm.

Replay fitness measures how well the model can generate the traces in the event log. Alignment distance function defined by van der Aalst, et al. [35] is used to calcu-

late fitness of a process model. The generated process model should be simple, i.e. it only includes the necessary number of events and links to explain anything [35]. The *simplicity* measures defined in Buijs et al. [34] is used. *Precision* estimates how much the model accepts additional behaviour that is not seen in the log. Align precision metrics by Adriansyah, et al. [36] is used to test whether the generated model under-fits. In contrast to *Precision*, *generalization* addresses an overly precise model. As mentioned previously, it is not likely that a log is complete and noise-free. Therefore, the mined model should be robust enough so that the removal or addition of small percentage of traces in the log will not lead to a remarkably different model. Most importantly, processing time is also considered one of the critical criteria, we include the *time* taken by an algorithm to mine a process model as one of our quantitative criteria.

4.2 *k*-fold Cross Validation

Based on the existing techniques, evaluating the quality of a mined model is achieved by calculating the *replay fitness*, the *simplicity*, the *precision*, and the *generalization* measures of the model on all of the traces found in a log, and usually the log used during the evaluation is the same log that is used to build the model. However, it is well-known in the statistics and machine learning communities that it is a methodological mistake to learn and test the performance of a prediction function on the same data as the generated model will in all likelihood get 100% accuracy on the training data but perform very poorly on a new set of input data. This phenomenon is called *over-fitting*. To avoid over-fitting, the available data should be separated into a training data set that is used to generate the model, and a test data set that is used to evaluate the quality of the generated model. The most common approach to do this is called *k-fold cross validation* and this is the evaluation method that we use to evaluate the *generalization* measures of each model [35].

With *k*-fold cross validation, the log is split into *k* approximately equal sized partitions. Each partition is used exactly once as the test set while the remaining data is used as the training set. The performance of the algorithm is the average of the values computed on each iteration. For example, in 3-fold cross validation, the log is divided into 3 equal sized groups (A, B, C). First, the algorithm uses A and B as training data. The performance of the generated model (P_1) is then tested on C. Next, the algorithm uses B and C as training data, and evaluates the performance of the constructed model (P_2) on A. Finally, A and C are used as training data, and the performance of the generated model (P_3) is tested on B. The overall performance of the algorithm is measured by averaging P_1, P_2 and P_3. In our evaluation, we use 10-fold cross validation. Because k-fold cross validation ensures that our model does not over-fit the training data (i.e. the model is general enough), the performance measured in each iteration is *simplicity*, *replay fitness*, and *precision*. The overall performance of a technique on a log is calculated by averaging the performance of that technique on each fold.

In the process mining area, some researchers avoid over-fitting by evaluating the performance of their generated model on "noisy" logs. These logs are created by adding noise (such as artificial start and end events, incorrect event labelling, event mutation, traces addition and removal, etc.) to original logs. However, as mentioned previously, without an explicit reference model, we do not know which specific

instances in the original logs is noise. Therefore, artificial logs may actually generate positive examples that we did not observe in the original log and we may incorrectly label them as negative instances [18]. This is also the reason why we do not think stratified k-fold cross validation, where we artificially create negative examples, as proposed by [30] is appropriate.

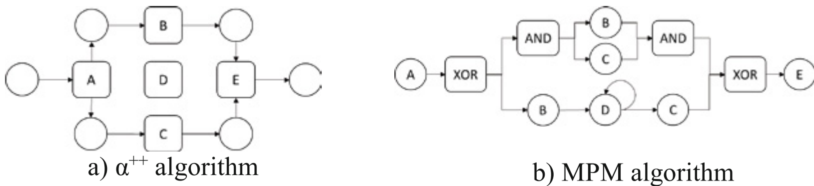
A one-way analysis of variance and paired t-tests is then used to examine statistically significant differences in the performance of each algorithm. This way we can generate a process model on several data sets.

4.3 Setup

Similar to other discovery algorithms, our MPM algorithm is implemented as a plugin of ProM [29]. In our evaluation, we use synthetics and real event log data to demonstrate the fact that the MPM algorithm can significantly improve the performance of the existing approaches, especially the α-algorithm and its variants. We do not use parameter fine-tuning or metadata to enhance the performance of our algorithm. We have also used the default settings for α^{++}, genetic miner, ILP and AGNEs. To further extend the capability of Heuristic Miner, we configure it to discover long distance dependencies based on completed events' values and positions on a trace.

