Mandatory Resume

Process mining enables organizations to get insights about their business process executions. Among the key tasks in process mining, process discovery is a central task that aims to construct interpretable process models that reflect recorded behavior, as process models serve as the baseline for many other tasks. These models are evaluated along four quality dimensions: fitness, precision, generalization, and simplicity. While recent state-of-the-art discovery methods, such as Inductive Miner and Split Miner, are efficient and widely adopted, they lack the ability to balance the four quality dimensions. In contrast, genetic algorithms offer flexibility and potential for modeling complex dependencies, but have historically lagged in both performance and adoption due to their slow convergence and limited scalability.

We revisit the genetic approach to process discovery and propose a new genetic process discovery algorithm named Genetic Tree Miner (GTM). The GTM algorithm is designed to efficiently explore the process model search space while maintaining high-quality model output, and with a significant speed-up in discovery time compared to other genetic-based discovery algorithms. Several improvements distinguish GTM from prior genetic-based techniques. The population of process models in the GTM is represented as process trees because of their clear hierarchical structure, which ensures soundness. To push the initial population towards high-quality models, we create individuals by sublog samples combined with discovery via the inductive miner. This approach provides individual process models with some correct behavior instead of complete randomness, but still ensures population diversity by the sublog samples. This approach intends to facilitate fast convergence towards an acceptable process model.

One of the main reasons for genetic algorithms' known low performance is due to the objective function evaluation, which is a time-consuming step. To accelerate the evaluation step, we introduce a C++ implementation of fitness and precision metrics, which yields objective function evaluation speedup compared to traditional Java/Python-based tools. Another technique to lower the computationally expensive task of objective function evaluation is a log sampling strategy that filters the event log depending on the number of unique traces in order to balance the accuracy and computational costs. The search for better process models works by performing crossovers and tailored mutation operations. Crossovers are done by swapping subtrees of two process trees, and mutations can either be repositioning an activity leaf, removing a subtree, changing an operator type, or inserting an activity as a loop. In our crossover and mutation implementation, we ensure both model validity and variety. To guide the search, we implement a refined objective function where we emphasize a balance of fitness, precision, and two simplicity metrics to avoid overly complex and overfitted process models. The GTM includes various hyperparameters, and we intend to propose a robust default set of hyperparameter values that are applicable across diverse event logs. This is done by using Bayesian Optimization of the hyperparameters performed on three event logs, and using these results to identify the default values.

Through an extensive experimental evaluation on 13 publicly available real-life event logs, we benchmark GTM against Inductive Miner and Split Miner. Results show that GTM consistently outperforms both methods in terms of F1-score, achieving competitive performance on fitness and precision while maintaining reasonable simplicity. GTM-10 (10-second timeout) already surpasses Inductive Miner in most cases; with increased time (GTM-60 and GTM-300), GTM dominates on F1-score and performs on par in generalization. Our results demonstrate that genetic algorithms remain a viable and competitive strategy for process discovery, provided that evaluation, generation, and search strategies are carefully optimized. The GTM successfully narrows the gap between flexibility and efficiency in genetic process discovery. We provide open-source access to the GTM implementation, including datasets, benchmarking scripts, and documentation.

All You Need Is Evolution: Rethinking Genetic Algorithms for Process Discovery

Jeppe Berg Axelsen, Frederik Hecter Kowalski, and Richard Nygård

Aalborg University

Keywords: process discovery · genetic algorithms · empirical evaluation.

Abstract. Process mining focuses on deriving accurate and understandable process models from event logs, with the goal of balancing key quality dimensions: fitness, precision, generality, and simplicity. Genetic algorithms have shown promise in navigating this trade-off due to their ability to explore a wide search space. However, traditional genetic process discovery approaches often suffer from long convergence times and typically fall short when compared to state-of-the-art algorithms such as the Inductive Miner and Split Miner. We revisit genetic process discovery and propose a novel enhancement to the evolutionary framework that significantly improves both its efficiency and effectiveness. Our approach introduces improvements such as the ability to generate good initial populations, highly efficient fitness estimation, and adaptive log filtering, which accelerates convergence and guides the search toward higher-quality models. Through extensive evaluation on benchmark event logs, we demonstrate that our enhanced genetic algorithm achieves competitive performance, producing process models that rival those discovered by leading algorithms in terms of fitness, precision, and structural quality. These results suggest that, with the right improvements, genetic approaches can play a valuable role in the process discovery toolbox.

1 Introduction

In today's data-driven world, many businesses increasingly generate and store vast amounts of data during the execution of their real-world business processes. This data contains valuable insights into how the processes are actually executed, which can differ from how they are intended to work. Therefore, this data contains information which have the potential to help improve these processes and enable a business to become more efficient. These potentials can be achieved using techniques from process mining [1], which is a field that bridges the gap between data science and process science. The process mining techniques can help reconstruct process flows, identify bottlenecks, detect deviations, and recommend improvements.

One of the main tasks in process mining is process discovery [1, p. 163], which we provide an overview of in Figure 1. The starting point is an interaction between a real-world process and a software system. The system captures the execution of a process and stores the information as event logs, which contain detailed records of each activity performed. An event log serves as the input to a process discovery algorithm, which aims to derive a process model that reflects the real-world process flow. A process model serves as a basis for further process analysis tasks, so the discovery algorithm's ability to derive a high-quality process model is crucial. Many algorithms exist for process discovery, but they face different challenges such as dealing with noise, complexity, concurrency, and incomplete data. These challenges make the process discovery a field where researching and developing better algorithms is still relevant.

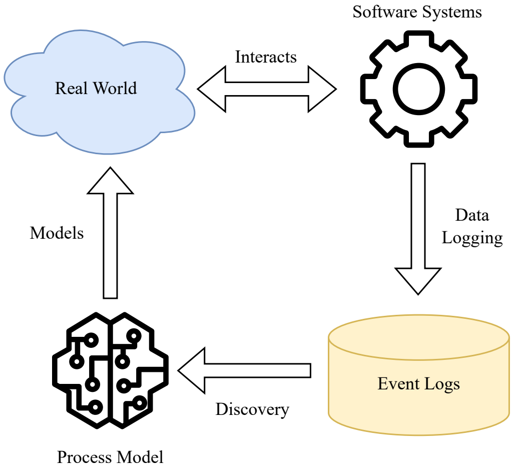


Fig. 1: Overview of process discovery elements and their relations.

Genetic algorithms (GAs) [2], which are used for search and optimization problems in large and complex solution spaces, have already been used for discovering process models, but have been criticized for their slow performance due to expensive computational operations. As a result, these types of algorithms are often underrepresented in benchmark competitions alongside more modern discovery methods, such as the Inductive Miner (IM) [3] and Split Miner (SM) [4].

We introduce a refined genetic algorithm for process discovery, Genetic Tree Miner (GTM), which is enhanced in terms of speed and performance. Our optimizations of the genetic approach enable GTM to be a competitive discovery algorithm compared to IM and SM on commonly used benchmark event logs. A key optimization is that we implemented a faster approach to calculate fitness and precision metrics in C++ rather than using Python and Java tools. To search for better process models, we focus on fitness [5], precision [6], and a refined simplicity metric. We find a single set of hyperparameters for the GTM, using Bayesian Optimization [7] techniques. We ensure that all activities recorded in the event log are represented in the discovered model. The results of our benchmark test show that GTM is a competitive algorithm, and that GAs in general are relevant in the research for better process discovery algorithms.

1.1 Related Work

Early process discovery approaches extract simple control-flow patterns from logs. One of the first process discovery algorithms developed was the α-algorithm [8]. It analyzes directly follows relationships in the event log to construct a Petri net by identifying patterns such as sequence, concurrency, and choice. While its theoretical foundation is simple and sound, Alpha Miner struggles with noise, infrequent behavior, and non-free-choice constructs, often producing unsound or overly simplistic models in real-life logs.

Later, the Heuristic Miner [9] was introduced. Heuristic Miner addresses some of the limitations of the α-algorithm by incorporating frequency-based heuristics. It computes dependency measures between activities to construct a more robust process model, capable of handling incomplete and imprecise logs. The Heuristic Miner has been shown to produce relatively good fitness and precision on noisy logs, but it has difficulties producing sound and simple models.

ILP Miner [10] frames process discovery as an optimization problem, utilizing integer linear programming to derive Petri nets that perfectly reproduce the event log. This approach guarantees soundness and fitting precision under the assumption that the event log is complete. However, ILP Miner suffers from high computational complexity and poor scalability to large or noisy logs [11].

This leads us to the state-of-the-art methods, which are both efficient and provide guarantees on the structure of the mined models. The most recognized process discovery algorithm is the IM [3], which constructs block-structured process trees by recursively splitting the DFG of the event log, which is constructed using frequency thresholds. Its core strength lies in its guarantee of soundness: the resulting model is always deadlock-free and complete by construction. This makes IM particularly suited for applications where semantic correctness and replayability are crucial. IM also handles infrequent behavior gracefully by allowing configurable noise thresholds during log splitting. Another core strength of the IM is that the discovered model is fitting. In recent years, the SM [4] [12] has been emerging as a reliable process discovery algorithm designed to balance model quality in terms of fitness, precision, generalization, and simplicity. It begins by constructing a DFG from the event log, filtering infrequent behavior based on configurable thresholds. It then applies a gateway-splitting procedure to identify and differentiate between exclusive choices and parallel branches, enabling SM to discover good and well-balanced process models. If the DFG is acyclic, SM guarantees the production of a sound process model; however, if the DFG is cyclic, only a deadlock-free process model is guaranteed.

The use of GAs for the process discovery problem has been investigated [13–17]. The pioneering work [13] showed the initial applications of GAs to process discovery; nevertheless, it fails to show the practical applications on real-life event logs. An extension of this work is the Evolutionary Tree Miner (ETM) [14], which is a flexible algorithm allowing the user to guide the discovered process tree by weighting the four quality measures. ETM uses process trees as the internal representation and therefore guarantees sound workflow nets. ETM introduces node mutation, subtree removal, and node addition. However, they provide a limited evaluation of ETM compared to other methods on a broad variety of real-life event logs. An evaluation of ETM on real-life eventlogs [4], shows that even after 60 minutes of running time, the ETM struggles to produce well-balanced process models.

In contrast to ETM and other Genetic-based process discovery approaches, our method introduces several innovations that enhance performance, scalability, and model quality. It is often the case in GAs that the objective function evaluation is the most time-consuming part. To mitigate this, we provide efficient and fast fitness and precision implementation in C++, offering significant runtime improvements over previous tools developed in Java or Python. Additionally, we introduce a mutation specifically designed to capture simple self-loop behaviors. Our objective function is refined by removing the generalization metric and improving the simplicity component to more effectively penalize unnecessary model complexity. We conduct a hyperparameter search using Bayesian Optimization [7] to find suitable and robust hyperparameter values that can be used by the end user. Unlike some existing approaches, our method avoids caching of intermediate process trees, which simplifies the architecture and reduces memory usage. We also ensure that every activity in the event log is uniquely represented in the discovered model, preventing activity omission. Furthermore, we refrain from reducing the process tree after discovery, allowing both beneficial and potentially problematic constructs to remain visible for inspection. Collectively, these contributions lead to a more efficient and effective genetic approach to process discovery, as demonstrated through a comprehensive evaluation on real-life event logs.

---

2 Background

In this section, we present an overview of relevant concepts to establish a foundation for our process discovery algorithm. This includes key concepts of process mining, process model representations, and quality measures commonly used to evaluate discovered process models.

2.1 Event Logs

A fundamental concept in process mining is the event log, which contains records of executions of real-world business processes. Each trace in the log corresponds to a single instance of the process and is a sequence of events, where each event includes at least an activity label, a case identifier, and a timestamp. We define an abstraction of an event log, which captures the essential structure of an event log while omitting timestamps and additional attributes such as resources.

Definition 1 (Simple event log). *[1] Let A be a set of activity names. The simple trace π is a sequence of activities, i.e., $\pi \in A^+$. A simple event log L is a multi-set of traces over A, i.e., $L \in \mathcal{B}(A^+)$.*

An example of a simple event log is shown in Figure 2a, illustrating three traces with different activity sequences.

2.2 Petri Nets

In the context of process mining, Petri nets [18] play a central role as the target model for representing discovered process behavior. Their semantics support formal reasoning, making it possible to verify properties such as soundness, precision, and fitness with respect to observed event data.

Definition 2 (Petri Net). *[1] Let A be a finite set of observable activities such that $\tau \notin A$. A (labelled)* Petri net *is a four-tuple $N = (P, T, F, \ell)$ where*

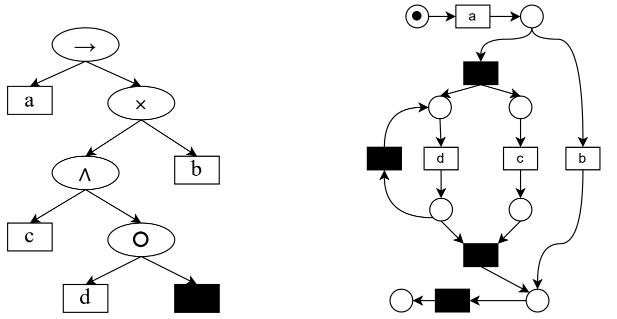
- *P is a finite set of places,*
- *T is a finite set of transitions such that $P \cap T = \emptyset$,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, and*
- *$\ell : T \to A \cup \{\tau\}$ is a labelling function assigning activity names to transitions; here τ represents the silent (unobservable) activity.*

We extend the labelling function from Definition 2 to sequences of transitions, such that $\ell(t_0 t_1 \dots t_n) = \ell(t_0)\ell(t_1)\dots\ell(t_n)$ where we define $\ell(\tau) = \epsilon$ as the empty string. It allows us to relate entire execution traces of a Petri net to the observed traces in the event log. This is essential when evaluating a Petri net to see how well it represents the behavior in its corresponding event log.

In order to formally reason about the dynamic behavior of systems modeled by Petri nets, it is essential to specify not only the net structure but also the current distribution of tokens across its places. This leads to the concept of a marked Petri net, which extends a Petri net with a marking function that captures the state of the system.

