Genetic Process Mining

W.M.P. van der Aalst, A.K. Alves de Medeiros, and A.J.M.M. Weijters

Department of Technology Management, Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{w.m.p.v.d.aalst,a.k.medeiros,a.j.m.m.weijters}@tm.tue.nl

Abstract. The topic of process mining has attracted the attention of both researchers and tool vendors in the Business Process Management (BPM) space. The goal of process mining is to *discover* process models from event logs, i.e., events logged by some information system are used to extract information about activities and their causal relations. Several algorithms have been proposed for process mining. Many of these algorithms cannot deal with concurrency. Other typical problems are the presence of duplicate activities, hidden activities, non-free-choice constructs, etc. In addition, real-life logs contain noise (e.g., exceptions or incorrectly logged events) and are typically incomplete (i.e., the event logs contain only a fragment of all possible behaviors). To tackle these problems we propose a completely new approach based on genetic algorithms. As can be expected, a genetic approach is able to deal with noise and incompleteness. However, it is not easy to represent processes properly in a genetic setting. In this paper, we show a genetic process mining approach using the so-called *causal matrix* as a representation for individuals. We elaborate on the relation between Petri nets and this representation and show that genetic algorithms can be used to discover Petri net models from event logs.

Keywords: Process Mining, Petri Nets, Genetic Algorithms, Process Discovery, Business Process Intelligence, Business Activity Monitoring.

1 Introduction

Buzzwords such as Business Process Intelligence (BPI) and Business Activity Monitoring (BAM) illustrate the practical interest in techniques to extract knowledge from the information recorded by today's information systems. Most information systems support some form of logging. For example, Enterprise Resource Planning (ERP) systems such as SAP R/3, PeopleSoft, Oracle, JD Edwards, etc. log transactions at various levels. Any Workflow Management (WfM) system records audit trails for individual cases. The Sarbanes-Oxley act is forcing organizations to log even more information. The availability of this information triggered the need for process mining techniques that analyze event logs.

The goal of process mining is to extract information about processes from transaction logs [3]. We assume that it is possible to record events such that (i) each event refers to an *activity* (i.e., a well-defined step in the process), (ii) each event refers to a *case* (i.e., a process instance), (iii) each event *can* have a *performer* also referred to as *originator* (the actor executing or initiating the

activity), and (iv) events *can* have a *timestamp* and are totally ordered. Table 1 shows an example of a log involving 18 events and 8 activities. In addition to the information shown in this table, some event logs contain more information on the case itself, i.e., data elements referring to properties of the case.

case id	activity id	originator	timestamp	case id	activity id	originator	timestamp
case 1	activity A	John	09-3-2004:15.01	case 3	activity E	Pete	10-3-2004:12.50
case 2	activity A	John	09-3-2004:15.12	case 3	activity F	Carol	11-3-2004:10.12
case 3	activity A	Sue	09-3-2004:16.03	case 4	activity D	Pete	11-3-2004:10.14
case 3	activity D	Carol	09-3-2004:16.07	case 3	activity G	Sue	11-3-2004:10.44
case 1	activity B	Mike	09-3-2004:18.25	case 3	activity H	Pete	11-3-2004:11.03
case 1	activity H	John	10-3-2004:09.23	case 4	activity F	Sue	11-3-2004:11.18
case 2	activity C	Mike	10-3-2004:10.34	case 4	activity E	Clare	11-3-2004:12.22
case 4	activity A	Sue	10-3-2004:10.35	case 4	activity G	Mike	11-3-2004:14.34
case 2	activity H	John	10-3-2004:12.34	case 4	activity H	Clare	11-3-2004:14.38

Table 1. An event log (audit trail).

Event logs such as the one shown in Table 1 are used as the starting point for mining. We distinguish three different perspectives: (1) the process perspective, (2) the organizational perspective and (3) the case perspective. The *process perspective* focuses on the control-flow, i.e., the ordering of activities. The goal of mining this perspective is to find a good characterization of all possible paths, e.g., expressed in terms of a Petri net or Event-driven Process Chain (EPC). The *organizational perspective* focuses on the originator field, i.e., which performers are involved and how are they related. The goal is to either structure the organization by classifying people in terms of roles and organizational units or to show relation between individual performers (i.e., build a social network [2]). The *case perspective* focuses on properties of cases. Cases can be characterized by their path in the process or by the originators working on a case. However, cases can also be characterized by the values of the corresponding data elements. For example, if a case represents a replenishment order it is interesting to know the supplier or the number of products ordered.

The process perspective is concerned with the "How?" question, the organizational perspective is concerned with the "Who?" question, and the case perspective is concerned with the "What?" question. In this paper we will focus completely on the process perspective, i.e., the ordering of the activities. This means that here we ignore the last two columns in Table 1. (Although the timestamps determine the order of events (activities) in a case, the actual timestamps are not used during mining.) For the mining of the other perspectives we refer to [3] and http://www.processmining.org. Note that the ProM tool described in this paper is able to mine the other perspectives and can also deal with other issues such as transactions, e.g., in the ProM tool we consider different event types such as "schedule", "start", "complete", "abort", etc. However, for reasons of simplicity we abstract from this in this paper and consider activities to be atomic as shown in Table 1.

If we abstract from the other perspectives, Table 1 contains the following information: case 1 has event trace A, B, H, case 2 has event trace A, C, H, case 3 has event trace A, D, E, F, G, H, and case 4 has event trace A, D, F, E, G, H. If we analyze these four sequences we can extract the following information about the process (assuming some notion of completeness and no noise). The underlying process has 8 activities (A, B, ..., H). A is always the first activity to be executed and H is always the last one. After A is executed, activities B, C *or* D can be executed. In other words, after A, there is a *choice* in the process and only one of these activities can be executed next. When B or C are executed, they are followed by the execution of H (see cases 1 and 2). When D is executed, both E *and* F can be executed in any order. Since we do not consider explicit parallelism, we assume E and F to be concurrent (see cases 3 and 4). Activity G synchronizes the parallel branches that contain E and F. Activity H is executed whenever B, C *or* G has been executed. Based on these observations, the Petri net shown in Figure 1 is a good model for the event log containing the four cases. Note that each of the four cases can be "reproduced" by the Petri net shown in Figure 1, i.e. the Petri net contains all observed behavior. In this particular case, also the reverse holds, i.e., all possible firing sequences of the Petri net shown in Figure 1 are contained in the log. Generally, this is not the case since in practice it is unrealistic to assume that all possible behavior is always contained in the log, cf. the discussion on completeness in [4].

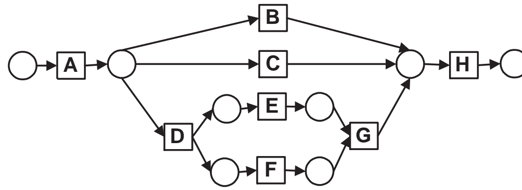


Fig. 1. Petri net discovered based on the event log in Table 1.