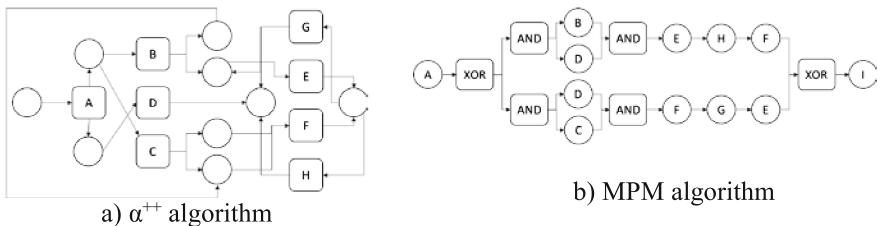
4.4 Synthetic Data

We compare the performance of MPM, α^{++}, Genetic miner, ILP, AGNEs and Heuristic Miners on the artificial logs example that are used in [10, 11, 37, 18]. There are about 300 to 350 traces and maximum 10 unique events in each log.



a) α^{++} algorithm b) MPM algorithm

Fig. 3. Log *T* = {*ABCE, ACBE, ABDDCE*}

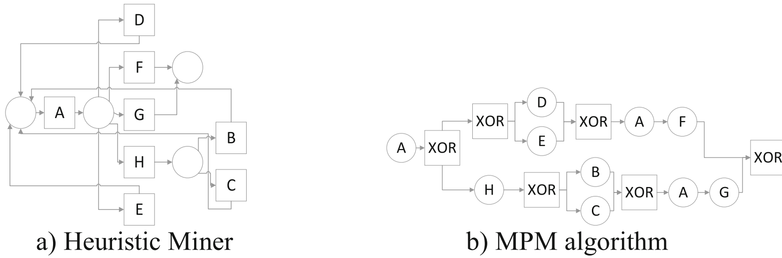


a) α^{++} algorithm b) MPM algorithm

Fig. 4. Log T = {ABDEHFI, ADBEHFI, ACDFGEI, ADCFGEI}

Due to the fact that the α^{++} algorithm builds a process model based on the relationship between *any two* events so that it does not allow an event to occur more than once in the model, it requires additional heuristics to handle long distance dependencies, short loops (maximum of two events) and non-free-choice constructs (combination of choice

and concurrency); and assumes that two or more events must occur concurrently if they have the same parents (i.e. bad precision). Therefore, it is possible for the α^{++} algorithm to produce *unsound* workflow nets as shown in Figures 3 and 4. Similarly, because Heuristic Miners also builds a casual matrix that represents the relationship between any two events, it cannot handle duplicate tasks as illustrated in Figure 5. Although AGNEs is more versatile than Heuristic Miners, it is still incapable of handling complex non-free choice constructs such as displayed in Figure 6.



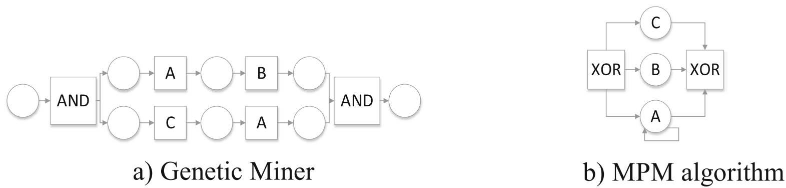
a) Heuristic Miner b) MPM algorithm

Fig. 5. Log T = {ADAF, AEAF, AHBAG, AHCAG}

a)AGNES b) MPM algorithm

Fig. 6. Log *T* = {*ABC, ABDE, ADBE*}

Our MPM algorithm discovers a process model by reading patterns from the *whole sequence* of events in traces. Thus, it has more stringent criteria than Heuristic Miners or α^{++}; it can handle duplicate tasks, long distance dependencies, loops of any length and non-free choice constructs. The process model discovered by MPM is always sound, and it is generally more accurate than the models mined by AGNEs, ILP, Heuristic Miners or α^{++}. However, MPM is incapable of generating a model that accurately represents duplicate tasks in a parallel process structure, as shown in Figure 7. Genetic Algorithm was the only algorithm that correctly mined this log.



a) Genetic Miner b) MPM algorithm

Fig. 7. Log T = {ACBA, ACAB, CAAB, CABA, ABCA}

As displayed in Table 1, ILP and MPM are the only algorithms that can consistently produce perfectly fitting models across all the synthetic data. Followed closely by Genetic, Heuristics and AGNEs miner. A paired t-test evaluation showed that there are significant differences at the 95% level in fitness performance between the ILP or MPM to α^{++}.

Table 1. The average *replay fitness*, *precision*, *simplicity* and run-time comparisons for the artificial logs

	Fitness	Precision	Simplicity	Time
α^{++}	0.7	0.5	1.0	250 ms
Genetic	0.9	0.9	0.7	1 hour
Heuristics	0.8	0.7	0.8	10 s
AGNEs	0.8	0.8	0.6	5 mins
ILP	1.0	0.6	0.9	2 mins
MPM	1.0	0.9	0.7	150 ms

ILP, α^{++} and Heuristic Miners tend to create overly general models making them much less precise. AGNEs can produce models that are more precise than α^{++}, ILP and Heuristic Miners so there is only a 90% level of significant difference between Genetic Algorithm and MPM to AGNEs.