Definition 3 (Marked Petri Net). *A* marked Petri net *is a pair (N, M) where $N = (P, T, F, \ell)$ is a Petri net and where $M : P \to \mathbb{N}^0$ is a* marking *assigning a number of tokens to each place.*

Case ID	Activity	Timestamp
1	a	08:34
1	b	08:45
2	a	09:02
2	c	09:15
2	d	09:30
2	d	09:45
3	a	10:05
3	d	10:15
3	d	10:25
3	c	10:35



(a) Event log example representing the set of traces $L = \{ab, acdd, addc\}$

(b) Process tree describing the language $ab + ad^*cd^*$, black box depicts the τ action

(c) A sound WF-net with the same trace language as the process tree (filled transitions have label τ)

Fig. 2: Event log, its process tree model and the corresponding Petri net

The marking M represents the state of the Petri net at a given point in time by specifying the number of tokens in each place.

Definition 4 (Subset of a Marking). *Given a marking M, a marking M' is a* subset *of M if*

$$\forall p \in P, \quad M'(p) \leq M(p).$$

We denote this as $M' \subseteq M$.

The evolution of a marked Petri net is governed by the firing rule of transitions, which consume and produce tokens according to the flow relation F and the current marking. This can be formulated as follows:

Definition 5 (Firing Rule). *A transition $t \in T$ can fire in a marking M and reach a marking M', written $M[t\rangle M'$, if (i) $M(p) > 0$ for every $p \in P$ such that $(p, t) \in F$ (t is enabled in M) and (ii) $M'(p) = M(p) - F(p, t) + F(t, p)$ where $F(x, y) = 1$ if $(x, y) \in F$, otherwise $F(x, y) = 0$.*

For a sequence of transitions $t_0, t_1, \ldots, t_n \in T$, we write $M_0[t_0 t_1 \ldots t_n\rangle M_n$ (or simply $M_0 \to^* M_n$ if we are not interested in transition names) if there are markings $M_1, \ldots, M_{n-1}$ such that $M_i[t_i\rangle M_{i+1}$ for all i, $0 \leq i < n$.

To facilitate reasoning about the structure and behavior of Petri nets, it is useful to introduce the notions of preset and postset of a transition.

Definition 6 (Preset and Postset of a transition). *[5] The preset of a transition, $\bullet t$, is the set of all places $p \in P$ such that $(p, t) \in F$. The postset of a transition, $t\bullet$, is the set of all places $p \in P$ such that $(t, p) \in F$.*

Intuitively, the preset $\bullet t$ consists of all input places from which a transition consumes tokens when it fires, while the postset $t\bullet$ includes all output places to which tokens are produced. These sets are fundamental in analyzing the flow of tokens and the causal relationships between events in the modeled process. Traces describe the observable behavior of a Petri net by recording sequences of executed activities. A trace corresponds to a sequence of transitions that can be fired from the initial marking, and whose labels form the observed activity sequence.

Definition 7 (Trace of a Petri Net). *A sequence of activities* $\pi \in A^*$ *is a* trace *of a marked Petri net* (N, M_0) *if there is a firing sequence of transitions* $t_0, t_1, \ldots, t_n \in T$ *such that* $M_0[t_0 t_1 \ldots t_n\rangle M_n$ *for some marking* M_n *and* $\ell(t_0 t_1 \ldots t_n) = \pi$*. A trace is said to be* complete *if it cannot be extended to a longer trace.*

To model complete processes with well-defined starting and ending points, the notion of a workflow net (WF-net) is often used. A WF-net ensures that every component of the Petri net contributes to the execution of a process from an initial to a final state. Soundness further guarantees desirable behavioral properties, such as proper completion and absence of deadlocks or leftover tokens.

Definition 8 (Sound WF-net). *A Petri net* $N = (P, T, F, \ell)$ *is a WF-net if*

- P *contains a unique start place* i *with no ingoing arcs,*
- P *contains a unique end place* o *with no outgoing arcs,*
- *all transitions and places are on some path from* i *to* o*.*

Let M_i *be the initial marking where* $M_i(i) = 1$ *and* M_f *be the final marking where* $M_f(o) = 1$ *and all other places in* M_i *and* M_f *have no other tokens. A WF-net is* sound *if*

- $M \to^* M_f$ *for all markings* M *where* $M_i \to^* M$*, and*
- $M_i \to^* M$ *and* $M(o) \geq 1$ *then* $M = M_f$*, and*
- $M_i \to^* M$ *for all places* p *then* $M(p) \leq 1$.

The Petri net in Figure 2c is an example of a sound WF-net. This ensures that, starting from the initial marking, the process can always reach proper completion, and that no extraneous tokens or deadlocks remain at the end of execution.

Definition 9 (Language of a WF-net). *Given a sound WF-net* N*, its* language $\mathcal{T}$ *is the set of all activity sequences in* A^* *for which there exists a firing sequence of transitions leading from the initial marking* M_i *to the final marking* M_f*. Formally,*

$$\mathcal{T}(N, M_i, M_f) = \{\ell(t_0 \ldots t_n) \mid t_0, \ldots, t_n \in T,\ M_i[t_0 \ldots t_n\rangle M_f\}.$$

In Figure 2c, the set of complete traces of the net can be described by the regular expression $ab + ad^*cd^*$.

2.3 Process Trees

An alternative model representation is the process tree [19], which has a hierarchical structure and guarantees soundness by design. Definition 10 inductively defines the set of process trees $\mathcal{P}$, together with the set of traces $\mathcal{T}(Q) \subseteq 2^{A^*}$ generated by the process tree $Q \in \mathcal{P}$.

Definition 10 (Process tree). *Let* A *be a finite set of activities with* $\tau \notin A$*. The set of process trees* $\mathcal{P}$ *over* A *is the smallest set such that* $\tau \in \mathcal{P}$ *where* $\mathcal{T}(\tau) = \{\epsilon\}$*,* $a \in \mathcal{P}$ *for any* $a \in A$ *where* $\mathcal{T}(a) = \{a\}$ *and if* $Q_1, \ldots, Q_n \in \mathcal{P}$ *for* $n \geq 2$ *then*

- $Q \equiv \rightarrow(Q_1, \ldots, Q_n) \in \mathcal{P}$ *(sequential composition) where* $\mathcal{T}(Q) = \mathcal{T}(Q_1) \circ \ldots \circ \mathcal{T}(Q_n)$*,*

- $Q \equiv \times(Q_1, \ldots, Q_n) \in \mathcal{P}$ *(choice operator) where* $\mathcal{T}(Q) = \mathcal{T}(Q_1) \cup \ldots \cup \mathcal{T}(Q_n)$*,*
- $Q \equiv \wedge(Q_1, \ldots, Q_n) \in \mathcal{P}$ *(parallel composition) where* $\mathcal{T}(Q) = \mathcal{T}(Q_1) \diamond \ldots \diamond \mathcal{T}(Q_n)$*,*
- $Q \equiv \bigcirc(Q_1, \ldots, Q_n) \in \mathcal{P}$ *(loop operator) where* $\mathcal{T}(Q) = \mathcal{T}(Q_1) \circ \big((\mathcal{T}(Q_2) \cup \ldots \cup \mathcal{T}(Q_n)) \circ \mathcal{T}(Q_1)\big)^*$.

Here $\circ$*,* $\cup$*, and* $*$ *are the classical composition, union, respectively. Kleene star operations on languages, and the shuffle (interleaving) operator* $\diamond$ *used in parallel composition is defined as* $aw \diamond a'w' = \{a \circ (w \diamond a'w')\} \cup \{a' \circ (aw \diamond w')\}$ *for* $a, a' \in A$ *and* $w, w' \in A^*$*, where* $w \diamond \epsilon = \epsilon \diamond w = \{w\}$*. It is extended to languages as follows:* $L \diamond L' = \bigcup_{w \in L, w' \in L'} w \diamond w'$*.*

An example of a process tree (in the standard graphical notation) is given in Figure 2b. The trace language includes all the traces from the log L depicted in Figure 2a. It is well known [20] that for every process tree $Q \in \mathcal{P}$ there is a sound WF-net (N) such that $\mathcal{T}(Q) = \mathcal{T}(N, M_i, M_f)$. Figure 2c shows such a WF-net converted from the process tree in Figure 2b.

2.4 Process Model Quality Metrics

When evaluating the quality of a discovered process model, it is essential to adopt a multidimensional perspective. A model's usefulness cannot be determined by a single criterion alone, as different metrics capture different aspects of quality. We can evaluate process models using the widely recognized: fitness, simplicity, precision, and generalization. Fitness measures how much of the observed behavior in the event log can be reproduced by the process model, where high fitness indicates that the model can reproduce most of the behavior in the event log. Fitness alone cannot be used to evaluate performance. An overly complex model that includes every trace in the event log has perfect fitness but limited use in practice. Simplicity measures the structural complexity of a model and favors the simplest possible models, whereas simpler models are preferred because they are easier for humans to interpret. Precision quantifies the amount of extra behavior the model allows other than what is observed, whereas models allowing too much behavior should be penalized by this dimension. Precision is closely related to the generalization dimension as it deals with a model's ability to generalize to new, unseen behavior not included in the event log. Since event logs only contain positive examples, the model should ideally allow for unseen behavior [1]. Many different measures have been proposed to calculate these four dimensions. We will here present the ones implemented in the PM4PY library [21], which we proceed to detail.

Fitness Fitness can be measured using token-based replay. Token-based replay assesses fitness by replaying each trace on the Petri net, penalizing any discrepancy between the trace and the Petri net. Replaying a trace yields four token counts: *consumed*, *produced*, *missing*, and *remaining* tokens. Following the firing rule, whenever a transition is fired in a Petri net, a token is consumed from each input place, and a new token is produced for each output place. Missing tokens are inserted whenever they are needed to replay a trace. The remaining tokens are the ones left in the Petri net after replaying the trace.

Definition 11 (Log-level Fitness). *Let* L *denote an event log and* N *denote a Petri Net. The log-level fitness of* N *is:*

$$F(L, N) = \frac{1}{2}\left(1 - \frac{\sum_{\pi \in L} m(\pi, N)}{\sum_{\pi \in L} c(\pi, N)}\right) + \frac{1}{2}\left(1 - \frac{\sum_{\pi \in L} r(\pi, N)}{\sum_{\pi \in L} p(\pi, N)}\right),$$

where m*,* c*,* r*, and* p *represent:*

- $m(\pi, N)$*: The number of missing tokens during the execution of* π *in* N*,*
- $c(\pi, N)$*: The number of consumed tokens during the execution of* π *in* N*,*
- $r(\pi, N)$*: The number of remaining tokens after the execution of* π *in* N*,*
- $p(\pi, N)$*: The number of produced tokens during the execution of* π *in* N*.*

A fitness score of $F(L, N) = 1$ indicates perfect alignment between the L and N, meaning that N fully explains all behavior observed in L without any missing or remaining tokens.

When evaluating fitness, it is important to account for silent transitions, which are transitions in a Petri net that do not correspond to any observable event in the log. These transitions play a crucial role in modeling routing behavior, such as parallelism, choice, and loops, without requiring a corresponding activity in the event log. They are used to guide the model's token flow to enable the replay of visible transitions. However, incorrect or excessive use of silent transitions can result in additional missing or remaining tokens, thus negatively impacting the fitness score. Consequently, although silent transitions are not matched to events, they still influence fitness by affecting the structural alignment between the trace and the model.

Precision Precision can be measured by comparing the state space of a Petri net execution while replaying a log. The precision measure presented in [6] uses the notion of escaping edges, which represents behavior permitted by the model but not reflected in the log.

Definition 12 (Direct Successor Function). *Let* A *be the set of all activities. Define the function*

$$\Gamma(L) : A^* \to \mathcal{P}(A)$$

such that for any prefix $\pi \in A^*$*, the value* $\Gamma(L)(\pi)$ *is the set of activities that directly follow the prefix* π *in at least one trace of the event log L. Formally,*

$$\Gamma(L)(\pi) = \{a \in A \mid \pi \circ a \circ \pi' \in L\}.$$

Definition 13 (Precision). *Let L be an event log,* π *be a trace in L, and N be a WF-net. Precision for log* L *and WF-net* N *is defined as:*

$$P(L, N) = 1 - \frac{\sum_{i=1}^{|L|} \sum_{j=1}^{|\pi|+1} |S_{ij} \setminus \Gamma(L)(\pi_1 \circ \ldots \circ \pi_j)|}{\sum_{i=1}^{|L|} \sum_{j=1}^{|\pi|+1} |S_{ij}|}$$

Where, M_{ij} *is the marking after replaying* j *event of trace* π_i *in the log and* S_{ij} *is the number of enabled transitions in marking* M_{ij}*, formally:* $S_{ij} = \{t \in T \mid M_{ij}(\bullet t) > 0\}$

When $P(L, N) = 1$, the model perfectly describes the behavior in the log (no escaping edges) and allows for no additional behavior. Vice versa, when $P(L, N)$ nears 0, the model allows an increasing amount of behavior not observed in the log.

Generalization A generalization measure captures the behavior of the model beyond the observed traces.

Definition 14 (Generalization Measure). *[22] Let T be the set of transitions in a process model. For each transition $t \in T$, let $\xi(t)$ denote the number of times transition t has been executed in the event log. The generalization of the process model is defined as*

$$Q = 1 - \frac{\sum_{t \in T} \left(\sqrt{\xi(t)}\right)^{-1}}{\mid T \mid}.$$

Definition 14 is based on the intuition that a model is generic if all transitions in the model are frequently fired. Hence, a high generalization means the process model is less specific to the event log and allows a lot of unobserved behavior.

Simplicity Simplicity is a measure of complexity uncoupled from observed behavior and thus merely accounts for the process model's structure. The goal of simplicity is to balance model complexity with the ability to represent the observed event log accurately. A simpler model is often preferred because it is easier to interpret, maintain, and communicate to stakeholders. There exist many different ways of computing the simplicity of a model, e.g., the number of edges and behavioral complexity. We use a specific simplicity detailed in [23]. This simplicity measure attributes complexity to the number of AND-splits and OR-splits. Consequently, a Petri net with many such constructs receives a low simplicity.

3 C++ Implementation of Existing Fitness and Precision Metrics

In this section, we introduce the algorithmic details on how to calculate the variant of fitness in [5] and precision in [6] using Petri nets, including how to deal with silent transitions. We implement both in C++ to improve their respective runtimes over their implementations in the PM4PY Python library.