Existing approaches for mining the process perspective [3–6, 11, 13, 18] have problems dealing with issues such as duplicate activities, hidden activities, non-free-choice constructs, noise, and incompleteness. The problem with *duplicate activities* occurs when the same activity can occur at multiple places in the process. This is a problem because it is no longer clear to which activity some event refers. The problem with *hidden activities* is that essential routing decisions are not logged but impact the routing of cases. *Non-free-choice* constructs are problematic because it is not possible to separate choice from synchronization. We consider two sources of *noise*: (1) incorrectly logged events (i.e., the log does not reflect reality) or (2) exceptions (i.e., sequences of events corresponding to "abnormal behavior"). Clearly noise is difficult to handle. The problem of *incompleteness* is that for many processes it is not realistic to assume that all possible behavior is contained in the log. For processes with many alternative routes and parallelism, the number of possible event traces is typically exponential in the number of activities, e.g., a process with 10 binary choices in a sequence will

have $2^{10}(= 1024)$ possible event sequences and a process with 10 activities in parallel will have even $10!(= 3628800)$ possible event sequences.

We can consider process mining as a search for the most appropriate process out of the search space of candidate process models. Mining algorithms can use different strategies to find the most appropriate model. Two extreme strategies can be distinguished (i) *local strategies* primarily based on a step by step building of the optimal process model based on local information, and (ii) *global strategies* primarily based on a one strike search for the optimal model. Most process mining approaches use a local strategy. An example of a such a local strategy is used by the α-algorithm [4] where only local information about binary relations between events is used. A genetic search is an example of a global search strategy; because the quality or fitness of a candidate model is calculated by comparing the process model with all traces in the event log the search process takes place at a global level. For a local strategy there is no guarantee that the outcome of the locally optimal steps (at the level of binary event relations) will result in a globally optimal process model. Hence, the performance of such local mining techniques can be seriously hampered when the necessary information is not locally available because one erroneous example can completely mess up the derivation of a right model. Therefore, we started to use genetic algorithms.

In this paper, we present a *genetic algorithm to discover a Petri net given a set of event traces*. Genetic algorithms are adaptive search methods that try to mimic the process of evolution [9]. These algorithms start with an initial population of individuals (in this case process models). Populations evolve by selecting the fittest individuals and generating new individuals using genetic operators such as *crossover* (combining parts of two of more individuals) and *mutation* (random modification of an individual). Our initial work on applying genetic algorithms to process mining [14] shows that a direct representation of individuals in terms of a Petri net is not a very convenient. First of all, the Petri net contains places that are not visible in the log. (Note that in Figure 1 we cannot assign meaningful names to places.) Second, the classical Petri net is not a very convenient notation for generating an initial population because it is difficult to apply simple heuristics. Third, the definition of the genetic operators (crossover and mutation) is cumbersome. Finally, the expressive power of Petri nets is in some cases too limited (combinations of AND/OR-splits/joins). Therefore, we use an internal representation named *casual matrix*. However, we use Petri nets to give semantics to this internal representation and adopt many ideas from Petri nets (e.g., playing the token game to measure fitness). Moreover, in this paper we focus on the relation between the casual matrix and Petri nets.

The remainder of this paper is organized as follows. First, we discuss some related work (Section 2) and start with some preliminaries (Section 3). Then, in Section 4, we present the causal matrix as our internal representation. In Section 5 we explore the relation between the causal matrix and Petri nets. Section 6 introduces the genetic algorithm and in Section 7 some experimental results are given. Finally, we conclude the paper.

2 Related Work

The idea of process mining is not new [3, 4, 2, 5, 6, 11, 13, 18]. Most of the scientific papers aim at the control-flow perspective, although a few focus on other perspectives such as the organizational perspective [2]. It is also interesting to note that some commercial tools such as ARIS PPM offer some limited form of process mining as discussed in this paper. However, most tools in the BPI/BAM arena focus on key performance indicators such as flow time and frequencies.

Given the many papers on mining the process perspective it is not possible to give a complete overview. Instead we refer to [3, 4]. Historically, Cook et al. [6] and Agrawal et al. [5] started to work on the problem addressed in this paper. Herbst et al. [11] took an alternative approach which allows for dealing with duplicate activities. The authors of this paper have been involved in different variants of the so-called α-algorithm [4, 18]. Each of the approaches has its pros and its cons. Most approaches that are able to discover concurrency have problems dealing with issues such as duplicate activities, hidden activities, non-free-choice constructs, noise, and incompleteness.

There have been some papers combining Petri nets and genetic algorithms, cf. [12, 15–17]. However, these papers do not try to discover a process model based on some event log.

For readers familiar with Petri net theory, it is important to discuss the relation between this work and the work on regions [8]. The seminal work on regions investigates which transition systems can be represented by (compact) Petri nets (i.e., the so-called synthesis problem). Although there are related problems such as duplicate transitions, etc., the setting is quite different because our notion of completeness is much weaker than perfect knowledge of the underlying transition system. We assume that the log contains only a fraction of the possible behavior, as mentioned in the introduction.

The approach in this paper is the first approach using genetic algorithms for process discovery. Some more details about the experimental/genetic-side of this approach can be found in a technical report [14]. The goal of using genetic algorithms is to tackle problems such as duplicate activities, hidden activities, non-free-choice constructs, noise, and incompleteness, i.e., overcome the problems of some of the traditional approaches. However, in this paper we focus on the initial idea and the representation rather than a comparison with existing non-genetic algorithms.

3 Preliminaries

This section briefly introduces the basic *Petri net* terminology and notations, and also discusses concepts such as *WF-nets* and *soundness*.

Definition 1 (Petri net). *A Petri net is a triple (P, T, F). P is a finite set of places, T is a finite set of transitions ($P \cap T = \emptyset$), and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).*

For any relation/directed graph $G \subseteq N \times N$ we define the preset $\bullet n = \{(m_1, m_2) \in G \mid n = m_2\}$ and postset $n\bullet = \{(m_1, m_2) \in G \mid n = m_1\}$ for any node $n \in N$. We use $\overset{G}{\bullet}\, n$ or $n \overset{G}{\bullet}$ to explicitly indicate the context G if needed. Based on the flow relation F we use this notation as follows. $\bullet t$ denotes the set of input places for a transition t. The notations $t\bullet$, $\bullet p$ and $p\bullet$ have similar meanings, e.g., $p\bullet$ is the set of transitions sharing p as an input place. Note that we do not consider multiple arcs from one node to another.

At any time a place contains zero or more *tokens*, drawn as black dots. This state, often referred to as *marking*, is the distribution of tokens over places, i.e., $M \in P \rightarrow \mathbb{N}$. For any two states M_1 and M_2, $M_1 \leq M_2$ iff for all $p \in P$: $M_1(p) \leq M_2(p)$. We use the standard *firing rule*, i.e., a transition t is said to be *enabled* iff each input place p of t contains at least one token, an enabled transition may *fire*, and if transition t fires, then t *consumes* one token from each input place p of t and *produces* one token for each output place p of t.

Given a Petri net (P, T, F) and a state M_1, we have the standard notations for a transition t that is enabled in state M_1 and firing t in M_1 results in state M_2 (notation: $M_1 \xrightarrow{t} M_2$) and a firing sequence $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ leads from state M_1 to state M_n via a (possibly empty) set of intermediate states (notation: $M_1 \xrightarrow{\sigma} M_n$). A state M_n is called *reachable* from M_1 (notation $M_1 \xrightarrow{*} M_n$) iff there is a firing sequence σ such that $M_1 \xrightarrow{\sigma} M_n$. Note that the empty firing sequence is also allowed, i.e., $M_1 \xrightarrow{*} M_1$.