Simplicity of Genetic Miner, AGNEs and MPM is rather low due to the duplication of several events. There are no significant differences at the 95% level in terms of the average simplicity between these algorithms compared to Heuristics miner. α^{++} and ILP are significantly better at the 95% level compared to Genetic Miner, AGNEs and MPM.

While Genetic Miner will sometimes produce a model that is more accurate than MPM, MPM can generate a similar model in significantly less time. Unlike ILP, MPM always generated sound model. Furthermore, MPM can incrementally build and improve the mined model in near real time as it receives new traces of events, i.e. the model becomes more accurate as it sees more unique traces of events.

4.5 Real-Life Log Data

Similar to the previous section, to evaluate the performance of MPM, α^{++}, Genetic miner, AGNEs, ILP and Heuristic Miners we used the real-life Hospital log obtained from [32]. For each log, we let each of the algorithms run for 5 days and if they exceeded that we counted them as DNF (Did Not Finish).

From Table 2, we can see a similar pattern to the one that we found with the synthetic data. Genetic Miner is the algorithm that takes the longest to finish. It takes at least 5 days for Genetic Miner to return any sort of result. Therefore, we could not comment on the simplicity, precision and replay fitness of the model generated by Genetic Miner. The worst performing algorithm in terms of fitness and precision is α^{++}. Even though the model generated by ILP can replay the trace log perfectly with just enough number of nodes, the model tend to under-fit. Unlike with the synthetic

data, Heuristic Miners significantly outperforms AGNEs in terms of fitness, precision, simplicity and running time with the real-life log data analysis at the 95% level. Heuristic Miners could generate a model significantly faster than AGNEs, and the model is significantly more accurate and robust than that generated from AGNEs. We argue that this difference is caused by the introduction of incorrect false negative examples in AGNEs. Real-life logs contain much noise and tend to be incomplete. As such, it is fairly easy for AGNEs to regard an unobserved positive behaviour as a negative example. On the other hand, Heuristic Miners decides the relationship between two processes based on the probability of process B following process A given the evidence of prior processes as such it is more robust to noise. However, MPM is still significantly better than Heuristic Miners at the 95% in terms of fitness and run-time.

Table 2. Average *replay fitness*, *precision*, *simplicity* and run-time of different techniques across multiple logs

	Fitness	Precision	Simplicity	Time
α^{++}	0.3	0.4	0.9	10 mins
Genetic	DNF	DNF	DNF	>5 days
Heuristics	0.7	0.8	0.8	1 hour
AGNEs	0.5	0.6	0.3	20 hours
ILP	1.0	0.5	0.8	2 hours
MPM	0.9	0.9	0.7	9 mins

5 Conclusion and Future work

In this paper, we propose a novel technique called *Maximum Pattern Mining* (MPM) to discover a process model from event logs. We implemented our technique and evaluated it against the well-known process discovery algorithms: α^{++}, Genetic miner, ILP AGNEs and Heuristic Miners algorithms. The results from our experiments show that MPM performs better or comparable in terms of *fitness*, *precision*, *simplicity* and *run-time efficiency*. It can handle much more general cases, such as loops of any length and long distance dependencies. However to achieve high fitness and precision, MPM tends to use duplicate events in the model which caused low simplicity score. In the future, we will implement a graph that will allow users to define the tasks' abstraction level and hopefully increase MPM's simplicity score. As MPM was able to efficiently generate an accurate model from a real-life log in near real time, event-stream mining is feasible.