Algorithm 1 demonstrates the calculation of fitness for a single Petri net and event log. The algorithm initializes token-based replay and starts iterating over the activities in each trace in the event log. First, it identifies the transition that corresponds to the activity a_j. Regarding ℓ^{-1}, if ℓ is not injective, the algorithm can be extended to make a random choice between the ambiguous transitions or use other more sophisticated techniques. If some transition, t, cannot fire, the algorithm attempts to move tokens into places in the preset of t by firing silent transitions. As detailed in Algorithm 3 that shows silent transitions handling, the set of places with no tokens in the preset of t is denoted as δ, and the set of places with at least one token, that are not in the preset of t, is denoted as λ. Algorithm 3 attempts to move tokens from λ to δ through the shortest possible silent transition paths until t is enabled or the maximum iteration limit is reached. If t is enabled, it is fired; otherwise, missing tokens are inserted at places in the preset of t and τ-transitions are reverted. Afterwards, trace replay continues and the token counts are updated according to Definition 11. Algorithm 1 terminates once all traces have been replayed and returns the fitness score.

Algorithm 1 Fitness

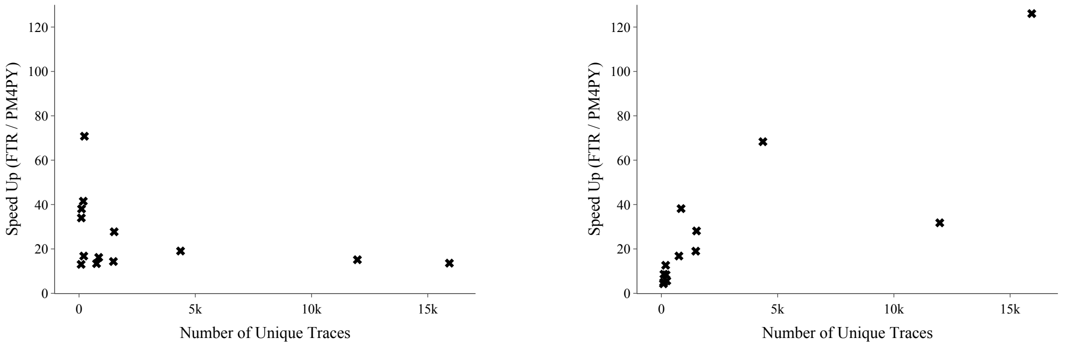
```
Require: Marked Petri net: $N = (P, T, F, \ell, M_i)$, final marking: $M_f$, event log: $L$
Initialize missing, remaining, produced, consumed $\leftarrow 0$
for $\pi = a_1 a_2 \ldots a_m \in L$ do
    $M_{curr} \leftarrow M_i$
    produced $\leftarrow$ produced $+\Sigma_{p \in P} M_i(p)$
    for $j = 1$ to $m$ do
        $t \leftarrow \ell^{-1}(a_j)$                // If $\ell$ is not injective pick a random transition such that $\ell(t) = a_j$
        if $\exists p \in \bullet t.\ M_{curr}(p) = 0$ then
            Find $t_1 \ldots t_n \in T^*$ s.t. $\ell(t_1) = \ell(t_2) = \ldots = \ell(t_n) = \tau$ and $M_{curr}[t_1 \ldots t_n t\rangle$    // Algorithm 3
            if $M_{curr}[t_1 \ldots t_k t\rangle M_{new}$ for some marking $M_{new}$ then
                $M_{curr} \leftarrow M_{new}$
                consumed $\leftarrow$ consumed $+ \sum_{k=1}^{n} \mid \bullet t_k \mid$
                produced $\leftarrow$ produced $+ \sum_{k=1}^{n} \mid t_k \bullet \mid$
            else
                for $p \in \bullet t$ do
                    if $M_{curr}(p) = 0$ then
                        $M_{curr}(p) = 1$
                        missing $\leftarrow$ missing $+1$
                    end if
                end for
            end if
        end if
        $M_{curr}[t\rangle M_{new}$
        $M_{curr} \leftarrow M_{new}$
        consumed $\leftarrow$ consumed $+ |\bullet t|$
        produced $\leftarrow$ produced $+ |t \bullet|$
    end for
    if not $M_f \subseteq M_{curr}$ then
        if exists $t_1 \ldots t_n \in T^*$ s.t. $\ell(t_1) = \ell(t_2) = \ldots = \ell(t_n) = \tau$ and $M_{curr}[t_1 \ldots t_n t\rangle M'$ and
        $M_f \subseteq M'$ then
                                                                                    // Algorithm 3
            $M_{curr} \leftarrow M'$
            consumed $\leftarrow$ consumed $+ \sum_{k=1}^{n} \mid \bullet t_k \mid$
            produced $\leftarrow$ produced $+ \sum_{k=1}^{n} \mid t_k \bullet \mid$
        else
            for $p \in M_f$ do
                if $M_{curr}(p) = 0$ then
                    $M_{curr}(p) = 1$
                    missing $\leftarrow$ missing $+1$
                end if
            end for
        end if
    end if
    consumed $\leftarrow$ consumed + $\Sigma_{p \in P} M_f(p)$
    remaining $\leftarrow$ remaining + $\Sigma_{p \in P}(M_{curr}(p) - M_f(p))$
end for
fitness $\leftarrow 0.5 \cdot (1 - \frac{missing}{consumed}) + 0.5 \cdot (1 - \frac{remaining}{produced})$
return fitness
```

Algorithm 2 Precision

```
Require: Marked Petri net: $N = (P, T, F, \ell, M_i)$, final marking: $M_f$, event log: $L$
 1: Initialize esc_edges, allowed_tasks ← 0
 2: for $\pi = a_1 a_2 \ldots a_m \in L$ do
 3:     $M_{curr} \leftarrow M_i$
 4:     for $j = 1$ to $m$ do
 5:         $t \leftarrow \ell^{-1}(a_j)$          // If $\ell$ is not injective pick a random transition such that $\ell(t) = a_j$
 6:         if $\exists p \in \bullet t.\ M_{curr}(p) = 0$ then
 7:             if exists $t_1 \ldots t_n \in T^*$ s.t. $\ell(t_1) = \ell(t_2) = \ldots = \ell(t_n) = \tau$ and $M_{curr}[t_1 \ldots t_n t\rangle M_{new}$ then
 8:                                                                                  // Algorithm 3
 9:                 $M_{curr} \leftarrow M_{new}$
10:             else
11:                 Break the for-loop and continue on line 2.
12:             end if
13:         end if
14:         allowed_tasks ← allowed_tasks + $|\{t \in T \mid \forall p \in \bullet t, M_{\text{curr}}(p) > 0\}|$
15:         esc_edges ← esc_edges + $|\{t \in T \mid \forall p \in \bullet t, M_{\text{curr}}(p) > 0\} \setminus \mathit{\Gamma}(L)[a_1 \ldots a_{j-1}]|$
16:         $M_{curr}[t\rangle M_{new}$
17:         $M_{curr} \leftarrow M_{new}$
18:     end for
19: end for
20: return $1 - \frac{\text{esc\_edges}}{\text{allowed\_tasks}}$
```

Algorithm 3 Finding sequences of silent transitions that enable transition t

```
Require: Marked Petri net: $N = (P, T, F, \ell, M_{curr})$, transition: $t$
 1: iterations ← 0
 2: max_iterations ← 10
 3: $\pi_{final} \leftarrow \epsilon$
 4: repeat
 5:     $\delta \leftarrow \{p \in \bullet t \mid M_{curr}(p) = 0\}$          // Set of places in $\bullet t$ that are missing tokens
 6:     $\lambda \leftarrow \{p \in P \mid M_{curr}(p) > 0 \land p \notin \bullet t\}$          // Set of places with a surplus of tokens
 7:     firing_sequences ← all sequences of transitions that are on some shortest path, going through $\tau$-
        transitions only, between some place in $\lambda$ to some place in $\delta$
 8:     if exists $\pi \in$ firing_sequences s.t. $M_{curr}[\pi\rangle M_{new}$ then
 9:         $M_{curr} \leftarrow M_{new}$
10:         $\pi_{final} \leftarrow \pi_{final} \circ \pi$
11:     else
12:         return $\epsilon$          // No firing sequence was identified
13:     end if
14:     iterations ← iterations $+1$
15:     if iterations $>$ max_iterations then
16:         return $\epsilon$
17:     end if
18: until $\delta = \emptyset$
19: return $\pi_{final}$
```



(a) Speed-up of fitness in C++ when compared to PM4PY implementation in Python. Each point shows the speed-up on a single event log.

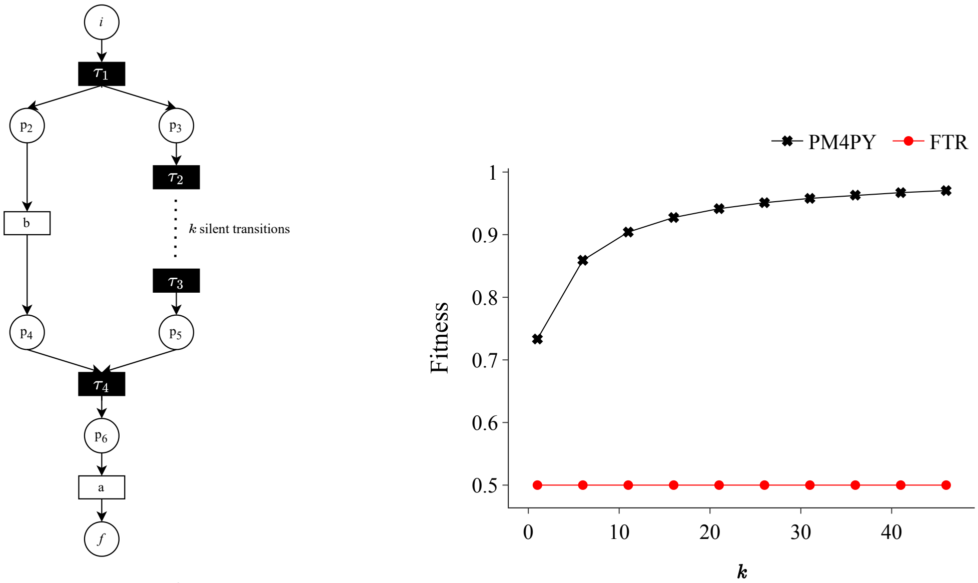
(b) Speed-up of precision in C++ when compared to PM4PY implementation in Python. Each point shows the speed-up on a single event log.

Fig. 3

Algorithm 2 outlines the algorithmic procedure for calculating precision for a given Petri net and event log. Similarly to fitness, it relies on token-based replay and silent transition handling as described in Algorithm 3. At each event in a trace, the algorithm determines the number of allowed tasks and escaping edges. As defined in Definition 13, the number of allowed tasks corresponds to the number of enabled transitions in the current marking. Additionally, a prefix map (Definition 12) is precomputed over the event log, and when given some trace prefix, it returns the set of activities that directly follow this prefix in at least one trace in the event log. The difference between the set returned by the prefix map and the set of allowed tasks yields the escaping edges that, together, are used to compute precision. The algorithm terminates once all traces have been replayed and returns the resulting precision score.

We test the speed-up of fitness and precision on 13 different event logs. Figure 3a shows the speed-up on fitness compared to PM4PY, revealing that we are at least one order of magnitude faster on all event logs. We achieve even higher speed-ups in precision, as shown in Figure 3b. Importantly, the speed-up scales with the size of the event logs. For example, on the biggest event log, which has more than 15.000 unique traces, we achieve a speed-up of more than 120x.

Lastly, we argue that a specific detail in PM4PY's implementation of fitness may lead to an unintended inflation in fitness. We illustrate this detail with the Petri net shown in Figure 4a with $k = 0$ and by replaying the trace a. Initially, both PM4PY and our algorithms obtain the firing sequence $\tau_1\tau_2\tau_3$ and marking $M = (p_2 : 1), (p_5 : 1)$ before having to insert a missing token. After reaching M, no more silent transitions can be fired due to the missing token in p_4, and a token is inserted into p_6 such that a is enabled and subsequently fired. The discrepancy occurs as PM4PY proceeds from the marking M and thus includes the produced and consumed tokens connected with firing $\tau_1\tau_2\tau_3$. We argue that the marking and token counts should be reverted, in this case to the initial marking, after failing to enable a via silent transitions. Consequently, PM4PY returns a fitness of 0.73 based on the token counts $p = 5$, $c = 5$, $m = 1$, $r = 1$, whereas we return 0.5 based on the token counts $c = 2$, $p = 2$, $m = 1$, $r = 1$, demonstrating the inflated consumed and produced token counts. The implication of this detail is shown in Figure 4b, where it is clear that fitness increases as k silent transitions are added to the WF-net, resulting in an erroneously high fitness. In contrast, we return consistent results regardless of k.



(a) WF-net with k silent transitions used for illustrating fitness with trace a.

(b) Growth of fitness as a function of k silent transitions in the WF-net (Figure 4a) when replaying trace a.

Fig. 4

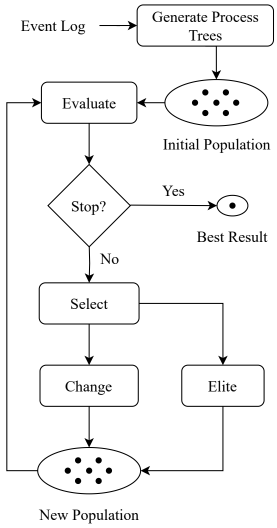
4 Genetic Tree Miner

In this section, we describe the core structure of our genetic algorithm for process discovery. Inspired by genetic algorithms, our method evolves a population of candidate solutions through selection, crossover, and mutation. We propose enhancements to the generation, objective function evaluation, and alter steps to improve both efficiency and model quality. An overview of our algorithm is provided in Figure 5 and detailed in Algorithm 4.

4.1 General Approach

In our approach, illustrated in Figure 5, we adopt the overall structure of genetically motivated algorithms [2]: from the input event log, we produce an initial population of process trees on which we evaluate a given objective function, and until we reach a stagnation criterion or a timeout, we select a number of the best elite individuals and modify the remaining ones in order to form a new population and repeat the whole process. We introduce several modifications to its internal mechanisms, including redesigning the building blocks for generating process trees, objective function evaluation, selection, and change. Each block plays a crucial role in the algorithm's ability to discover an acceptable process model and its overall run time. Hence, the purpose of our modifications is to improve on these parameters.