In this paper, we will focus on a particular type of Petri nets called *WorkFlow nets* (WF-nets) [1, 7, 10].

Definition 2 (WF-net). *A Petri net* $PN = (P, T, F)$ *is a WF-net (Workflow net) if and only if:*

(i) There is one source place $i \in P$ *such that* $\bullet i = \emptyset$.

(ii) There is one sink place $o \in P$ *such that* $o\bullet = \emptyset$.

(iii) Every node $x \in P \cup T$ *is on a path from* i *to* o.

A WF-net represents the life-cycle of a case that has some initial state represented by a token in the unique input place (i) and a desired final state represented by a token in the unique output place (o). The third requirement in Definition 2 has been added to avoid "dangling transitions and/or places". In the context of workflow models or business process models, transitions can be interpreted as *tasks* or *activities* and places can be interpreted as *conditions*. Although the term "WorkFlow net" suggests that the application is limited to workflow processes, the model has wide applicability, i.e., any process where each case has a life-cycle going from some initial state to some final state fits this basic model.

The three requirements stated in Definition 2 can be verified statically, i.e., they only relate to the structure of the Petri net. To characterize desirable dynamic properties, the notation of *soundness* has been defined [1, 7, 10].

Definition 3 (Sound). *A procedure modelled by a WF-net* $PN = (P, T, F)$ *is sound if and only if:*

(i) *For every state M reachable from state i, there exists a firing sequence leading from state M to state o. Formally:* $\forall_M (i \stackrel{*}{\rightarrow} M) \Rightarrow (M \stackrel{*}{\rightarrow} o)$.[1]

(ii) *State o is the only state reachable from state i with at least one token in place o. Formally:* $\forall_M (i \stackrel{*}{\rightarrow} M \ \wedge \ M \geq o) \Rightarrow (M = o)$.

(iii) *There are no dead transitions in (PN, i). Formally:* $\forall_{t \in T} \ \exists_{M,M'} \ i \stackrel{*}{\rightarrow} M \stackrel{t}{\rightarrow} M'$.

Note that the soundness property relates to the dynamics of a WF-net. The first requirement in Definition 3 states that starting from the initial state (state i), it is always possible to reach the state with one token in place o (state o). The second requirement states that the moment a token is put in place o, all the other places should be empty. The last requirement states that there are no dead transitions (activities) in the initial state i.

4 Causal Matrix

After these preliminaries we return to the goal of this paper: genetic process mining. In order to apply a genetic algorithm we need to represent individuals. Each individual corresponds to a possible process model and its representation should be easy to handle. Our initial idea was to represent processes directly by Petri nets. Unfortunately, Petri nets turn out to be a less convenient way to represent processes in this context. The main reason is that in Petri nets there are places whose existence cannot be derived from the log, i.e., events only refer to the active components of the net (transitions). Because of this it becomes more difficult to generate an initial population, define genetic operators (crossover and mutation), and describe combinations of AND/OR-splits/joins. Note that given a log it is very easy to discover the activities and therefore the transitions that exist in the Petri net. However, enforcing certain routings by just connecting transitions through places is complex (if not impossible). Therefore, we will use a different internal representation. However, this representation and its semantics are closely linked to Petri nets as will be shown in Section 5.

Table 2 shows the internal representation of an individual used by our genetic mining approach. This so-called *causal matrix* defines the causal relations between the activities and in case of multiple input or output activities, the logic is depicted. Consider for example the row starting with A. This row shows that there is a not a causal relation between A and A (note the first 0 in the row), but there is a causal relation between A and B (note the first 1 in this row). The next two entries in the row show that there are also causal relations between A and C and A and D. The last element in the row shows the routing logic, i.e., $B \vee C \vee D$ indicates that A is followed by B, C, or D. The column labelled 'OUTPUT' shows the logic relating an activity to causally following activities. The first row below 'INPUT' shows the logic relating an activity to causally

[1] Note that there is an overloading of notation: the symbol i is used to denote both the *place* i and the *state* with only one token in place i. The same holds for o.

	INPUT								
	$true$	A	A	A	D	D	$E \wedge F$	$B \vee C \vee G$	
$\rightarrow$	A	B	C	D	E	F	G	H	*OUTPUT*
A	0	1	1	1	0	0	0	0	$B \vee C \vee D$
B	0	0	0	0	0	0	0	1	H
C	0	0	0	0	0	0	0	1	H
D	0	0	0	0	1	1	0	0	$E \wedge F$
E	0	0	0	0	0	0	1	0	G
F	0	0	0	0	0	0	1	0	G
G	0	0	0	0	0	0	0	1	H
H	0	0	0	0	0	0	0	0	$true$

Table 2. A causal matrix is used for the internal representation of an individual.

preceding activities. Note that the input condition of A is *true*, i.e., no input needed. Activity G has $E \wedge F$ as input condition, i.e., both E and F need to complete in order to enable G. Activity H has $B \vee C \vee G$ as input condition, i.e., B, C, or G needs to complete in order to enable H.

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}, \{F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{B, C, G\}\}$	$\{\}$

Table 3. A more succinct encoding of the individual shown in Table 2.

Table 3 shows a more convenient notation removing some of the redundancies. Note that the 0 and 1 entries in Table 2 can be trivially derived from the input and output conditions. Moreover, we assume that we can write the logical expressions in a normal form, e.g., $\{\{B, C, D\}\}$ corresponds to $B \vee C \vee D$, $\{\{E\}, \{F\}\}$ corresponds to $E \wedge F$, and $\{\{A, B\}, \{C, D\}\}$ corresponds to $(A \vee B) \wedge (C \vee D)$. In fact, the logical expression is represented by a set of sets corresponding to a conjunction of disjunctions, i.e., a kind of Conjunctive Normal Form (CNF).[2]

Let us now formalize the notion of a *causal matrix*.

Definition 4 (Causal Matrix). *A Causal Matrix is a tuple $CM = (A, C, I, O)$, where*

- *A is a finite set of activities,*
- *$C \subseteq A \times A$ is the causality relation,*

[2] Note that unlike the conjunctive normal form we do not allow for negation and also do not allow for “overlapping” disjunctions, cf. Definition 4.

- $I \in A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the input condition function,*[3]
- $O \in A \rightarrow \mathcal{P}(\mathcal{P}(A))$ *is the output condition function,*

such that

- $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$,[4]
- $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$,
- $\forall_{a \in A} \ \forall_{s,s' \in I(a)} \ s \cap s' \neq \emptyset \ \Rightarrow \ s = s'$,
- $\forall_{a \in A} \ \forall_{s,s' \in O(a)} \ s \cap s' \neq \emptyset \ \Rightarrow \ s = s'$,
- $C \cup \{(a_o, a_i) \in A \times A \mid a_o \overset{C}{\bullet} = \emptyset \wedge \overset{C}{\bullet} a_i = \emptyset\}$ *is a strongly connected graph.*

The mapping of Table 3 onto $CM = (A, C, I, O)$ is straightforward (the latter two columns represent I and O). Note that C can be derived from both I and O. Its main purpose is to ensure consistency between I and O. For example, if a_1 has an output condition mentioning a_2, then a_2 has an input condition mentioning a_1 (and vice versa). This is enforced by the first two constraints. The third and fourth constraint indicate that some activity a may appear only once in the conjunction of disjunctions, e.g., $\{\{A, B\}, \{A, C\}\}$ is not allowed because A appears twice. The last requirement has been added to avoid that the causal matrix can be partitioned in two independent parts or that nodes are not on a path from some source activity a_i to a sink activity a_o.

