GTM: An Enhanced Genetic Algorithm for Process Discovery

Jeppe Berg Axelsen, Frederik Hecter Kowalski, Richard Nygård and Jiří Srba

Dept. of Computer Science, Aalborg University, Selma Lagerlofs Vej 300, Aalborg, Denmark

Abstract

Process discovery plays a crucial role in extracting meaningful process models from event logs, with the goal of balancing four key quality dimensions: fitness, precision, generalization, and simplicity. While genetic algorithms offer a flexible approach to optimizing these often conflicting criteria, they traditionally suffer from long convergence times and limited competitiveness compared to state-of-the-art process discovery algorithms. We advance the field of genetic process discovery by introducing a revised approach, named Genetic Tree Miner (GTM), that is based on tournament selection of individuals that represent process trees and automatic identification of the best parameters for achieving optimal performance. We significantly speed-up the convergence process by implementing fast estimates of fitness and precision measures that we run on a strategically selected subset of the event log. Empirical evaluations on real-world benchmarks demonstrate that GTM achieves performance exceeding leading process discovery methods, often significantly improving classical measures, at the expense of slower but acceptable running times where even large logs can be analyzed in less than a minute. Given that genetic algorithms can be trivially parallelized, our approach shows promising potential for the application of genetic algorithms in today's process discovery tools.

Keywords

process discovery, genetic algorithms, process trees, evaluation

1. Introduction

Process discovery [1], a critical component of process mining [2], aims to construct accurate models of business processes from event logs. Genetic algorithms (GAs) [3] have been explored for this purpose due to their flexibility in designing objective functions tailored to specific optimization criteria and process characteristics. However, prior studies [4, 5], including those evaluating state-of-the-art methods like Split Miner [5], have often criticized GAs for their computational inefficiency and slow performance, rendering them less competitive compared to modern process discovery techniques [6, 5]. Despite their potential to discover process models that balance several optimization criteria, these limitations have hindered their widespread adoption in practical applications.

We address these challenges of GAs by introducing a refined genetic approach, Genetic Tree Miner (GTM), that significantly enhances both performance and in particular speed. Through targeted optimizations, we overcome the inefficiencies previously associated with GAs, enabling them to compete effectively with existing process discovery approaches.

As our main contributions, we enhance the genetic approach by providing a fast C++ implementation of fitness and precision metrics, outperforming current Java/Python tools by an order of magnitude, and we show that it is sufficient to focus on a weighted sum of these two measures together with a refined simplicity metric. We deploy Bayesian methods to synthesize suitable hyperparameters of our GA and streamline the used architecture to avoid caching, reduce memory use, and ensure that all event log activities are uniquely represented. Last but not least, we validate our approach on an extensive set of real-world benchmarks, demonstrating its ability to produce high-quality process models with improved computational efficiency. Our results highlight the potential of refined GAs to serve as a robust and scalable solution for process discovery, offering a compelling alternative to established methods.

Algorithms & Theories for the Analysis of Event Data 2026

1.1. Related Work

Early process discovery approaches, like the α-algorithm [7] and its variants, extract simple control-flow patterns from logs by analyzing the directly-follows graph (DFG) to identify patterns such as sequence, concurrency, and choice, however, they often struggle with real-life logs. Heuristic Miner [8] addresses some of the limitations of the α-algorithm and can handle incomplete and imprecise logs, but it often produces unsound and complicated models. Other approaches like ILP Miner [9] can guarantee maximum fitness, however, suffer from high computational complexity and poor scalability [10].

The state-of-the-art methods used in the current process discovery tools include Inductive Miner (IM) [6] and Split Miner (SM) [5, 11]. IM constructs block-structured process trees by recursively splitting the DFG of the event log, and its core strength is that it guarantees soundness and perfect fitness. More recently, SM [5, 11] has emerged as a reliable process discovery algorithm designed to balance model quality in terms of fitness, precision, generalization, and simplicity. It constructs a DFG and then applies a gateway-splitting procedure to identify and differentiate between exclusive choices and parallel branches in the discovered BPMN (business process modelling notation) model. SM excels in speed compared to IM and other methods and guarantees deadlock-freedom for cyclic process models and soundness for acyclic.

The use of GA for solving the problem of process discovery has already been investigated [12, 13, 14, 15, 16, 17, 18, 19]. The initial work [12] outlined a possible application of GAs to process discovery, nevertheless, it failed to show the practical deployment on real-life event logs. An extension of this work is the Evolutionary Tree Miner (ETM) [13], which is a flexible algorithm allowing the user to guide the discovery of process trees by weighting the four quality measures. As ETM uses process trees as the internal representation, it is guaranteed to produce sound workflow nets [20]. However, a recent evaluation [5] of ETM on real-life event logs indicates that even after 60 minutes of run time, it often fails to produce well-balanced process models and is not competitive with IM and SM. Similarly the work on diversity-guided evolutionary mining (DGEM) [16] uses processes graphs to represent processes and it also struggles with real-world logs and does not guarantee soundness.