Algorithm 4 depicts the details of our genetic algorithm. It repeatedly computes a new generation by creating a new population of candidates from the previous generation. Based on a given objective function, we first select a percentage of the best-performing individuals as an elite set, which we move directly into the new population. A new population also includes some new randomly generated trees. The remaining candidates are created by a *tournament selection* [24] on the recently



Algorithm 4 Genetic Tree Miner

```
$\mathcal{P}_1 \leftarrow$ generate POPULATION_SIZE individuals
elite_count $\leftarrow$ POPULATION_SIZE $\times$ ELITE_RATE
random_count $\leftarrow$ POPULATION_SIZE $\times$ RANDOM_CREATION_RATE
tournament_count $\leftarrow$ POPULATION_SIZE $\times$ TOURNAMENT_RATE
tournament_size $\leftarrow$ POPULATION_SIZE $\times$ TOURNAMENT_SIZE_RATE
for $i = 1$ to NUM_GENERATIONS do
    survivors $\leftarrow$ elite_count best performing individuals in $\mathcal{P}_i$
    new_individuals $\leftarrow$ generate random_count individuals
    $\mathcal{P}_{i+1} \leftarrow$ survivors $\cup$ new_individuals
    for $j = 1$ to tournament_count do
        tournament $\leftarrow$ randomly select tournament_size individuals from $\mathcal{P}_i$
        parent1, parent2 $\leftarrow$ two best individuals from tournament
        child $\leftarrow$ do crossover with parent1 and parent2
        child $\leftarrow$ mutate child with MUTATION_PROBABILITY
        $\mathcal{P}_{i+1} \leftarrow \mathcal{P}_{i+1} \cup \{child\}$
    end for
end for
return best individual in $\mathcal{P}_{\texttt{NUM\_GENERATIONS}}$
```

Fig. 5: General overview and pseudocode of the GTM algorithm.

evaluated population: We randomly select a number of individuals for a tournament, and pick the two best individuals from the tournament based on the objective function and perform a crossover (see Section 4.3) followed by a random mutation (see Section 4.4). The child is then added to the new population, and the process is repeated until we reach the number of individuals required for the full population.

4.2 Initial Population

To generate an initial population of process trees, we create a subset of traces from the event log by randomly selecting a small percentage (initial sampling rate) of the traces in the event log. We treat the subset of traces as an event log and use it to discover a process tree by the IM. We guarantee the coverability of all activities in each subset by including an additional number of traces until all activities are present in the subset. Repeating this process several times, we obtain a population of process trees discovered on different aspects of the event log. In this way, we preserve diversity in the initial population as well as a fast method to create the initial population.

4.3 Crossover

A crossover involves two process trees identified using the tournament selection. We randomly select a subtree in each of the process trees and swap these subtrees, leading to two new children as depicted in Figure 6. For each child, we remove duplicate activities and randomly insert missing activities. Then we check if the children are valid according to the process tree definition. If the first child or second child is a valid process tree, this is returned (left child has a priority). If none of the children are a valid process tree, we randomly return one of the parents.

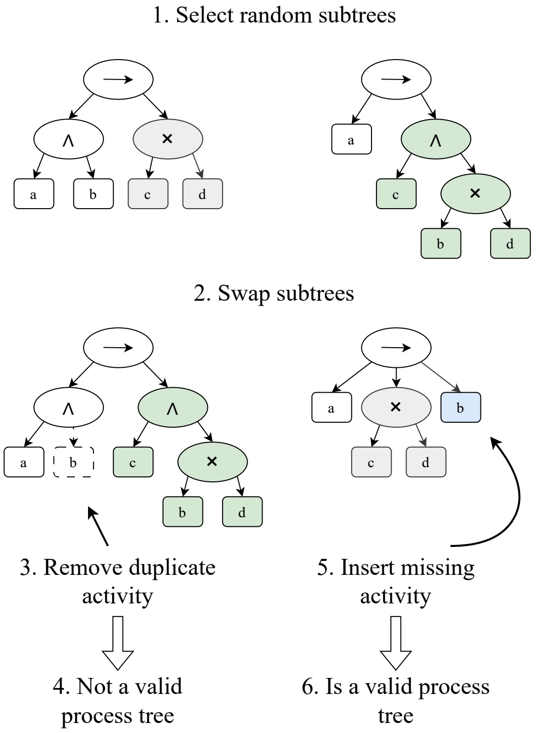

1. Select random subtrees
→
∧
×
a
b
c
d
→
a
∧
c
×
b
d
2. Swap subtrees
→
∧
∧
a
b
c
×
b
d
→
a
×
b
c
d
3. Remove duplicate activity
5. Insert missing activity
4. Not a valid process tree
6. Is a valid process tree

Fig. 6: Crossover example showing subtree exchange between two parent process trees. Grey and green indicate the selected subtrees; dashed lines show removed elements, and blue indicates inserted elements.

4.4 Mutations

To apply a mutation operation to a process tree, we randomly perform one of the following four mutation strategies as depicted in Figure 7, showing how the possible mutations transform the process tree of the valid child from the crossover example in Figure 6. Leaf replacement repositions one of the randomly selected activity nodes to a different position in the process tree. Operator swap randomly selects an operator node in the process tree and changes it to another randomly chosen operator. Subtree removal selects a random subtree in the process tree and removes it; then we create a new random tree with the activities from the removed subtree and insert it at a random point of the process tree. Loop addition selects a random activity node and changes it to a simple loop operator, adding the activity node and a τ activity node to the loop operator.

4.5 Fitness Estimation

In the objective function evaluation step, we measure the fitness, precision, refined simplicity, and simplicity of each candidate process tree with an objective function, which is a weighted average of these measures. The computation of fitness and precision is a time-consuming step in the algorithm, so minimizing the computation time is a major focus in our implementation. To speed up the computation of the objective function for every individual, we utilize a subset of the event log. How large a subset depends on the number of unique traces and is dictated by $f(t) = 0.5987 \cdot e^{-2.251\times10^{-4}\cdot t}$, where t is the number of unique traces. This ensures that on the smaller logs we consider about

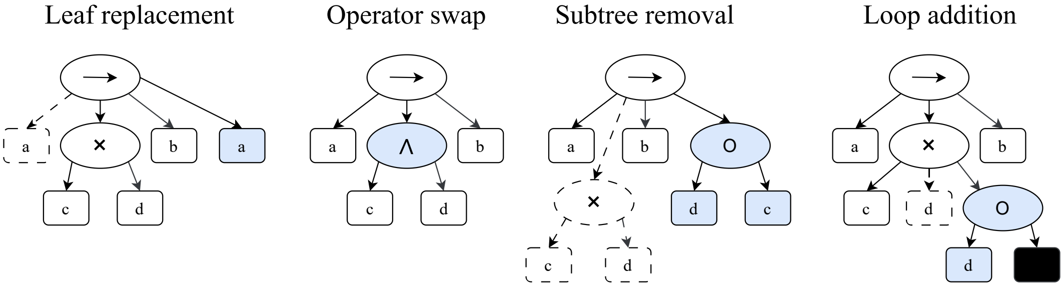


Fig. 7: Examples of mutation operations on the process tree resulting from Figure 6. Dashed lines indicate removed elements, and blue indicates inserted elements.

60% of unique traces, and on the larger logs we consider only about 5%. The function is shown in Figure 8, where the datasets that will be used later for benchmarking GTM are annotated. To ensure the coverability of all activities in the subset, we include an additional number of traces until all activities are present in the subset by selecting the most frequent traces containing the missing activities.

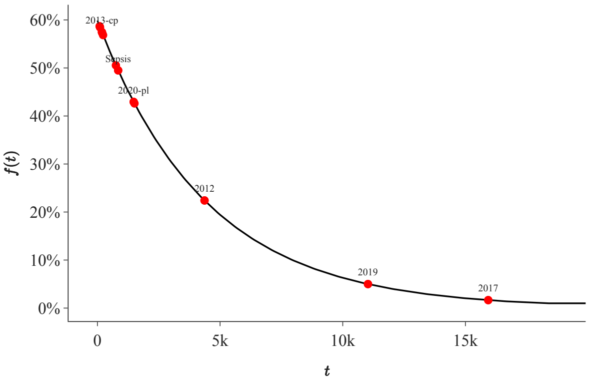


Fig. 8: The function $f(t) = 0.5987 \cdot e^{-2.251\times10^{-4}\cdot t}$ which is used to calculate the filtering of the event log. Each point represents an event log and its filtering percentage.

4.6 Stopping Criteria

To ensure that the search process remains computationally feasible and does not run indefinitely, we define a set of stopping criteria that are checked after each generation. The most critical of these is a maximum allowed duration. When the predefined time limit is exceeded, the search is terminated regardless of the current population state. This criterion ensures timely termination. In addition to the time limit, we incorporate a stagnation-based criterion, which halts the optimization if the objective score does not improve beyond a small threshold $\epsilon = 0.01$ for a given number of consecutive generations. These two criteria, duration and stagnation, are combined to ensure that the search is both effective and efficient, adapting dynamically to the progress of the optimization process.

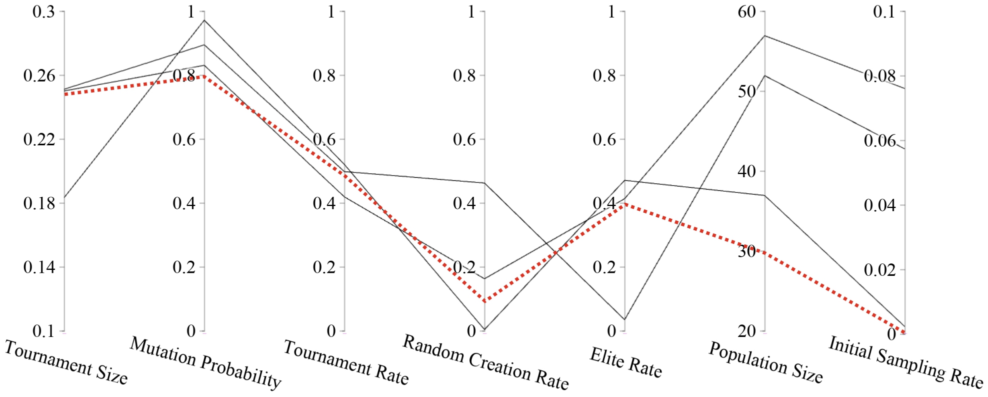


Fig. 9: Bayesian Optimization results on three event logs: 2013-cp, 2012, and Sepsis. The red dotted line highlights the proposed values for each hyperparameter.

4.7 Hyperparameters

Identifying suitable hyperparameters of genetic algorithms is a non-trivial task and their values play a crucial role for the performance of GTM; hence, we propose a set of default hyperparameter values for the user by running Bayesian Optimization (BO), as implemented in the Optuna library [25], on three event logs from the BPI Challenge [26]: 2013-cp [27], 2012 [28] and Sepsis [29]. BO optimizes the objective function, which is a weighted sum of four quality metrics: fitness (weight 0.5), precision (weight 0.3), simplicity (weight 0.1), and refined simplicity (weight 0.1), defined as $max\{0, 1 - \frac{\#places}{100}\}$. The results are depicted in Figure 9, and each black line represents the set of values that achieved the maximal objective score after 18 hours of running BO. Note that each axis displays the possible ranges in which the BO algorithm could search. A general tendency forms across the three event logs, indicating that a fixed set of values generalizes across all event logs. Based on the results, we propose the hyperparameter values highlighted with the red dotted line: 0.25 (tournament size rate), 0.8 (mutation probability), 0.5 (tournament rate), 0.1 (random creation rate), 0.4 (elite rate), 30 (population size), and 0.001 (initial sampling rate). We choose a lower population size and a reduced initial sampling rate than those suggested by the BO algorithm to speed up computation and to ensure rich diversity in the initial population, respectively, which has been shown to be beneficial [17].

5 Experimental Setup and Benchmarking

This section presents the experimental setup used to benchmark our proposed discovery method. We describe the event logs for benchmarking, quality measures, and baseline methods used for comparison. Furthermore, we outline implementation details and configurations of GTM to ensure reproducibility. Finally, we present the results of the performance benchmark.

5.1 Experimental Setup

To evaluate the performance and robustness of the proposed discovery method, we conduct our experiments on 13 diverse real-life event logs [26] commonly used for benchmarking in process

Event Log	Unique Traces	Traces	Average Trace Length	Unique Activities
2020-rfp [30]	89	6.886	5	19
2020-dd [31]	99	10.500	5	17
2013-op [32]	108	819	2	3
2013-cp [27]	183	1.487	4	4
2020-ptc [33]	202	2.099	8	29
RTF [34]	231	150.370	3	11
2020-id [35]	753	6.449	11	34
Sepsis [29]	846	1.050	14	16
2020-pl [36]	1.478	7.065	12	51
2013-i [37]	1.511	7.554	8	4
2012 [28]	4.366	13.087	20	23
2019 [38]	11.028	226.561	6	42
2017 [39]	15.930	31.509	38	26

Table 1: Overview of event log benchmark

mining research. No pre-processing or filtering was applied to the event logs except for 2019. The log is filtered by 10% due to size limitations in the commercial tool Apromore, which provides the implementation of SM. Additional information about the event logs is available in Table 1. We compare the performance of the process discovery algorithms using the variants of fitness, precision, generalization, and simplicity presented in Section 2 and as implemented in PM4PY.

To evaluate the effectiveness of GTM, we compare it against two widely-used algorithms: IM [3] and SM [4]. The IM is implemented in PM4PY, and we use the default configuration provided in the library. Regarding SM, we employ the commercially available implementation of SM in Apromore[1] with its default parameters. These two methods are chosen due to their strong performance and frequent use in both academic and commercial settings.

All experiments are conducted on a laptop equipped with an Apple M2 3.49 GHz, 16 GB of RAM, and running macOS Sequoia 15.4.1 using Python 3.13 and PM4PY 2.7.15.2. No multiprocessing is utilized, and GTM and IM are run on the same hardware. SM is run on Apromore servers, and no timing information is provided; however, other studies show that process discovery using SM is several times faster than using IM [4].

GTM is run with a stagnation limit of 50 generations with less than 1% improvement. We adopt the naming convention GTM-X, where X denotes the maximum allowed runtime in seconds. The objective function is composed of C++ fitness (0.5 weight), C++ precision (0.3 weight), PM4PY simplicity (0.1 weight), and our own simplicity measure (weight 0.1). To account for stochasticity in GTM, each GTM-X configuration is executed five times, and the median values across these runs are reported. Lastly, we provide all relevant source code, datasets, and instructions for reproducing the results in a public GitHub repository [40].