5 Relating the Causal Matrix and Petri nets

In this section we relate the causal matrix to Petri nets. We first map Petri nets (in particular WF-nets) onto the notation used by our genetic algorithms. Then we consider the mapping of the causal matrix onto Petri nets.

5.1 Mapping a Petri net onto a Causal Matrix

The mapping from an arbitrary Petri net to its corresponding causal matrix illustrates the expressiveness of the internal format used for genetic mining. First, we give the definition of the mapping $\Pi_{PN \rightarrow CM}$.

Definition 5 **($\Pi_{PN \rightarrow CM}$).** *Let $PN = (P, T, F)$ be a Petri net. The mapping of PN is a tuple $\Pi_{PN \rightarrow CM}(PN) = (A, C, I, O)$, where*

- $A = T$,
- $C = \{(t_1, t_2) \in T \times T \mid t_1 \bullet \cap \bullet t_2 \neq \emptyset\}$,
- $I \in T \rightarrow \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T} \ I(t) = \{\bullet p \mid p \in \bullet t\}$,
- $O \in T \rightarrow \mathcal{P}(\mathcal{P}(T))$ *such that* $\forall_{t \in T} \ O(t) = \{p \bullet \mid p \in t \bullet\}$.

[3] $\mathcal{P}(A)$ denotes the powerset of some set A.
[4] $\bigcup I(a_2)$ is the union of the sets in set $I(a_2)$.

Let PN be the Petri net shown in Figure 1. It is easy to check that $\Pi_{PN \to CM}(PN)$ is indeed the causal matrix in Table 2. However, there may be Petri nets PN for which $\Pi_{PN \to CM}(PN)$ is not a causal matrix. The following lemma shows that for the class of nets we are interested in, i.e., WF-nets, the requirement that there may not be two different places in-between two activities is sufficient to prove that $\Pi_{PN \to CM}(PN)$ represents a causal matrix as defined in Definition 4.

Lemma 1. *Let $PN = (P, T, F)$ be a WF-net with no duplicate places in between two transitions, i.e., $\forall_{t_1,t_2 \in T}\ |t_1 \bullet \cap \bullet t_2| \leq 1$. $\Pi_{PN \to CM}(PN)$ represents a causal matrix as defined in Definition 4.*

Proof. Let $\Pi_{PN \to CM} = (A, C, I, O)$. Clearly, $A = T$ is a finite set, $C \subseteq A \times A$, and $I, O \in A \to \mathcal{P}(\mathcal{P}(A))$. $C = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup I(a_2)\}$ because $a_1 \in \bigcup I(a_2)$ if and only if $a_1 \bullet \cap \bullet a_2 \neq \emptyset$. Similarly, $C = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup O(a_1)\}$. $\forall_{a \in A}\ \forall_{s,s' \in I(a)}\ s \cap s' \neq \emptyset \ \Rightarrow\ s = s'$ because $\forall_{t_1,t_2 \in T}\ |t_1 \bullet \cap \bullet t_2| \leq 1$. Similarly, $\forall_{a \in A}\ \forall_{s,s' \in O(t)}\ s \cap s' \neq \emptyset \ \Rightarrow\ s = s'$. Finally, it is easy to verify that $C \cup \{(a_o, a_i) \in A \times A \mid a_o \bullet = \emptyset \ \wedge\ \bullet a_i = \emptyset\}$ is a strongly connected graph. □

The requirement $\forall_{t_1,t_2 \in T}\ |t_1 \bullet \cap \bullet t_2| \leq 1$ is a direct result of the fact that in the conjunction of disjunctions in I and O, there may not be any overlaps. This restriction has been added to reduce the search space of the genetic mining algorithm, i.e., the reason is more of a pragmatic nature. However, for the success of the genetic mining algorithm such reductions are of the utmost importance.

5.2 A Naive Way of Mapping a Causal Matrix onto a Petri net

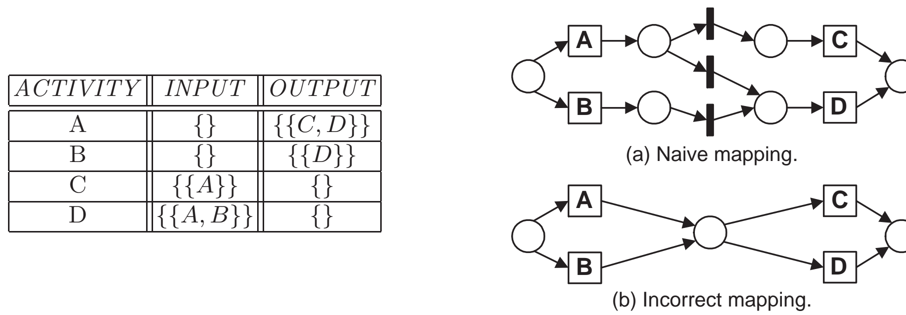
The mapping from a causal matrix onto a Petri net is more involved because we need to "discover places" and, as we will see, the causal matrix is slightly more expressive than classical Petri nets.[5] Let us first look at a naive mapping.

Definition 6 **($\Pi^N_{CM \to PN}$).** *Let $CM = (A, C, I, O)$ be a causal matrix. The naive Petri net mapping is a tuple $\Pi^N_{CM \to PN}(CM) = (P, T, F)$, where*

- $P = \{i, o\} \cup \{i_{t,s} \mid t \in A \ \wedge\ s \in I(t)\} \cup \{o_{t,s} \mid t \in A \ \wedge\ s \in O(t)\}$,
- $T = A \cup \{m_{t_1,t_2} \mid (t_1, t_2) \in C\}$,
- $F = \{(i, t) \mid t \in A \wedge \overset{C}{\bullet} t = \emptyset\} \cup \{(t, o) \mid t \in A \ \wedge\ t \overset{C}{\bullet} = \emptyset\} \cup \{(i_{t,s}, t) \mid t \in A \ \wedge\ s \in I(t)\} \cup \{(t, o_{t,s}) \mid t \in A \ \wedge\ s \in O(t)\} \cup \{(o_{t_1,s}, m_{t_1,t_2}) \mid (t_1, t_2) \in C \ \wedge\ s \in O(t_1) \ \wedge\ t_2 \in s\} \cup \{(m_{t_1,t_2}, i_{t_2,s}) \mid (t_1, t_2) \in C \ \wedge\ s \in I(t_2) \ \wedge\ t_1 \in s\}$.