More recently, an improved hybrid approach called DEMiner [17] was proposed. It combines an overall exploration via a genetic algorithm with guided local search and uses the causal matrix to represent individuals in a population. DEMiner, similarly as ProDiGen [18] which also uses causal matrix chromozome representation, showed encouraging results on smaller synthetic logs, however, did not demonstrate the capability to discover well-balanced models from noisy logs and real-world logs.

The current state-of-the-art method in genetic algorithms is implemented in the tool X-Processes [19], claiming to improve both the effectiveness and efficiency of earlier GA methods while being superior to the best non-genetic discovery algorithms at the expense of slower running time. X-Processes uses binary causal matrices as the basic process model and we include this method to our experimental evaluation and confirm their conclusion about the effectiveness to discover sound and balanced process models while using harmonic-mean of all four evaluation criteria (fitness, precision, generalization and simplicity) as its optimization function. The method works well on smaller real-world logs but cannot efficiently deal with the larger logs that are also included in our benchmark.

Our GTM method discovers process trees like e.g. ETM, however, we introduce several innovations that enhance performance, scalability, and quality of discovered process models. To mitigate the slow convergence of GAs, we implement the most CPU-demanding components like fitness and precision evaluation in C++, offering significant runtime improvements over previous tools developed in Java or Python [11, 21]. Additionally, we introduce a mutation specifically designed to capture simple self-loop behavior. Our objective function does not include the generalization metric, and we instead refine the simplicity measure to speed up the performance. Similar to ETM, we employ tournament selection in the crossover process.

In addition, we conduct an exhaustive hyperparameter search using Bayesian optimization [22] to find suitable and robust parameters that work for a wide range of event logs. Unlike other competing genetic 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 the inspection.

Collectively, these contributions lead to a robust and efficient genetic process discovery approach that is outperforming X-Processes, IM as well as SM. While the IM and SM methods are still faster than our GTM algorithm, we manage to find superior quality models for the logs in the benchmark in less than one minute.

2. Process Discovery Background

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 naturally extend the labelling function ℓ 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 all occurrences of τ are deleted.

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. 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 $t_0, t_1, \dots, t_n \in T$, we write $M_0[t_0 t_1 \dots t_n\rangle M_n$ (or simply $M_0 \to^* M_n$ if we are not interested in transition names) if there are markings $M_1, \dots, M_{n-1}$ such that $M_i[t_i\rangle M_{i+1}$ for all i, $0 \leq i < n$. 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 $t_0, t_1, \dots, t_n \in T$ such that $M_0[t_0 t_1 \dots t_n\rangle M_n$ for some marking M_n and $\ell(t_0 t_1 \dots t_n) = \pi$. A trace is complete if it cannot be extended to a longer trace. The set of all complete traces (language) of a marked Petri net (N, M_0) is denoted by $\mathcal{T}(N, M_0)$. In Figure 1c we can see an example of a Petri net where places are denoted by circles and transitions by rectangles annotated with the activity names. The set of complete traces of the net can be described by the regular expression $ab + ad^*cd^*$.

We now recall the notion of sound WF-nets (see e.g. [2]).

Definition 1 (Sound WF-net). *A Petri net $N = (P, T, F, \ell)$ is a workflow net (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_o be the final marking where $M_o(o) = 1$ and all other places in M_i and M_o have no other tokens. A WF-net is sound *if*

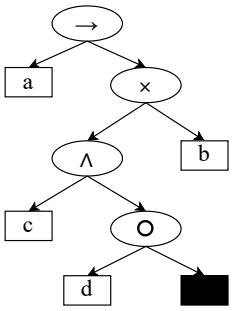
- *$M \to^* M_o$ for all markings M where $M_i \to^* M$, and*
- *if $M_i \to^* M$ and $M(o) \geq 1$ then $M = M_o$.*

The net in Figure 1c is a sound WF-net.

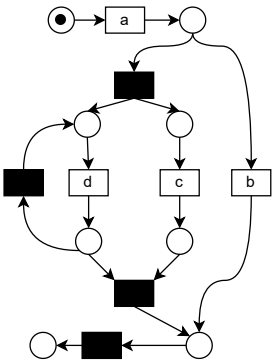
We are now ready to define the *process discovery problem.* For a given event log $L \subseteq \mathcal{B}(A^*)$, i.e. a finite multiset of traces over the set of activities A, we want to algorithmically construct a sound WF-net N such that $\mathcal{T}(N, M_i)$ describes in the best possible way the event log L. The requirements on the process model N usually include: *fitness* [23] meaning that all (or most of the) traces from L are

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 represeting the set of traces $L = \{ab, acdd, addc\}$



(b) Process tree describing the language $ab + ad^*cd^*$, the 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)

Figure 1: Event log, its process tree model and the corresponding Petri net

included in $\mathcal{T}(N, M_i)$, *precision* [24] that requires that the set of traces in $\mathcal{T}(N, M_i)$ should be as small as possible while still guaranteeing fitness, *generalization* [25] meaning that traces that are not in L but are the likely to be produced by the process should be included in $\mathcal{T}(N, M_i)$ as well and *simplicity* [18] that prefers simpler (smaller) models that can express the process behaviour. These four measures are often contradiciting and we aim for a good balance between them. This can be expressed by an *objective function* that computes, for example, a weighted average over the four measures.