5.2 Results

Table 3 presents a comprehensive benchmark of GTM, IM, and SM on all event logs, highlighting the strengths and trade-offs of each approach. The aggregated results are shown in Table 2. GTM-10

---

[1] http://apromore.org/

Discovery Method	**Accuracy**			**Generalization**	**Simplicity**	**Objective Score**	**Time (s)**
	F1-score	Fitness	Precision				
GTM-10	0.78	0.96	0.73	**0.94**	0.69	0.84	8.62
GTM-60	0.91	0.95	0.90	0.92	0.70	0.89	35.34
GTM-300	**0.97**	0.97	**0.98**	0.92	0.68	**0.93**	100.05
IM	0.51	**0.99**	0.39	0.90	0.62	0.66	7.44
SM	0.80	0.70	0.94	0.91	**0.78**	0.80	-

Table 2: Aggregated results of reported values in Table 3.

results are reported for all event logs, GTM-60 results are reported for those event logs where GTM-10 fails to converge within its time limit, and, in turn, GTM-300 results are reported for event logs where GTM-60 fails to converge within its time limit.

All GTM configurations consistently outperform IM and SM on F1-score. Generally, we see that IM exhibits high fitness but tends to yield low precision. In contrast, SM generally achieves near-perfect precision but with reduced fitness. GTM instead provides a balanced trade-off between precision and fitness as evidenced by its superior F1-scores. GTM-10 beats IM on F1-score on all logs with only a slight increase in average runtime of 1.18 seconds. Although GTM-10 does fall behind SM on some of the larger and more complex logs, this deficit is quickly eliminated when GTM is given a higher time limit. Specifically, GTM-10 performs better than IM and SM on 8 out of 13 logs, GTM-60 on 12 out of 13 logs for and GTM-300 is better on all 13 logs.

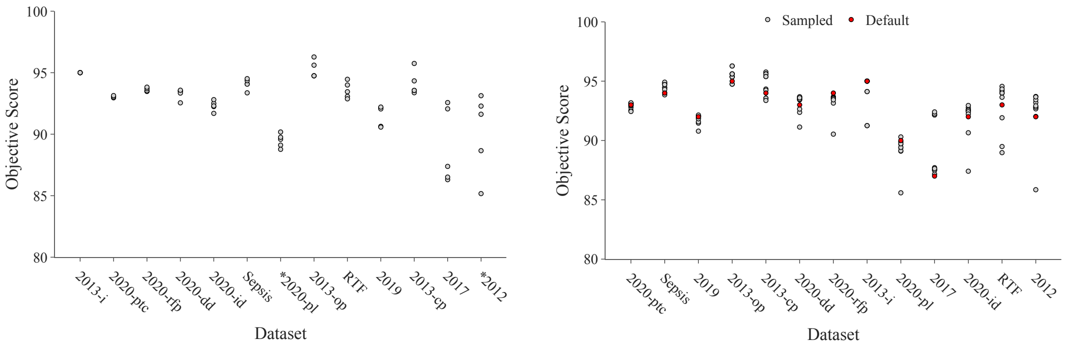
With respect to generalization, all methods exhibit comparable performance, as reflected by their average scores, which differ by no more than 0.04. In the simplicity dimension, SM performs very well demonstrated by its simplicity being the highest on 12 out of 13 event logs. This is also reflected in the average simplicity in Table 2, where SM is noticeably better than the two other methods, however, often at the expense of lower fitness compared to both GTM and IM.

In summary, GTM-10 outperforms IM and competes with SM on most logs. GTM-60 achieves the most balanced performance at the expense of longer (but acceptable) running times compared to IM and SM. GTM-300 performs significantly better on the F1-score on the largest logs compared to any of the other miners. In addition, GTM is more than an order of magnitude faster than the previous approaches based on genetic algorithms [4].

Log Name	Discovery Method	Accuracy			Generalization	Simplicity	Objective Score	Time (s)
		F1-score	Fitness	Precision				
2013-op	GTM-10	**0.96**	0.95	0.97	**0.96**	0.82	**0.95**	1.15
	IM	0.95	**1.00**	0.91	0.93	0.69	0.83	0.01
	SM	0.82	0.70	**0.99**	**0.96**	**1.00**	0.85	-
2020-dd	GTM-10	**0.97**	0.97	**0.97**	**0.89**	**0.69**	**0.92**	8.68
	IM	0.56	**1.00**	0.39	0.85	0.60	0.75	1.21
	SM	0.91	0.92	0.89	0.82	0.67	0.88	-
RTF	GTM-10	**0.97**	0.95	**1.00**	**0.99**	0.80	**0.94**	9.86
	IM	0.77	**1.00**	0.63	0.97	0.62	0.63	0.63
	SM	0.88	0.79	0.99	**0.99**	**0.92**	0.87	-
2020-rfp	GTM-10	**0.99**	0.98	**1.00**	0.87	0.60	0.93	10.0
	GTM-60	**0.99**	0.99	**1.00**	**0.89**	0.62	**0.94**	10.17
	IM	0.50	**1.00**	0.33	0.87	0.60	0.73	0.07
	SM	0.82	0.83	0.82	0.81	**0.66**	0.81	-
2020-ptc	GTM-10	**0.99**	0.98	**1.00**	**0.93**	0.65	0.91	10.01
	GTM-60	**0.99**	0.98	**1.00**	**0.93**	0.64	**0.92**	39.52
	IM	0.27	**0.99**	0.16	0.89	0.59	0.59	0.29
	SM	0.76	0.64	0.96	0.88	**0.70**	0.76	-
2013-i	GTM-10	**0.97**	0.97	0.98	**0.96**	0.79	**0.95**	10.02
	GTM-60	**0.97**	0.97	0.98	**0.96**	0.79	**0.95**	22.24
	IM	0.77	**1.00**	0.63	0.87	0.67	0.79	0.13
	SM	0.87	0.77	**1.00**	0.92	**0.85**	0.84	-
2020-id	GTM-10	0.54	0.96	0.38	0.90	0.62	0.71	10.03
	GTM-60	0.91	0.92	0.90	0.91	0.65	0.86	60.01
	GTM-300	**0.99**	**0.98**	**1.00**	**0.92**	0.62	**0.92**	160.02
	IM	0.38	0.97	0.23	0.88	0.62	0.51	0.49
	SM	0.85	0.76	0.97	0.90	**0.66**	0.82	-
2020-pl	GTM-10	0.31	0.97	0.19	**0.91**	0.60	0.65	10.12
	GTM-60	0.92	0.86	0.99	0.88	0.59	0.80	60.09
	GTM-300	**0.98**	0.97	**1.00**	0.87	0.54	**0.90**	300.09
	IM	0.18	**1.00**	0.10	0.86	0.59	0.52	4.6
	SM	0.81	0.70	0.96	0.86	**0.65**	0.77	-
2019	GTM-10	0.49	0.97	0.32	**0.95**	0.61	0.66	10.07
	GTM-60	0.55	0.97	0.38	0.94	0.59	0.72	60.02
	GTM-300	**0.99**	0.99	**1.00**	0.94	0.54	**0.92**	271.74
	IM	0.38	**1.00**	0.23	0.92	0.59	0.56	36.17
	SM	0.68	0.51	**1.00**	0.91	**0.73**	0.70	-
2017	GTM-10	0.50	0.95	0.34	0.98	0.66	0.77	10.03
	GTM-60	**0.92**	0.94	**0.90**	0.95	0.66	**0.88**	60.02
	GTM-300	0.89	0.99	0.80	**0.99**	0.65	0.87	106.66
	IM	0.26	**1.00**	0.15	0.95	0.63	0.63	42.99
	SM	0.76	0.71	0.80	0.95	**0.73**	0.75	-
2013-cp	GTM-10	**0.95**	0.91	**1.00**	**0.93**	0.85	**0.94**	2.2
	IM	0.89	**1.00**	0.79	0.88	0.66	0.86	0.02
	SM	0.76	0.62	0.99	0.92	**1.00**	0.81	-
Sepsis	GTM-10	0.97	0.97	0.97	0.92	0.65	0.92	10.01
	GTM-60	**0.99**	0.98	**0.99**	**0.95**	0.71	**0.94**	53.12
	IM	0.49	**1.00**	0.32	0.90	0.62	0.66	0.22
	SM	0.81	0.69	0.98	0.92	**0.79**	0.81	-
2012	GTM-10	0.55	0.95	0.39	**0.98**	0.67	0.74	10.0
	GTM-60	0.78	0.94	0.67	0.90	0.66	0.84	60.05
	GTM-300	**0.96**	0.93	**1.00**	**0.98**	0.77	**0.92**	300.07
	IM	0.24	**0.97**	0.14	0.95	0.61	0.56	9.93
	SM	0.64	0.49	0.92	**0.98**	**0.82**	0.68	-

Table 3: Performance comparison of process discovery methods. All process discovery methods use default parameters (and only the last three logs in the table were used to identify GTM's default parameters).

5.3 Convergence and Stability



(a) Objective score for five distinct runs with GTM-300; the stars mark the event logs where stagnation was not achieved within the 300-second time limit.

(b) Objective score for 10 distinct runs with GTM-300 using varying hyperparameters. The red solid points denote GTM-300 with default values.

Fig. 10

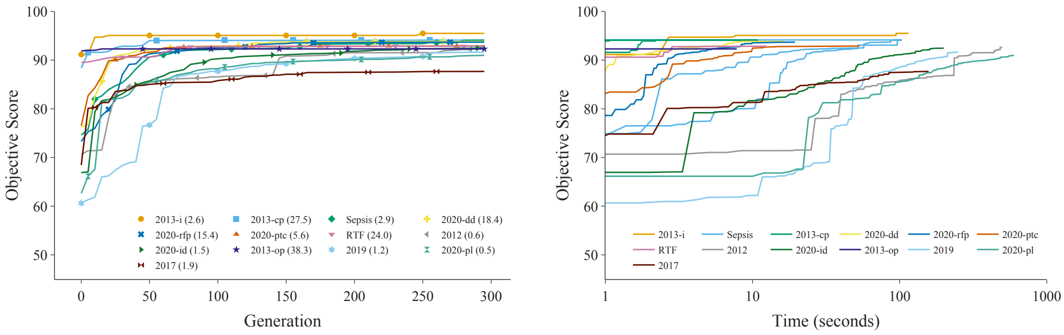
GTM is inherently stochastic due to randomness in tournament selection, crossover, and mutation, and may yield different results when run repeatedly. We therefore evaluate its consistency by running GTM-300 five times on each event log. The results are observed in Figure 10a. Each point shows the objective score of a distinct run, and the x-axis is sorted in ascending order by the size of the objective scores' interquartile range. On 11 out of 13 event logs, the variance in objective score is negligible. A higher variance can be observed on two of the three largest event logs, indicating some correlation between the log size and variance. Hence, the stochastic properties of GTM remain a potential weakness compared to deterministic algorithms such as IM and SM, which yield consistent results when run repeatedly.

In Figure 10b, we extend the robustness evaluation by examining GTM's sensitivity to hyperparameter changes. Specifically, we vary the default hyperparameters by up to 10%, adjust the population size by up to 5, and subsequently run GTM-300 10 times on the sampled configurations on each log. Ideally, GTM should be robust to such variations and provide stable performance as long as reasonable values are chosen. Overall, we see stable performance with few fluctuations and instabilities. On some logs, performance even improves as a result of the changes, suggesting that fine-tuning hyperparameters for specific logs could yield better results.

A potential explanation for the correlation between log size and variance in objective scores is provided in Figure 11a. On smaller event logs, GTM converges within the first 50 generations and quickly reaches the stagnation limit, resulting in consistent and stable performance. In contrast, on larger logs, the objective score improves continuously up to the 300th generation. Observe also Figure 11b showing that GTM does not converge within 300 seconds on the largest logs and continues to improve after the time limit. These results suggest that increasing the timit could help GTM achieve a more stable performance on larger event logs.

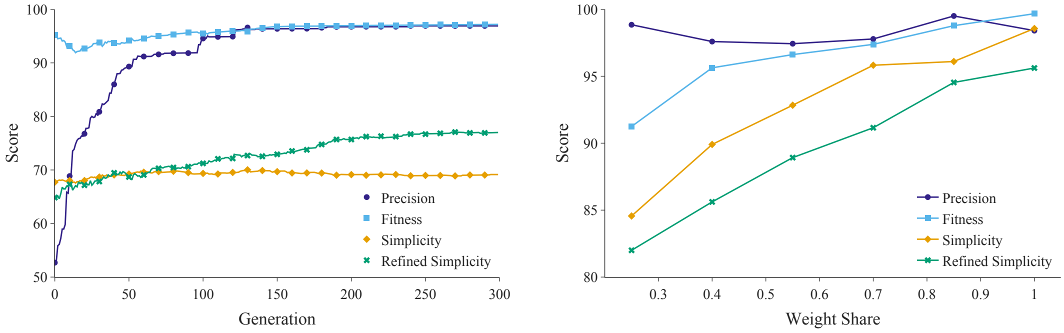
5.4 Objective Function Sensitivity Analysis and Decomposition

A key advantage of GTM over established process discovery algorithms such as IM and SM is the possibility to tailor the objective function to specific process discovery quality criteria. It remains



(a) Convergence of GTM on all event logs. The parentheses for each log specify the number of generations per second.

(b) Convergence behavior of GTM across all event logs as a function of runtime.



(c) Decomposition of the objective function over 300 generations. Each line is the mean of the respective metric scores across all event logs.

(d) Mean score across all event logs for the respective metrics. A weight share is the assigned weight for a specific metric while the other metrics are assigned to an equal share of the remaining weight (weights sum to 1).

Fig. 11

to be shown that GTM actually produces outputs accordingly to changes in the objective function. Figure 11d demonstrates the development of each metric when given increasingly larger weight shares. Starting from a weight of 25% up to 100%, there is a clear tendency for quality metrics to improve as their weights in the objective function increase, with precision being the only exception, as it remains high regardless of its weight, indicating an inherent bias towards process models with high precision. Thus, GTM successfully optimizes fitness, refined simplicity, and simplicity.

We further decompose the objective function in Figure 11c to better understand the underlying mechanisms driving GTM's overall performance during the evolutionary process. The lines are the mean over all event logs and show the (unweighted) score of a metric for the best tree in each generation. Notably, GTM initializes with almost perfect fitness, likely due to IM trees in the initial population. As a side effect of the IM trees, precision is initially low but quickly improves within the first 100 generations. While refined simplicity slowly improves, simplicity remains almost constant throughout the generations, probably due to its low weight.