The mapping $\Pi^N_{CM \to PN}$ maps activities onto transitions and adds input places and output places to these transitions based on functions I and O. These places are local to one activity. To connect these local places, one transition m_{t_1,t_2} is added for every $(t_1, t_2) \in C$. Figure 2 shows a causal matrix and its naive mapping $\Pi^N_{CM \to PN}$ (we have partially omitted place/transition names).

[5] Expressiveness should not be interpreted in a formal sense but in the sense of convenience when manipulating process instances, e.g., crossover operations.



ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{C, D\}\}$
B	$\{\}$	$\{\{D\}\}$
C	$\{\{A\}\}$	$\{\}$
D	$\{\{A, B\}\}$	$\{\}$

Fig. 2. A causal matrix (left) and two potential mappings onto Petri nets (right).

Figure 2 shows two WF-nets illustrating the need for "silent transitions" of the form m_{t_1,t_2}. The dynamics of the WF-net shown in Figure 2(a) is consistent with the causal matrix. If we try to remove the silent transitions, the best candidate seems to be the WF-net shown in Figure 2(b). Although this is a sound WF-net capturing incorporating the behavior of the WF-net shown in Figure 2(a), the mapping is *not* consistent with the causal matrix. Note that Figure 2(b) allows for a firing sequence where B is followed by C. This does not make sense because $C \notin \bigcup O(B)$ and $B \notin \bigcup I(C)$. Therefore, we use the mapping given in Definition 6 to give Petri-net semantics to causal matrices.

It is easy to see that a causal matrix defines a WF-net. However, note that the WF-net does not need to be sound.

Lemma 2. *Let* $CM = (A, C, I, O)$ *be a causal matrix.* $\Pi^N_{CM \rightarrow PN}(CM)$ *is a WF-net.*

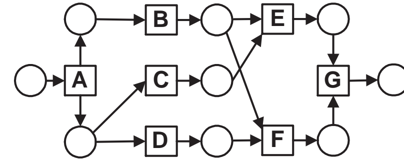
Proof. It is easy to verify the three properties mentioned in Definition 2. Note that the "short-circuited" C is strongly connected and that each m_{t_1,t_2} transition makes a similar connection in the resulting Petri net. □

Figure 3 shows that despite the fact that $\Pi^N_{CM \rightarrow PN}(CM)$ is a WF-net, the introduction of silent transitions may introduce a problem. Figure 3(b) shows the WF-net based on Definition 6, i.e., the naive mapping. Clearly, Figure 3(b) is not sound because there are two potential deadlocks, i.e., one of the input places of E is marked and one of the input places of F is marked but none of them is enabled. The reason for this is that the choices introduced by the silent transitions are not "coordinated" properly. If we simply remove the silent transitions, we obtain the WF-net shown in Figure 3(a). This network is consistent with the causal matrix. This can easily be checked because applying the mapping $\Pi_{PN \rightarrow CM}$ defined in Definition 5 to this WF-net yields the original causal matrix shown in Figure 3.

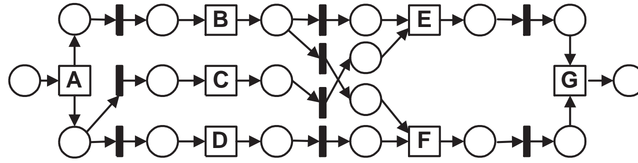
Figures 2 and 3 show a dilemma. Figure 2 demonstrates that silent transitions are needed while Figure 3 proves that silent transitions can be harmful. There are two ways to address this problem taking the mapping of Definition 6 as a starting point.

First of all, we can use *relaxed soundness* [7] rather than soundness [1]. This implies that we only consider so-called sound firing sequences and thus avoid the

ACTIVITY	*INPUT*	*OUTPUT*
A	$\{\}$	$\{\{B\},\{C,D\}\}$
B	$\{\{A\}\}$	$\{\{E,F\}\}$
C	$\{\{A\}\}$	$\{\{E\}\}$
D	$\{\{A\}\}$	$\{\{F\}\}$
E	$\{\{B\},\{C\}\}$	$\{\{G\}\}$
F	$\{\{B\},\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\},\{F\}\}$	$\{\}$



(a) Mapping without silent transitions.



(b) Naive mapping.

Fig. 3. Another causal matrix (left) and two potential mappings onto Petri nets (right).

two potential deadlocks in Figure 3(b). See [7] for transforming a relaxed sound WF-net into a sound one.

Second, we can change the firing rule such that silent transitions can only fire if they actually enable a non-silent transition. The enabling rule for non-silent transitions is changed as follows: *a non-silent transition is enabled if each of its input places is marked or it is possible to mark all input places by just firing silent transitions*, i.e., silent transitions only fire when it is possible to enable a non-silent transition. Note that non-silent and silent transitions alternate and therefore it is easy to implement this semantics in a straightforward and localized manner.

In this paper we use the second approach, i.e., a slightly changed enabling/-firing rule is used to specify the semantics of a causal matrix in terms of a WF-net. This semantics allows us also to define a notion of fitness required for the genetic algorithms. Using the Petri-net representation we can play the "token game" to see how each event trace in the log fits the individual represented by a causal matrix.

5.3 A More Sophisticated Mapping

Although not essential for the genetic algorithms, we elaborate a bit on the dilemma illustrated by figures 2 and 3. The dilemma shows that the causal net representation is slightly more expressive than ordinary Petri nets. (Note the earlier comment on expressiveness!) Therefore, it is interesting to see which causal matrices can be directly mapped onto a WF-net without additional silent transitions. For this purpose we first define a mapping $\Pi^R_{CM\rightarrow PN}$ which only works for a restricted class of causal matrices.

Definition 7 **($\Pi^R_{CM\rightarrow PN}$).** *Let $CM = (A, C, I, O)$ be a causal matrix. The restricted Petri net mapping of is a tuple $\Pi^R_{CM\rightarrow PN}(CM)$, where*

- $X = \{(T_i, T_o) \in \mathcal{P}(A) \times \mathcal{P}(A) \mid \forall_{t \in T_i}\ T_o \in O(t) \ \wedge\ \forall_{t \in T_o}\ T_i \in I(t)\}$,
- $P = X \cup \{i, o\}$,
- $T = A$,
- $F = \{(i,t) \mid t \in T \wedge \overset{C}{\bullet} t = \emptyset\} \cup \{(t,o) \mid t \in T \ \wedge\ t \overset{C}{\bullet} = \emptyset\} \cup \{((T_i, T_o), t) \in X \times T \mid t \in T_o\} \cup \{(t, (T_i, T_o)) \in T \times X \mid t \in T_i\}$.

If we apply this mapping to the causal matrix shown in Figure 3, we obtain the WF-net shown in Figure 3(a), i.e., the desirable net without the superfluous silent transitions. However, in some cases the $\Pi^R_{CM \to PN}$ does not yield a WF-net because some connections are missing. For example, if we apply $\Pi^R_{CM \to PN}$ to the causal matrix shown in Figure 2, then we obtain a result where there are no connections between A, B, C, and D. This makes sense because there does not exist a corresponding WF-net. This triggers the question whether it is possible to characterize the class of causal matrices for which $\Pi^R_{CM \to PN}$ yields the correct WF-net.