References

1. van der Aalst, W.M.P.: Process Mining: Overview and Opportunities. ACM Transactions on Management Information Systems **3**(2) article 7 (2012)
2. 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)
3. Cook, J., Wolf, A.: Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology **7**, 215–249 (1998)
4. Datta, A.: Automating the discovery of AS-IS business process models: probabilistic and algorithmic approaches. Information Systems Research **9**, 275–301 (1998)
5. Mannila, H., Meek, C.: Global partial orders from sequential data. In: Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2000), pp. 161–168 (2000)
6. Schimm, G.: Process miner - a tool for mining process schemes from event-based data. In: Flesca, S., Greco, S., Leone, N., Ianni, G. (eds.) JELIA 2002. LNCS (LNAI), vol. 2424, pp. 525–528. Springer, Heidelberg (2002)
7. Schimm, G.: Mining exact models of concurrent workflows. Computers in Industry **53**, 265–281 (2004)
8. 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**, 1128–1142 (2004)
9. Alves de Medeiros, A.K., van Dongen, B.F., van der Aalst, W.M.P., Weijters, A.J.M.M.: Process Mining: Extending the Alpha-Algorithm to Mine Short Loops. BETA Working Paper Series, vol. 113. TU Eindho- ven (2004)
10. Wen, L., van der Aalst, W.M.P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. Data Mining and Knowledge Discovery **15**, 145–180 (2007)
11. Wen, L., Wang, J., van der Aalst, W.M.P., Huang, B., Sun, J.: A novel approach for process mining based on event types. Journal of Intelligent Information Systems **32**, 163–190 (2009)
12. Weijters, A.J.M.M., van der Aalst, W.M.P., Alves de Medeiros, A.K.: Process Mining with the Heuristics Miner algorithm. BETA Working Paper Series, vol. 166. TU Eindhoven (2006)
13. Günther, C.W., van der Aalst, W.M.: 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)
14. Maruster, L., Weijters, A.J.M.M., van der Aalst, W.M.P., van den Bosch, A.: A rule-based approach for process discovery: dealing with noise and imbalance in process logs. Data Mining and Knowledge Discovery **13**, 67–87 (2006)
15. Ferreira, H., Ferreira, D.: An integrated life cycle for workflow management based on learning and planning. IJCIS **15**, 485–505 (2006)
16. Lamma, E., Mello, P., Montali, M., Riguzzi, F., Storari, S.: Inducing declarative logic-based models from labeled traces. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 344–359. Springer, Heidelberg (2007)
17. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust process discovery with artificial negative events. Journal of Machine Learning Research **10**, 1305–1340 (2009)
18. Blockeel, H., De Raedt, L.: Top-down induction of first-order logical decision trees. Artificial Intelligence **101**, 285–297 (1998)
19. Greco, G., Guzzo, A., Pontieri, L., Sacca, D.: Discovering expressive process models by clustering log traces. IEEE Transactions on Knowledge and Data Engineering **18**, 1010–1027 (2006)

20. Greco, G., Guzzo, A., Pontieri, L.: Mining taxonomies of process models. Data & Knowledge Engineering **67**, 74–102 (2008)
21. 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 System Modeling **9**, 87–111 (2010)
22. Carmona, J., Cortadella, J., Kishinevsky, M.: New region-based algorithms for deriving bounded Petri nets. IEEE Transactions on Computers **59**, 371–384 (2010)
23. Herbst, J., Karagiannis, D.: Workflow mining with InWoLvE. Computers in Industry **53**, 245–264 (2004)
24. Folino, F., Greco, G., Guzzo, A., Pontieri, L.: Discovering expressive process models from noised log data. In: Proceedings of the 2009 International Database Engineering & Applications Symposium, pp. 162–172. ACM (2009)
25. 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 (2009)
26. Rembert, A.J., Omokpo, A., Mazzoleni, P., Goodwin, R.T.: Process discovery using prior knowledge. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 328–342. Springer, Heidelberg (2013)
27. Ferreira, D.R., Gillblad, D.: Discovering process models from unlabelled event logs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 143–158. Springer, Heidelberg (2009)
28. Sole, M., Carmona, J.: Region-Based Folding in Process Discovery. IEEE Transactions on Knowledge and Data Engineering **25**(1), 192–205 (2013)
29. Günther, C.W., Verbeek, E.: XES Standard version 2 (2014). http://www.xes-standard.org/_media/xes/xesstandarddefinition-2.0.pdf
30. Rozinat, A., de Medeiros, A.K.A., Günther, C.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The need for a process mining evaluation framework in research and practice. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.Y. (eds.) BPM 2007. LNCS, vol. 9428, pp. 84–89. Springer, Heidelberg (2007)
31. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proc. 8th ACM Intern. Conf. Knowl. Discov. Data Mining, pp. 429–435. ACM (2002)
32. 3TU Data Center, BPI Challenge, Event Log, data.3tu.nl/repository/collection:event_logs_real
33. Bose, R.P.J.C., van der Aalst, W.M.: Abstractions in process mining: a taxonomy of patterns. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 159–175. Springer, Heidelberg (2009)
34. Buijs, J.C.A.M., van Dongen, B. F., van der Aalst, W. M. P. : Quality Dimensions in Process Discovery: The importance of Fitness, Precision, Generalization and Simplicit., IJCIS **2**(1) (2014)
35. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Replaying History on Process Models for Conformance Checking and Performance Analysis. WIREs Data Mining and Knowledge Discover **2**(2), 182–192 (2012)
36. Adriansyah, A., Munoz-Game, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Measuring Precision of Modeled Behaviour. Information systems and e-Business Management (2015)
37. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: A genetic algorithm for discovering process trees. In: IEEE Congress on Evolutionary Computation (CEC 2012), pp. 1–8. IEEE Computer Society (2012)