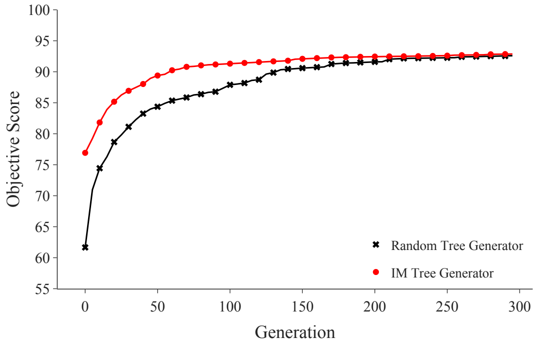


Fig. 12: Convergence of GTM Initial Population vs. Random Initial Population. A line represents the mean objective score across all event logs.

5.5 Impact of Initial Population

One could dispute the actual effectiveness of GTM and argue that most of its high performance originates from using IM-produced process trees in the initial population. We disprove this by comparing the convergence rate using randomly generated trees in the initial population to the default initial population method. The advantage of the default method is evident by Figure 12; GTM achieves better objective scores in the early generations, which we just saw is particularly useful for larger logs due to the slower convergence rates, yet it also demonstrates that we propose well-performing crossover and mutation changes that guarantee convergence even without injecting IM generated process trees into the initial generation.

6 Conclusion

We presented GTM, a novel and sound process discovery method that leverages evolutionary search to construct high-quality process models. Our method is designed to be highly flexible, allowing it to adapt to a wide range of process structures, and produces sound models by design. Compared to previous genetic approaches such as ETM, GTM achieves superior performance with significantly reduced computation times—over an order of magnitude faster—making it practically viable for real-world usage. GTM advances genetic process discovery through a highly efficient C++-based fitness implementation, a novel self-loop mutation operator, and a refined objective function that better penalizes model complexity.

The results show that GTM-60 achieves the highest F1-score on the majority of logs under constrained runtime settings. It consistently outperforms both IM and SM in terms of overall model accuracy while maintaining competitive simplicity and generalization. This indicates that GTM is well-suited for diverse process mining scenarios where both performance and interpretability are critical.

To foster reproducibility and contribute to the process mining community, we have open-sourced all source code, event logs, and evaluation scripts in a public GitHub repository [40]. Furthermore, we will make our contributions available to PM4PY by creating a pull request fixing the identified issues with fitness estimation.

For future work, investigating adaptive strategies for dynamically tuning hyperparameters based on characteristics of the input event log. Additionally, exploring online adjustment of these parameters during the discovery run to improve convergence speed and model quality. Overall, GTM represents a significant step forward in the design of evolutionary process discovery methods and provides a strong foundation for further research and application.

7 Acknowledgments

We want to express our sincere gratitude to Jiří Srba for his invaluable guidance, feedback, and support throughout the development of this thesis and the related paper currently under submission to a conference.

Parts of this text were refined at the sentence level using AI-based tools to improve clarity and style. All technical content, research contributions, and substantive ideas remain exclusively those of the authors, except where otherwise noted.

8 Bibliographical Remarks

Parts of this thesis build upon work carried out during a previous semester project [41]. Specifically, Figure 1, Figure 2, and the text in Section 2.4 (shorter and modified version) are taken from [41]. The following definitions—Definition 2, 3, 5, 7, 8, 9, and 10— together with other minor text modifications, were reviewed and refined by the supervisor during the preparation of the conference submission. The students drafted the first version of all the definitions and text. Similarly, Section 4 was shortened and streamlined in direct collaboration with the supervisor.

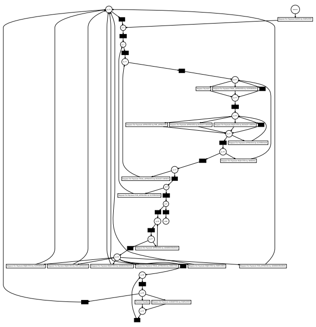
References

1. Wil van der Aalst. *Process Mining: Data Science in Action*. Springer Berlin Heidelberg, 2016. URL: `http://dx.doi.org/10.1007/978-3-662-49851-4`, `doi:10.1007/978-3-662-49851-4`.
2. Ana Karla A. de Medeiros, A. J. M. M. Weijters, and Wil M. P. van der Aalst. Genetic process mining: An experimental evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007. `doi:10.1007/s10618-006-0061-7`.
3. Sander Leemans, Dirk Fahland, and Wil Aalst. Process and deviation exploration with inductive visual miner. *CEUR Workshop Proceedings*, 1295:46, 01 2014.
4. Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Artem Polyvyanyy. Split miner: automated discovery of accurate and simple business process models from event logs. *Knowledge and Information Systems*, 59, 05 2019. `doi:10.1007/s10115-018-1214-x`.
5. Alessandro Berti and Wil M.P. van der Aalst. Reviving token-based replay: Increasing speed while improving diagnostics. In *ATAED@Petri Nets/ACSD*, 2019. URL: `https://api.semanticscholar.org/CorpusID:190004028`.
6. Jorge Muñoz-Gama and Josep Carmona. A fresh look at precision in process conformance. In Richard Hull, Jan Mendling, and Stefan Tai, editors, *Business Process Management. BPM 2010*, volume 6336 of *Lecture Notes in Computer Science*, pages 211–226. Springer, Berlin, Heidelberg, 2010. `doi:10.1007/978-3-642-15618-2_16`.
7. Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012. URL: `https://arxiv.org/abs/1206.2944`, `arXiv:1206.2944`.
8. W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004. `doi:10.1109/TKDE.2004.47`.
9. A. Weijters, Wil Aalst, and Alves Medeiros. Process mining with the heuristics miner-algorithm. *Cirp Annals-manufacturing Technology - CIRP ANN-MANUF TECHNOL*, 166, 01 2006.
10. J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik. Process discovery using integer linear programming. In Kees M. van Hee and Rüdiger Valk, editors, *Applications and Theory of Petri Nets*, pages 368–387, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
11. H. M. W. Verbeek and Wil M. P. van der Aalst. Decomposed process mining: The ilp case. In Fabiana Fournier and Jan Mendling, editors, *Business Process Management Workshops*, pages 264–276, Cham, 2015. Springer International Publishing.
12. Adriano Augusto, Marlon Dumas, and Marcello La Rosa. Automated discovery of process models with true concurrency and inclusive choices, 2021. URL: `https://arxiv.org/abs/2105.06016`, `arXiv:2105.06016`.
13. Wil Aalst, Ana Medeiros, and A. Weijters. Genetic process mining. volume 14, pages 48–69, 06 2005. `doi:10.1007/11494744_5`.
14. J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. A genetic algorithm for discovering process trees. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, 2012. `doi:10.1109/CEC.2012.6256458`.
15. M.L. Eck, van, J.C.A.M. Buijs, and B.F. Dongen, van. Genetic process mining: Alignment-based process model mutation. In F. Fournier and J. Mendling, editors, *Business Process Management Workshops (BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers)*, Lecture Notes in Business Information Processing, pages 291–303, Germany, 2015. Springer. `doi:10.1007/978-3-319-15895-2_25`.
16. Thomas Molka, David Redlich, Wasif Gilani, Xiao-Jun Zeng, and Marc Drobek. Evolutionary computation based discovery of hierarchical business process models. In Witold Abramowicz, editor, *Business Information Systems*, pages 191–204, Cham, 2015. Springer International Publishing.
17. Thomas Molka, David Redlich, Marc Drobek, Xiao-Jun Zeng, and Wasif Gilani. Diversity guided evolutionary mining of hierarchical process models. GECCO '15, page 1247–1254, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2739480.2754765`.
18. C. A. Petri. Kommunikation mit automaten. 1962. URL: `https://api.semanticscholar.org/CorpusID:117254333`.

19. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In José-Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency*, pages 311–329, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
20. Sebastiaan J. van Zelst. Translating workflow nets to process trees: An algorithmic approach. *CoRR*, abs/2004.08213, 2020. URL: https://arxiv.org/abs/2004.08213, arXiv:2004.08213.
21. Alessandro Berti, Sebastiaan van Zelst, and Daniel Schuster. PM4Py: A process mining library for Python. *Software Impacts*, 17:100556, 2023. URL: https://www.sciencedirect.com/science/article/pii/S2665963823000933, doi:10.1016/j.simpa.2023.100556.
22. Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M.P. van der Aalst. Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. *Int. J. Cooperative Inf. Syst.*, 23, 2014. URL: https://api.semanticscholar.org/CorpusID:17410482.
23. Borja Vázquez-Barreiros, Manuel Mucientes, and Manuel Lama. Prodigen: Mining complete, precise and minimal structure process models with a genetic algorithm. *Information Sciences*, 294:315–333, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0020025514009694, doi:10.1016/j.ins.2014.09.057.
24. Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.
25. Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.
26. Task Force on Process Mining. Event logs - task force on process mining, n.d. Accessed: 2025-05-22. URL: https://www.tf-pm.org/resources/logs.
27. Ward Steeman. Bpi challenge 2013, closed problems, 2013. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2013_closed_problems/12714476/1, doi:10.4121/uuid:c2c3b154-ab26-4b31-a0e8-8f2350ddac11.
28. Boudewijn van Dongen. Bpi challenge 2012, 2012. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2012/12689204/1, doi:10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f.
29. Felix Mannhardt. Sepsis cases - event log, 2016. URL: https://data.4tu.nl/articles/dataset/Sepsis_Cases_-_Event_Log/12707639/1, doi:10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.
30. Boudewijn van Dongen. Bpi challenge 2020: Request for payment, 2020. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Request_For_Payment/12706886/1, doi:10.4121/uuid:895b26fb-6f25-46eb-9e48-0dca26fcd030.
31. Boudewijn van Dongen. Bpi challenge 2020: Domestic declarations, 2020. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Domestic_Declarations/12692543/1, doi:10.4121/uuid:3f422315-ed9d-4882-891f-e180b5b4feb5.
32. Ward Steeman. Bpi challenge 2013, open problems, 2013. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2013_open_problems/12688556/1, doi:10.4121/uuid:3537c19d-6c64-4b1d-815d-915ab0e479da.
33. Boudewijn van Dongen. Bpi challenge 2020: Prepaid travel costs, 2020. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Prepaid_Travel_Costs/12696722/1, doi:10.4121/uuid:5d2fe5e1-f91f-4a3b-ad9b-9e4126870165.
34. M. (Massimiliano) de Leoni and Felix Mannhardt. Road traffic fine management process, 2015. URL: https://data.4tu.nl/articles/dataset/Road_Traffic_Fine_Management_Process/12683249/1, doi:10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.
35. Boudewijn van Dongen. Bpi challenge 2020: International declarations, 2020. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_International_Declarations/12687374/1, doi:10.4121/uuid:2bbf8f6a-fc50-48eb-aa9e-c4ea5ef7e8c5.
36. Boudewijn van Dongen. Bpi challenge 2020: Travel permit data, 2020. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2020_Travel_Permit_Data/12718178/1, doi:10.4121/uuid:ea03d361-a7cd-4f5e-83d8-5fbdf0362550.
37. Ward Steeman. Bpi challenge 2013, incidents, 2013. URL: https://data.4tu.nl/articles/dataset/BPI_Challenge_2013_incidents/12693914/1, doi:10.4121/uuid:500573e6-accc-4b0c-9576-aa5468b10cee.

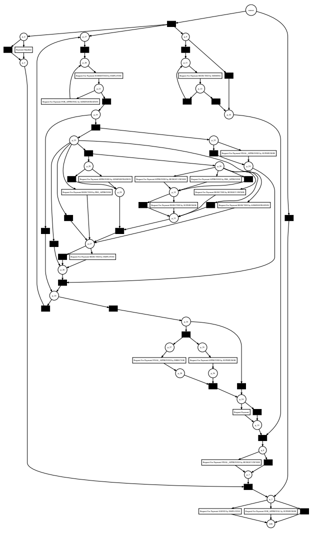
38. Boudewijn van Dongen. Bpi challenge 2019, 2019. URL: `https://data.4tu.nl/articles/dataset/BPI_Challenge_2019/12715853/1`, `doi:10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1`.
39. Boudewijn van Dongen. Bpi challenge 2017, 2017. URL: `https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884/1`, `doi:10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b`.
40. Genetic tree miner. `https://github.com/jaxels20/genetic-tree-miner-thesis`, 2025. Accessed: 2025-05-20.
41. Axelsen, Kowalski, and Nygård. Aau miner: Leveraging graph neural networks for process discovery. Semester project submitted at AAU, Department of Computer Science, January 2025.