Definition 8 (Simple). *Let $CM = (A, C, I, O)$ be a causal matrix. CM is simple if and only if* $\forall_{t_A, t_B \in T}\ \forall_{T_A \in O(t_A)}\ \forall_{T_B \in O(t_B)}\ \forall_{t_C \in (T_A \cap T_B)}\ \forall_{T_C \in I(t_C)} \{t_A, t_B\} \subseteq T_C \ \Rightarrow\ T_A = T_B$ *and* $\forall_{t_A, t_B \in T}\ \forall_{T_A \in I(t_A)}\ \forall_{T_B \in I(t_B)}\ \forall_{t_C \in (T_A \cap T_B)}\ \forall_{T_C \in O(t_C)} \{t_A, t_B\} \subseteq T_C \ \Rightarrow\ T_A = T_B$.

Clearly the causal matrix shown in Figure 3 is simple while the one in Figure 2 is not. The following lemma shows that $\Pi^R_{CM \to PN}$ provides indeed the correct mapping if the causal matrix is simple.

Lemma 3. *Let $CM = (A, C, I, O)$ be a causal matrix. If CM is simple, then each of the following properties holds:*

(i) $\forall_{(t_1, t_2) \in C}\ \exists_{T_1, T_2 \in \mathcal{P}(A)}\ t_1 \in T_1 \ \wedge\ t_2 \in T_2 \ \wedge\ (\forall_{t \in T_1}\ T_2 \in O(t)) \ \wedge\ (\forall_{t \in T_2}\ T_1 \in I(t))$,

(ii) $\Pi^R_{CM \to PN}(CM)$ *is a WF-net, and*

(iii) $\Pi_{PN \to CM}(\Pi^R_{CM \to PN}(CM)) = CM$.

Proof. We only provide a sketch of the full proof (a more detailed proof is beyond the scope of this paper). The first property can be derived by using the following observation: $(t_1, t_2) \in C$ iff $\exists_{T_2 \in O(t_1)} t_2 \in T_2$ iff $\exists_{T_1 \in O(t_2)} t_1 \in T_1$. Hence there is exactly one T_1 and T_2 such that $t_1 \in T_1$, $t_2 \in T_2$, $T_2 \in O(t_1)$, and $T_1 \in O(t_2)$. For $t \in T_1$ we need to prove that $T_2 \in O(t)$. This follows from the definition of simple by taking $t_A = t_1$ and $t_B = t$. The other cases are similar. The second property follows from the first one because if $(t_1, t_2) \in C$ then a connecting place between t_1 and t_2 is introduced by the set X. The rest of the proof is similar to the proof of Lemma 2. The third property can be shown using similar arguments. Note that no information is lost during the mapping onto the WF-net $\Pi^R_{CM \to PN}(CM)$ and that $\Pi_{PN \to CM}$ retranslates the sets T_i and T_o in the places of X to functions I and O. □

In this section, we discussed the relation between the representation used by our genetic algorithm and Petri nets. We used this relation to give semantics to our representation. It was shown that this representation is slightly more expressive than Petri nets because any WF-net can be mapped into causal matrix while the reverse is only possible after introducing silent transitions and modifying the firing rule or using relaxed soundness. We also characterized the class of causal matrices that can be mapped directly. In the next sections, we will demonstrate the suitability of the representation for genetic process mining.

6 Genetic Algorithm

In this section we explain how our genetic algorithm (GA) works. Figure 4 describes its main steps. In the following subsections (6.1 – 6.3) we roughly explain the most important building blocks of our genetic approach: (i) the initialization process, (ii) the fitness measurement, and (iii) the genetic operators. For a more detailed explanation about the algorithm we refer to [14].

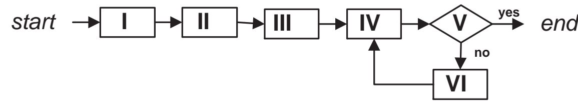


Fig. 4. Main steps of our genetic algorithm. (I) Read the event log. (II) Calculate dependency relations among activities. (III) Build the initial population. (IV) Calculate individuals' fitness. (V) Stop and return the fittest individuals? (VI) Create next population by using the genetic operators.

6.1 Initial Population

The initial population is randomly built by the genetic algorithm. As explained in Section 4, individuals are causal matrices. When building the initial population, we roughly follow Definition 4. Given a log, all individuals in any population of the genetic algorithm have the same set of activities (or tasks) A. This set contains the tasks that appear in the log. However, the causality relation C and the condition functions I and O may be different for every individual in the population. Additionally, to guide the GA algorithm during the building of the initial population, the initialization of the causality relation C is supported by the dependency measure heuristics [18]. The motivation behind this heuristic is simple. If, in the event log, the pattern t_1t_2 appears frequently and t_2t_1 only as an exception, then there is a high probability that t_1 and t_2 are in the causality relation (i.e., $(t_1, t_2) \in C$). The conditions functions I and O are randomly built. As a result, the initial population can have any individual in the search space defined by a set of activities A. The higher the amount of tasks that a log contains, the bigger this search space. Given the event log in Table 1, Table 4 shows two individuals that could be created during the initialization.

Individual1

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C, B, G\}\}$	$\{\}$

Individual2

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E, F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C\}, \{B\}, \{G\}\}$	$\{\}$

Table 4. Two randomly created individuals for the log in Table 1.

6.2 Fitness Calculation

If an individual in the genetic population correctly describes the registered behavior in the event log, the fitness of that individual will be high. In our approach the fitness is strongly related to the number of correctly parsed traces from the event log. Note that in case of noisy situation, we cannot aim at mining a process model that can correctly parse *all* traces, because the traces with noise cannot also be parsed by the desired model.

The parsing technique we use for the causal matrix is very similar to the firing rule for Petri nets as discussed in Section 5.2. We use the naive semantics with silent transitions that only fire when needed and simply play the "token game". When the activity to be parsed is not enabled, the parsing process does not stop. The problem is registered and the parsing proceeds as if the activity was enabled (conceptually, this is equivalent to adding the necessary missing tokens in the Petri net to enable the activity and, then, firing it). We adopt this parsing semantics because it is more robust to noisy logs and it gives more information about the fitness of the complete process models (i.e not biased to only the first part of the process model). In a noise-free situation, the fitness of a model can be 1 (or 100%) (i.e. all traces can be parsed). In practical situations, the fitness value ranges from 0 to 1. The exact fitness of an individual to a given log is given by the formula:

$$\mathit{fitness} = 0.40 \times \frac{allParsedActivities}{numberOfActivitiesAtLog} \; + \; 0.60 \times \frac{allProperlyCompletedLogTraces}{numberOfTracesAtLog}$$

where: *numberOfActivitiesAtLog* is the number of activities in the log. For instance, the log shown in Table 1 has 18 activities. *numberOfTracesAtLog* is the number of log traces, e.g., in Table 1 there are 4. *allParsedActivities* is the sum of parsed activities (i.e. activities that could fire without the artificial addition of tokens) for all log traces. *allProperlyCompletedLogTraces* is the number of log traces that were properly parsed (i.e. after the parsing the end place is the only one to be marked).

6.3 Genetic Operations

We use elitism, crossover and mutation to build the population elements of the next genetic generation. Elitism means that a percentage of the fittest individuals in the current generation is copied into the next generation. Crossover and mutation are the basic genetic operators. Crossover creates new individuals (offsprings) based on the fittest individuals (parents) in the current population. So, crossover recombines the fittest material in the current population in the hope that the recombination of useful material in one of the parents will generate an even fitter population element. The mutation operation will change some minor details of a population element. The hope is that the mutation operator will insert new useful material in the population. The genetic algorithm (GA) stops when: (i) it finds an individual whose fitness is 1; *or* (ii) it computes n generations, where n is the maximum number of generation that is allowed; *or* (iii) the fittest individual has not changed for $n/2$ generations in a row. If none of these conditions hold, the GA creates a new population as follows:

Input: *current population, elitism rate, crossover rate and mutation rate*
Output: *new population*

1. *Copy "elitism rate $\times$ population size" of the best individuals in the current population to the next population.*
2. *While there are individuals to be created do:*
   (a) *Use tournament selection to select parent1.*
   (b) *Use tournament selection to select parent2.*
   (c) *Select a random number r between 0 (inclusive) and 1 (exclusive).*
   (d) *If r less than the crossover rate:*
   then *do crossover with parent1 and parent2. This operation generates two offsprings: offspring1 and offspring2.*
   else *offspring1 equals parent1 and offspring2 equals parent2.*
   (e) *Mutate offspring1 and offspring2. (This step is only needed if the mutation rate is non-zero.)*
   (f) *Copy offspring1 and offspring2 to the new population.* [6]
3. *Return the new population.*

Tournament Selection To select a parent the tournament selection algorithm randomly selects 5 individuals and returns the fittest individual among the five ones.

Crossover The most important and complex genetic operation in our genetic approach is the crossover operation. Starting point of the crossover operation are the two parents (i.e. parent1 and parent2). The result of applying the crossover

[6] Note: If the population size is n and the new population has already $n-1$ individuals, then only *offspring1* is copied into this new population.

operation are two offsprings (offspring1 and offspring2). First, the crossover algorithm randomly selects an activity t to be the *crossover point*. This means that the INPUT and OUTPUT of t in parent1 will be recombined with the INPUT and OUTPUT of t in parent2. Second, parent1 is copied to offspring1 and parent2 to offspring2. Third, the algorithm randomly selects a *swap point* for the INPUT(t) sets in both offsprings and another *swap point* for the OUTPUT(t) sets. The swap point determines which subsets of the INPUT/OUTPUT of t in the offspring are going to be *swapped* (or exchanged). A random swap point is chosen for every INPUT/OUTPUT and offsprings. The respective INPUT and OUTPUT sets of the crossover point at the two offsprings are then recombined by interchanging the subsets from the swap point until the end of the set. The recombined INPUT/OUTPUT sets are then checked to make sure that they are proper partitions. Finally, the two offsprings undergo a repair operation called *update related elements*.

Update Related Elements When the parents (and consequently the offsprings) have different causal matrices, the crossover operation may generate inconsistencies. Note that the boolean expression may contain activities whose respective cell in the causal matrix is zero. Similarly, an activity may not appear in the boolean expression after the crossover and the causal matrix still has a non-zero entry for it. So, after the INPUT/OUTPUT sets have being recombined, we need to check the consistency of the recombined sets with respect to the other activities boolean expressions and the causal matrix. When they are inconsistent, we need to update the causal matrix and the related boolean expressions of the other activities. As an example, assume *Parent1* equals *Individual1* and *Parent2* equals *Individual2* in Table 4. These two parents undergo crossover and mutation to generate two offsprings. Let activity D be the randomly selected crossover point. Since INPUT1(D) equals INPUT2(D), the crossover has no real effect for D's INPUT. Let us look at the D's OUTPUT sets. Both D's OUTPUT sets have a single subset, so the only possible swap point to select equals 0, i.e., before the first and only element. After swapping the subsets *Offpring1* (*Parent1* after crossover) has INPUT1(D)= $\{\{A\}\}$ and OUTPUT1(D)= $\{\{E, F\}\}$. Note that OUTPUT1(D) now contains F. So, the *update related elements* algorithm makes INPUT1(F)= $\{\{D\}\}$. *Offspring2* is updated in a similar way. The two offsprings are shown in Table 5.

Mutation The mutation works on the INPUT and OUTPUT boolean expressions of an activity. For every activity t in an individual, a new random number r is selected. Whenever r is less than the "mutation rate" [7], the subsets in INPUT(t) are randomly merged or split. The same happens to OUTPUT(t). As an example, consider *Offspring1* in Table 5. Assume that the random number r was less than the mutation rate for activity D. After applying the mutation,

[7] The mutation rate determines the probability that an individual's task undergoes mutation.

Offspring1

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E, F\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\{D\}\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C, B, G\}\}$	$\{\}$

Offspring2

ACTIVITY	INPUT	OUTPUT
A	$\{\}$	$\{\{B, C, D\}\}$
B	$\{\{A\}\}$	$\{\{H\}\}$
C	$\{\{A\}\}$	$\{\{H\}\}$
D	$\{\{A\}\}$	$\{\{E\}\}$
E	$\{\{D\}\}$	$\{\{G\}\}$
F	$\{\}$	$\{\{G\}\}$
G	$\{\{E\}, \{F\}\}$	$\{\{H\}\}$
H	$\{\{C\}, \{B\}, \{G\}\}$	$\{\}$

Table 5. Example of two offsprings that can be produced after a crossover between the two individuals in Table 4.

OUTPUT(D) changes from $\{\{E, F\}\}$ to $\{\{E\}, \{F\}\}$. Note that this mutation does not change an individual's causal relations, only its AND-OR/join-split may change.

7 Some experimental results

To test our approach we applied the algorithm to many examples, mostly artificially generated and some based on real-life logs [14]. In this paper we only consider two WF-nets; one with 8 and one with 12 activities. Both models contain concurrency and loops. The WF-net with 8 activities corresponds to the first example in this paper, i.e., the Petri net shown in Figure 1. The other WF-net represents a completely different process model. For each model we generated 10 different event logs with 1000 traces. Without noise the rediscovering of the underlying WF-nets was no problem for the genetic algorithm. To test the behavior of the genetic algorithm for event logs with noise, we used 6 different noise types: *missing head*, *missing body*, *missing tail*, *missing activity*, *exchanged activities* and *mixed noise*. If we assume a event trace $\sigma = t_1...t_{n-1}t_n$, these noise types behave as follows. *Missing head, body* and *tail* randomly remove subtraces of activities in the head, body and tail of σ, respectively. The head goes from t_1 to $t_{n/3}$. The body goes from $t_{(n/3)+1}$ to $t_{(2n/3)}$. The tail goes from $t_{(2n/3)+1}$ to t_n. *Missing activity* randomly removes *one* activity from σ. *Exchanged activities* exchange two activities in σ. *Mixed noise* is a fair mix of the other 5 noise types. Real life logs will typically contain mixed noise. However, the separation between the noise types allow us to better assess how the different noise types affect the genetic algorithm.