A Discovered Petri Nets on 2020-rfp



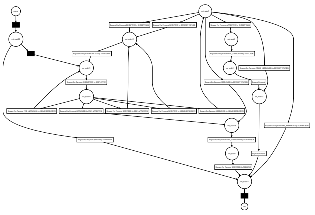
GTM-60:

$$\begin{aligned} F1 &= 0.99 \\ F &= 0.99 \\ P &= 1.00 \\ G &= 0.89 \\ S &= 0.62 \\ OF &= 0.94 \end{aligned}$$



IM:

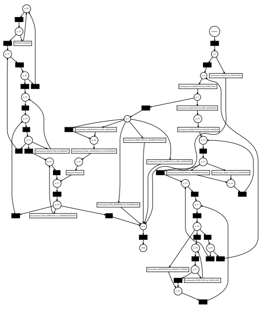
$$\begin{aligned} F1 &= 0.50 \\ F &= 1.00 \\ P &= 0.33 \\ G &= 0.87 \\ S &= 0.60 \\ OF &= 0.73 \end{aligned}$$



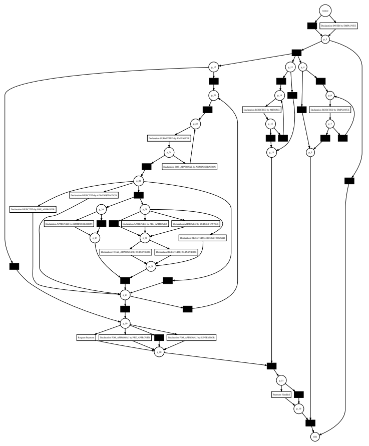
SM:

$$\begin{aligned} F1 &= 0.82 \\ F &= 0.83 \\ P &= 0.82 \\ G &= 0.81 \\ S &= 0.66 \\ OF &= 0.81 \end{aligned}$$

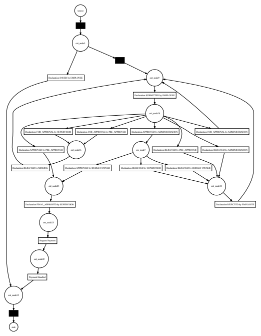
B Discovered Petri Nets - 2020-dd



GTM-10: $$\begin{aligned} F1 &= 0.97 \\ F &= 0.97 \\ P &= 0.97 \\ G &= 0.89 \\ S &= 0.69 \\ OF &= 0.92 \end{aligned}$$

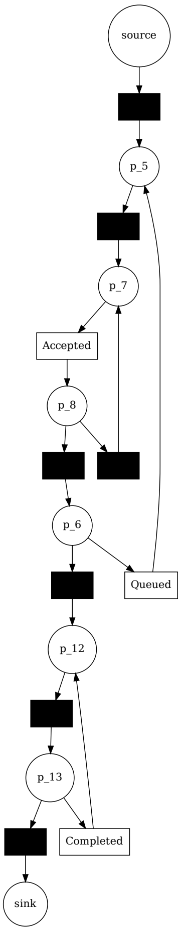


IM: $$\begin{aligned} F1 &= 0.56 \\ F &= 1.00 \\ P &= 0.39 \\ G &= 0.85 \\ S &= 0.60 \\ OF &= 0.75 \end{aligned}$$



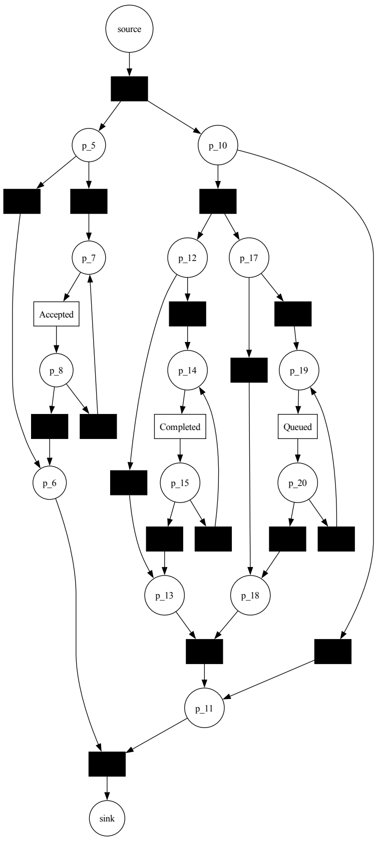
SM: $$\begin{aligned} F1 &= 0.91 \\ F &= 0.92 \\ P &= 0.89 \\ G &= 0.82 \\ S &= 0.67 \\ OF &= 0.88 \end{aligned}$$

C Discovered Petri Nets - 2013-op



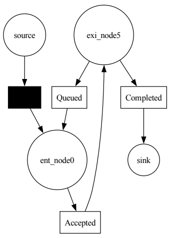
GTM-10:

$F1 = 0.96$
$F = 0.95$
$P = 0.97$
$G = 0.96$
$S = 0.82$
$OF = 0.95$



IM:

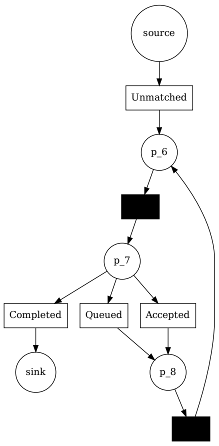
$F1 = 0.95$
$F = 1.00$
$P = 0.91$
$G = 0.93$
$S = 0.69$
$OF = 0.83$



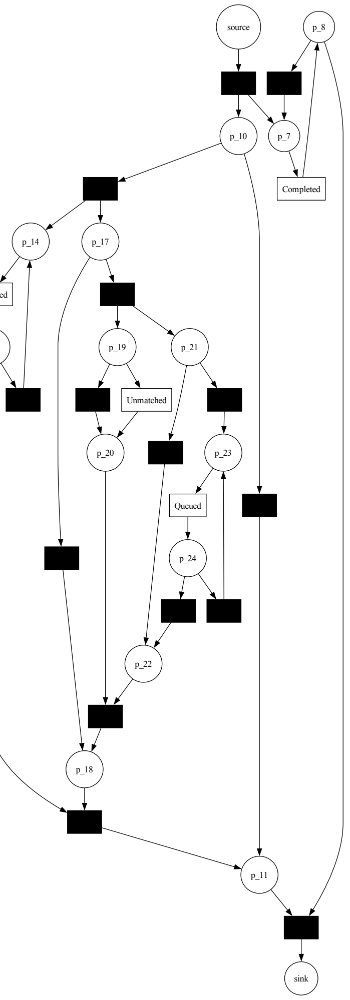
SM:

$F1 = 0.82$
$F = 0.70$
$P = 0.99$
$G = 0.96$
$S = 1.00$
$OF = 0.85$

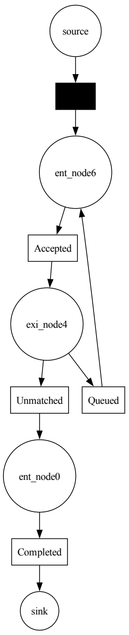
D Discovered Petri Nets - 2013-cp



GTM-10:
$F1 = 0.95$
$F = 0.91$
$P = 1.00$
$G = 0.93$
$S = 0.85$
$OF = 0.94$

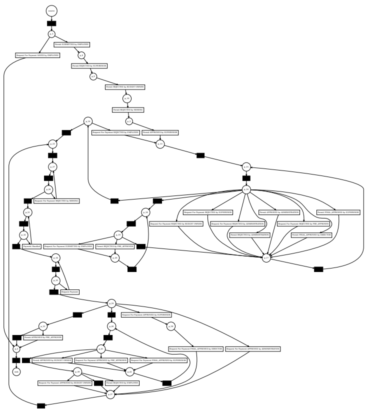


IM:
$F1 = 0.89$
$F = 1.00$
$P = 0.79$
$G = 0.88$
$S = 0.66$
$OF = 0.86$

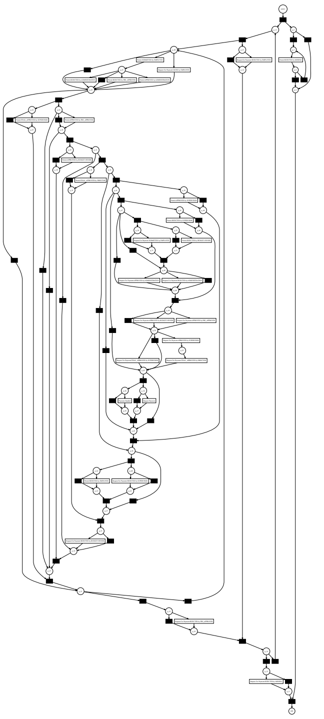


SM:
$F1 = 0.76$
$F = 0.62$
$P = 0.99$
$G = 0.92$
$S = 1.00$
$OF = 0.81$

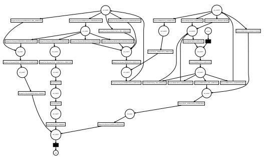
E Discovered Petri Nets - 2020-ptc



GTM-60:
$F1 = 0.99$
$F = 0.98$
$P = 1.00$
$G = 0.93$
$S = 0.64$
$OF = 0.92$

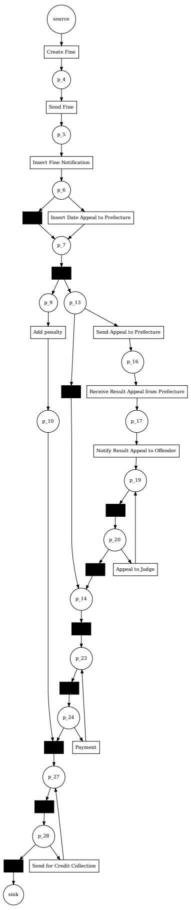


IM:
$F1 = 0.27$
$F = 0.99$
$P = 0.16$
$G = 0.89$
$S = 0.59$
$OF = 0.59$



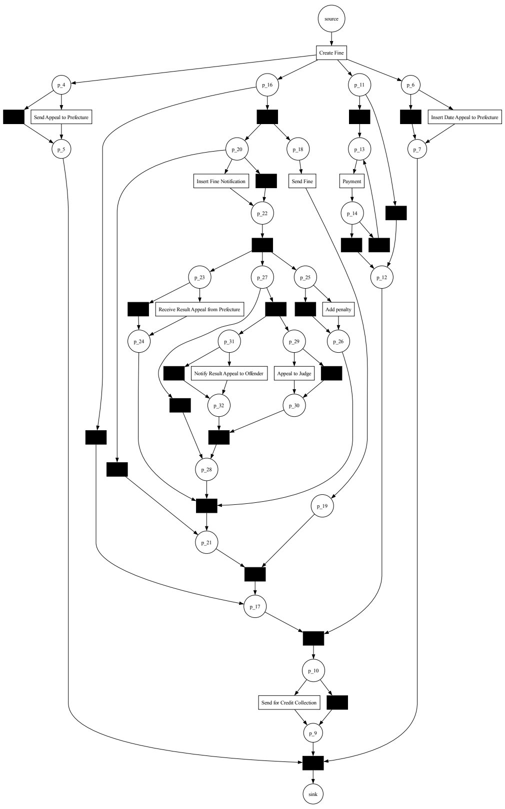
SM:
$F1 = 0.76$
$F = 0.64$
$P = 0.96$
$G = 0.88$
$S = 0.70$
$OF = 0.76$

F Discovered Petri Nets - RTF


source
Create Fine
p_4
Send Fine
p_5
Insert Fine Notification
p_6
Insert Date Appeal to Prefecture
p_7
p_9
p_13
Add penalty
Send Appeal to Prefecture
p_16
Receive Result Appeal from Prefecture
p_10
p_17
Notify Result Appeal to Offender
p_19
p_20
Appeal to Judge
p_14
p_23
p_24
Payment
p_27
p_28
Send for Credit Collection
sink

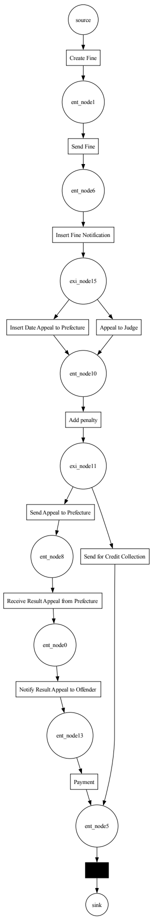
GTM-10:

$F1 = 0.97$
$F = 0.95$
$P = 1.00$
$G = 0.99$
$S = 0.80$
$OF = 0.94$



IM:

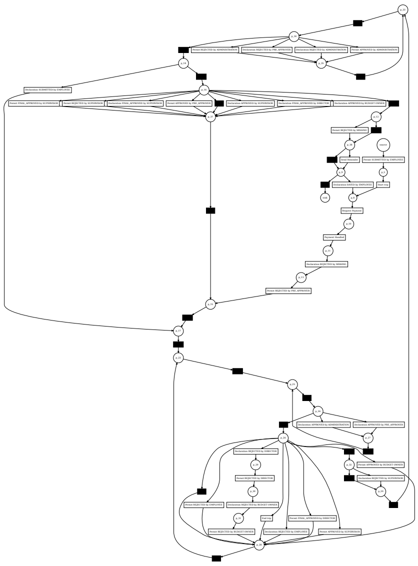
$F1 = 0.77$
$F = 1.00$
$P = 0.63$
$G = 0.97$
$S = 0.62$
$OF = 0.63$



SM:

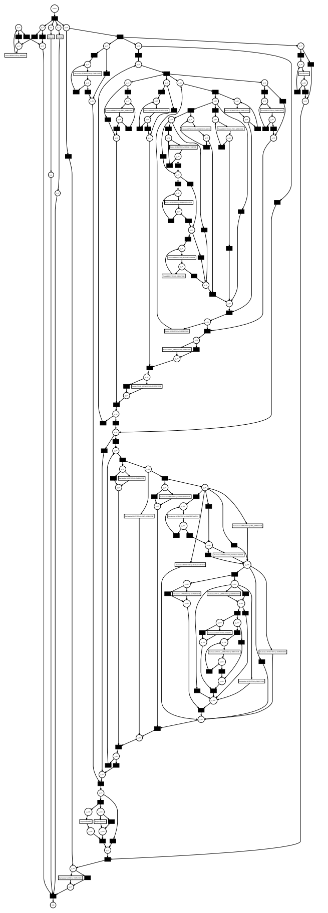
$F1 = 0.88$
$F = 0.79$
$P = 0.99$
$G = 0.99$
$S = 0.92$
$OF = 0.87$

G Discovered Petri Nets - 2020-id



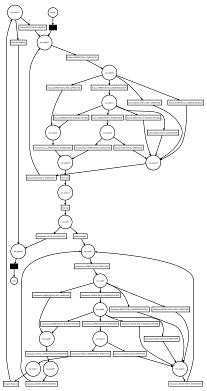
GTM-300:

$F1 = 0.99$
$F = 0.98$
$P = 1.00$
$G = 0.92$
$S = 0.62$
$OF = 0.92$



IM:

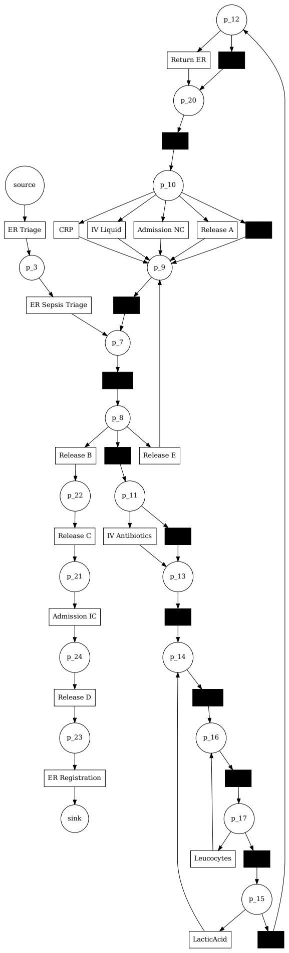
$F1 = 0.38$
$F = 0.97$
$P = 0.23$
$G = 0.88$
$S = 0.62$
$OF = 0.51$



SM:

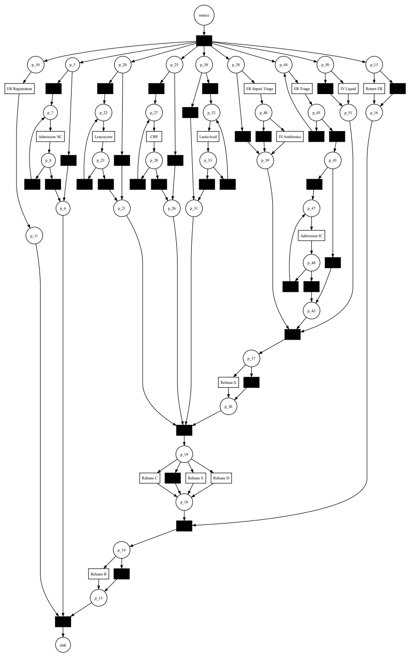
$F1 = 0.85$
$F = 0.76$
$P = 0.97$
$G = 0.90$
$S = 0.66$
$OF = 0.82$

H Discovered Petri Nets - Sepsis



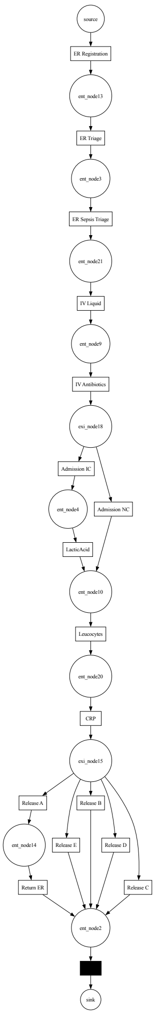
GTM-60:

$$\begin{aligned} F1 &= 0.99 \\ F &= 0.98 \\ P &= 0.99 \\ G &= 0.95 \\ S &= 0.71 \\ OF &= 0.94 \end{aligned}$$



IM:

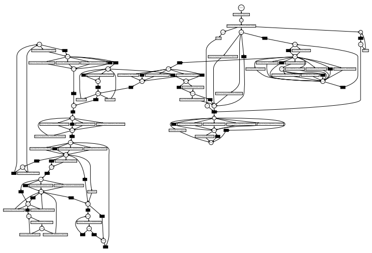
$$\begin{aligned} F1 &= 0.49 \\ F &= 1.00 \\ P &= 0.32 \\ G &= 0.90 \\ S &= 0.62 \\ OF &= 0.66 \end{aligned}$$



SM:

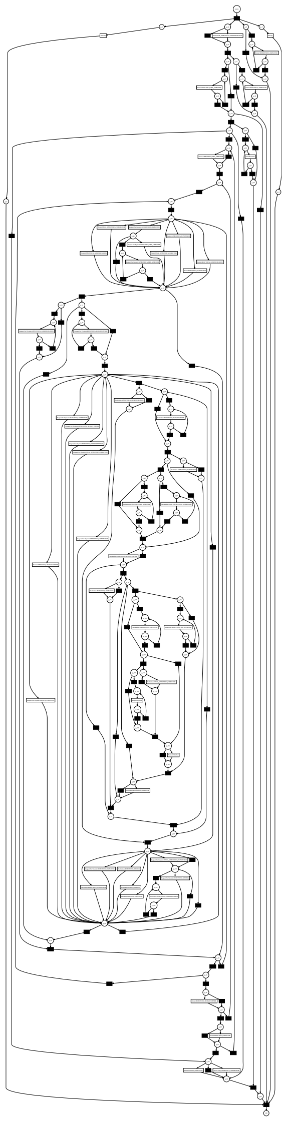
$$\begin{aligned} F1 &= 0.81 \\ F &= 0.69 \\ P &= 0.98 \\ G &= 0.92 \\ S &= 0.79 \\ OF &= 0.81 \end{aligned}$$

I Discovered Petri Nets - 2020-pl



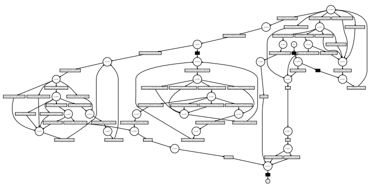
GTM-300:

$F1 = 0.98$
$F = 0.97$
$P = 1.00$
$G = 0.87$
$S = 0.54$
$OF = 0.90$



IM:

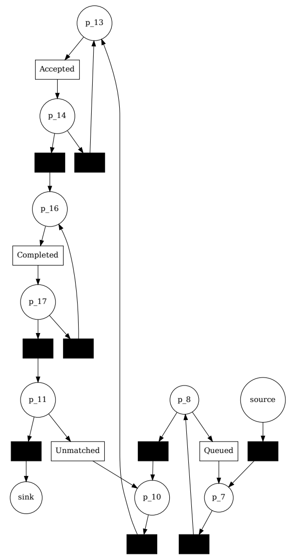
$F1 = 0.18$
$F = 1.00$
$P = 0.10$
$G = 0.86$
$S = 0.59$
$OF = 0.52$



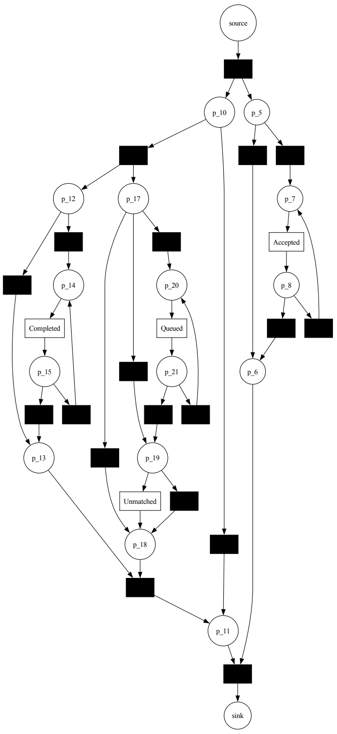
SM:

$F1 = 0.81$
$F = 0.70$
$P = 0.96$
$G = 0.86$
$S = 0.65$
$OF = 0.77$

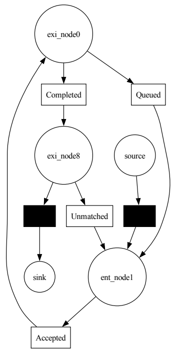
J Discovered Petri Nets - 2013-i



GTM-60:
$$\begin{aligned} F1 &= 0.97 \\ F &= 0.97 \\ P &= 0.98 \\ G &= 0.96 \\ S &= 0.79 \\ OF &= 0.95 \end{aligned}$$

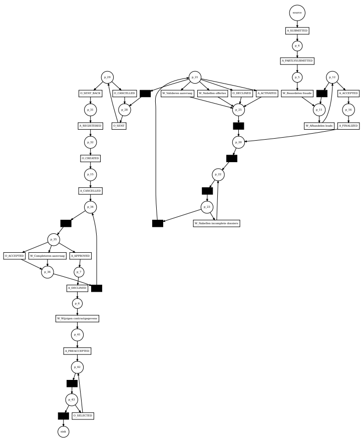


IM:
$$\begin{aligned} F1 &= 0.77 \\ F &= 1.00 \\ P &= 0.63 \\ G &= 0.87 \\ S &= 0.67 \\ OF &= 0.79 \end{aligned}$$



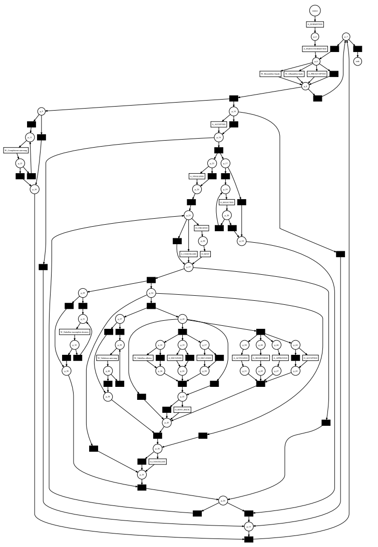
SM:
$$\begin{aligned} F1 &= 0.87 \\ F &= 0.77 \\ P &= 1.00 \\ G &= 0.92 \\ S &= 0.85 \\ OF &= 0.84 \end{aligned}$$

K Discovered Petri Nets - 2012



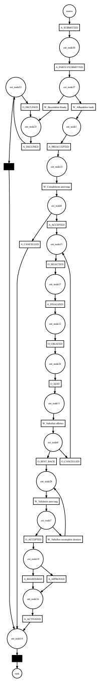
GTM-300:

$$F1 = 0.96$$
$$F = 0.93$$
$$P = 1.00$$
$$G = 0.98$$
$$S = 0.77$$
$$OF = 0.92$$



IM:

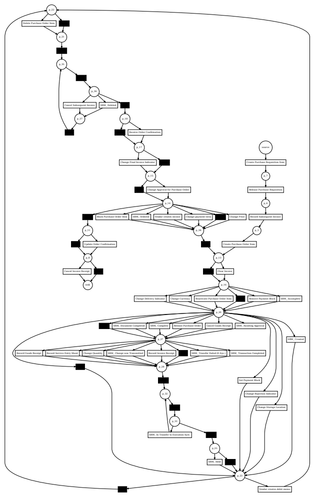
$$F1 = 0.24$$
$$F = 0.97$$
$$P = 0.14$$
$$G = 0.95$$
$$S = 0.61$$
$$OF = 0.56$$



SM:

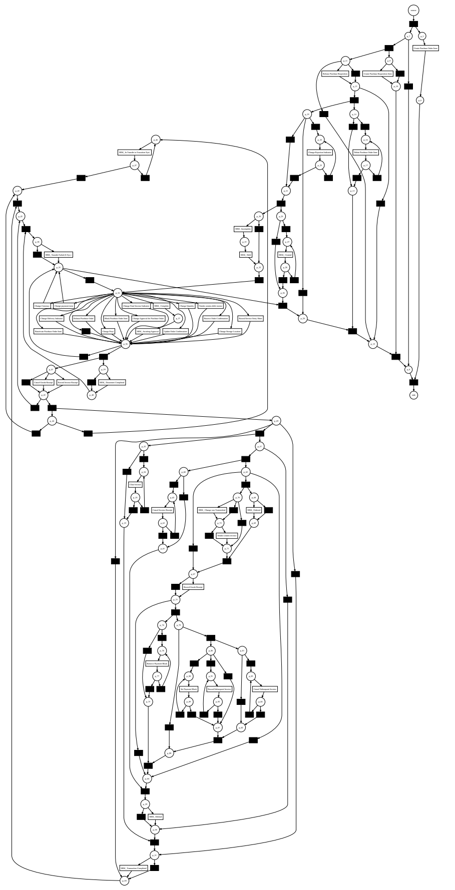
$$F1 = 0.64$$
$$F = 0.49$$
$$P = 0.92$$
$$G = 0.98$$
$$S = 0.82$$
$$OF = 0.68$$

L Discovered Petri Nets - 2019



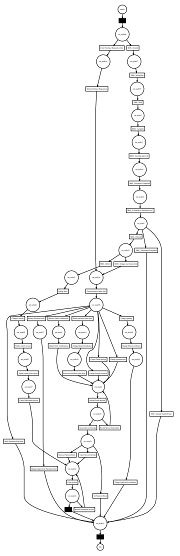
GTM-300:

$$\begin{aligned} F1 &= 0.99 \\ F &= 0.99 \\ P &= 1.00 \\ G &= 0.94 \\ S &= 0.54 \\ OF &= 0.92 \end{aligned}$$



IM:

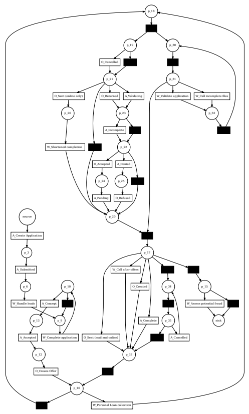
$$\begin{aligned} F1 &= 0.38 \\ F &= 1.00 \\ P &= 0.23 \\ G &= 0.92 \\ S &= 0.59 \\ OF &= 0.56 \end{aligned}$$



SM:

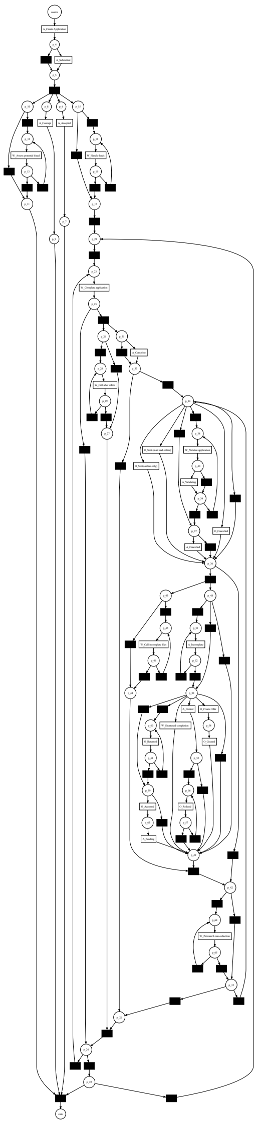
$$\begin{aligned} F1 &= 0.68 \\ F &= 0.51 \\ P &= 1.00 \\ G &= 0.91 \\ S &= 0.73 \\ OF &= 0.70 \end{aligned}$$

M Discovered Petri Nets - 2017



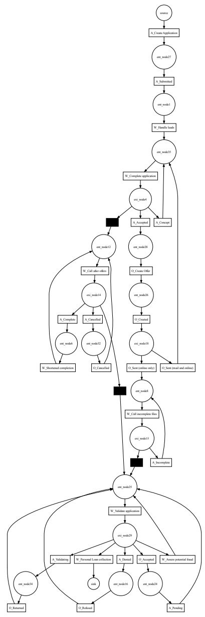
GTM-300:

$$\begin{aligned} F1 &= 0.89 \\ F &= 0.99 \\ P &= 0.80 \\ G &= 0.99 \\ S &= 0.65 \\ OF &= 0.87 \end{aligned}$$



IM:

$$\begin{aligned} F1 &= 0.26 \\ F &= 1.00 \\ P &= 0.15 \\ G &= 0.95 \\ S &= 0.63 \\ OF &= 0.63 \end{aligned}$$



SM:

$$\begin{aligned} F1 &= 0.76 \\ F &= 0.71 \\ P &= 0.80 \\ G &= 0.95 \\ S &= 0.73 \\ OF &= 0.75 \end{aligned}$$