For every noisy type, we generated logs with 5%, 10% and 20% of noise. So, every process model in our experiments has $6 \times 3 = 18$ noisy logs. For each event-log the genetic algorithms runs 10 experiments with a different random initialization. The populations had 500 individuals and were iterated for at most 100 generations. The crossover rate was 1.0 and the mutation rate was 0.01. The elitism rate was 0.01. Details about the experiments and results can be found in [14]. Here we summarize the main findings.

Overall the higher the noise percentage, the lower the probability the algorithm will come up with the original WF-net. In this particular setting the

algorithm can always handle the *missing tail* noise type. This is related to the high impact of *proper completion* in our fitness measure. Also the *exchanged activities* noise type does not harm the performance of the algorithm. This is related to the heuristic that is used during the initialization of the population. *Missing head* impacts more the algorithm because our fitness does not (yet!) punish individuals with missing tokens. *Missing body* and *missing activity* noise types are the most difficult to handle. This is also related to the heuristics during the building of the initial population because the removal of activities generates t_1t_2 "fake" subtraces that will not be counter-balanced by subtraces t_2t_1. Consequently, the probability that the algorithm will causally relate t_1 and t_2 is increased. Tables 6 and 7 contain the results obtained for the noisy logs of the two process models with 8 and 12 activities.

	noise type					
noise percentage	*Missing head*	*Missing tail*	*Missing body*	*Missing activity*	*Exchanged activities*	*Mixed noise*
5%	10/10	10/10	0/10	1/10	9/10	1/10
10%	10/10	10/10	1/10	1/10	5/10	3/10
20%	0/10	10/10	0/10	0/10	0/10	2/10

Table 6. Results of applying the genetic algorithm for noisy logs of the process model with 8 activities, i.e., Figure 1. The ratio relates the number of times the algorithm found the correct model by the number of times the algorithm ran.

	noise type					
noise percentage	*Missing head*	*Missing tail*	*Missing body*	*Missing activity*	*Exchanged activities*	*Mixed noise*
5%	10/10	10/10	0/10	0/10	10/10	2/10
10%	1/10	10/10	0/10	0/10	9/10	3/10
20%	1/10	10/10	0/10	0/10	8/10	2/10

Table 7. Results of applying the genetic algorithm for noisy logs of the process model with 12 activities.

Tables 6 and 7 show that our approach works well for relatively simple examples. Moreover in contrast to most of the existing approaches it is able to deal with noise. To improve our approach we are now refining the fitness calculation. For instance, the fitness should consider the number of tokens that remained in the individual after the parsing is finished as well as the number of tokens that needed to be added during the parsing.

The generic mining algorithm presented in this paper is supported by a plugin in the ProM framework (cf. http://www.processmining.org). Figure 5 shows screenshot of the plugin. This screenshot presents the result for the process model with 8 activities in terms of Petri nets and EPCs.

8 Conclusion

In this paper we presented our first experiences with a more global mining technique (e.g. a genetic algorithm). For convenience we did not use Petri nets for

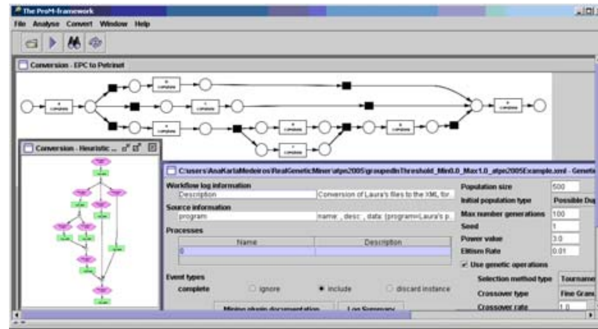


Fig. 5. A screenshot of the GeneticMiner plugin in the ProM framework analyzing the event log in Table 1 and generating the correct WF-net, i.e., the one shown in Figure 1.

the internal representation of the individuals in a genetic population. Instead we used causal matrices which are slightly more expressive. We elaborated on the relationships between our representation and Petri nets. The bottom line is that any WF-net can be mapped onto our notation and also some constructs that require duplicate activities (i.e., two transitions with the same label) or silent transitions (i.e., steps not visible in the log) can be discovered. This way the approach overcomes some of the problems with earlier algorithms. For example, it is possible to mine non-free choice constructs. The main added value of using a genetic algorithm is the ability to deal with noise and incompleteness. At this point in time we are fine-tuning our genetic algorithms based on many artificial examples (i.e., process mining based on simulation logs) and real-life examples. Experimental results on event logs with noise point out that we are on the right track on our quest to develop a genetic algorithm that mines process models. Our next steps will focus on further improvements of the fitness measurement so that it gives a better indication of the optimal fit between a process model and an event-log.

References

1. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
2. W.M.P. van der Aalst and M. Song. Mining Social Networks: Uncovering Interaction Patterns in Business Processes. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, Berlin, 2004.
3. W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.
4. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

5. R. Agrawal, D. Gunopulos, and F. Leymann. Mining Process Models from Workflow Logs. In *Sixth International Conference on Extending Database Technology*, pages 469–483, 1998.
6. J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.
7. J. Dehnert and W.M.P. van der Aalst. Bridging the Gap Between Business Models and Workflow Specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.
8. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures — Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
9. A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing. Springer-Verlag, Berlin, 2003.
10. K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In W.M.P. van der Aalst and E. Best, editors, *Application and Theory of Petri Nets 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 335–354. Springer-Verlag, Berlin, 2003.
11. J. Herbst. A Machine Learning Approach to Workflow Management. In *Proceedings 11th European Conference on Machine Learning*, volume 1810 of *Lecture Notes in Computer Science*, pages 183–194. Springer-Verlag, Berlin, 2000.
12. H. Mauch. Evolving Petri Nets with a Genetic Algorithm. In E. Cantu-Paz and J.A. Foster et al., editors, *Genetic and Evolutionary Computation (GECCO 2003)*, volume 2724 of *Lecture Notes in Computer Science*, pages 1810–1811. Springer-Verlag, Berlin, 2003.
13. A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow Mining: Current Status and Future Directions. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, Berlin, 2003.
14. A.K.A. de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Using Genetic Algorithms to Mine Process Models: Representation, Operators and Results. BETA Working Paper Series, WP 124, Eindhoven University of Technology, Eindhoven, 2004.
15. J.H. Moore and L.W. Hahn. Petri Net Modeling of High-Order Genetic Systems Using Grammatical Evolution. *BioSystems*, 72(1-2):177–86, 2003.
16. J.H. Moore and L.W. Hahn. An Improved Grammatical Evolution Strategy for Hierarchical Petri Net Modeling of Complex Genetic Systems. In G.R. Raidl et al., editor, *Applications of Evolutionary Computing, EvoWorkshops 2004*, volume 3005 of *Lecture Notes in Computer Science*, pages 63–72. Springer-Verlag, Berlin, 2004.
17. J.P. Reddy, S. Kumanan, and O.V.K. Chetty. Application of Petri Nets and a Genetic Algorithm to Multi-Mode Multi-Resource Constrained Project Scheduling. *International Journal of Advanced Manufacturing Technology*, 17(4):305–314, 2001.
18. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.