Our genetic algorithm instead on Petri nets operates on process trees (see e.g. [2, p. 80]). We shall now inductively define 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 2 (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 $\mathcal{T}(Q) = \mathcal{T}(Q_1) \parallel \ldots \parallel \mathcal{T}(Q_n)$,
- $Q \equiv \bigcirc(Q_1, \ldots, Q_n) \in \mathcal{P}$ *(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 resp. Kleene star operations on languages, and the

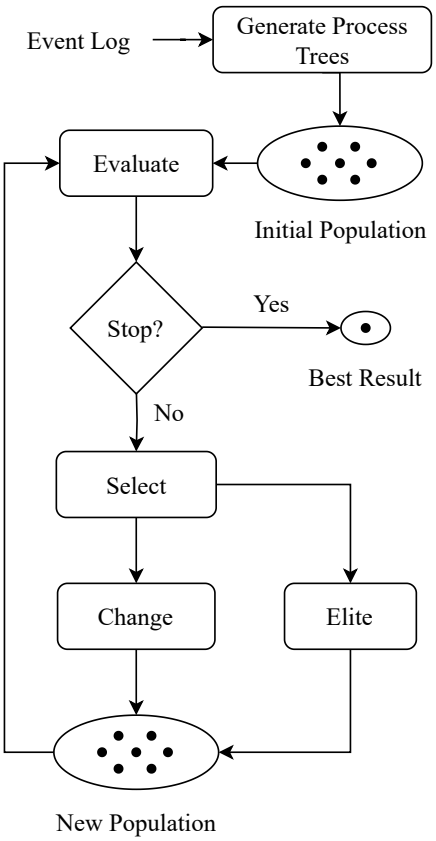


Figure 2: High-level overview of genetic process discovery

shuffle (interleaving) operator $\parallel$ *used in parallel composition is defined as*

$$aw \parallel a'w' = \{a \circ (w \parallel a'w')\} \cup \{a' \circ (aw \parallel w')\}$$

for $a, a' \in A$ *and* $w, w' \in A^*$*, where* $w \parallel \epsilon = \epsilon \parallel w = \{w\}$*. It is extended to languages as follows:*

$$L \parallel L' = \bigcup_{w \in L, w' \in L'} w \parallel w' \,.$$

An example of a process tree (in the standard graphical notation) is given in Figure 1b. The trace language includes all the traces from the log L depicted in Figure 1a and we hence achieve a perfect fitness. It is well known [20] that for every process tree $Q \in \mathcal{P}$ there is a sound WF-net (N, M_i) such that $\mathcal{T}(Q) = \mathcal{T}(N, M_i)$. Figure 1c shows such a WF-net corresponding to the process tree in Figure 1b; it was obtained by an automatic translation provided in the PM4Py library [21].

3. Genetic Tree Miner

We shall now describe the overall architecture of GTM.

3.1. General Approach

In our approach, illustrated in Figure 2, we adopt the overall structure of genetically motivated algorithms [3]: from the input event log, we produce an initial population of process trees on which we evaluate a given objective function. Until we reach a stagnation criterion or a timeout, we continuously select a number of the best elite individuals and modify the remaining ones in order to form a new

Algorithm 1 Genetic Tree Miner

```
P_1 ← generate POPULATION_SIZE individuals
elite_count ← POPULATION_SIZE × ELITE_RATE
random_count ← POPULATION_SIZE × RANDOM_CREATION_RATE
tournament_count ← POPULATION_SIZE × TOURNAMENT_RATE
tournament_size ← POPULATION_SIZE × TOURNAMENT_SIZE_RATE
for i = 1 to NUM_GENERATIONS do
    survivors ← select elite_count best performing individuals in P_i
    new_individuals ← select random_count randomly generated individuals
    P_{i+1} ← survivors ∪ new_individuals
    for j = 1 to tournament_count do
        tournament ← randomly select tournament_size individuals from the P_i.
        parent1, parent2 ← two best individuals from tournament
        child ← do crossover with parent1 and parent2
        child ← mutate child with MUTATION_PROBABILITY
        P_{i+1} ← P_{i+1} ∪ {child}
    end for
    if stagnation criterium is met then
        return best individual in P_{i+1}
    end if
end for
return best individual in P_{NUM_GENERATIONS}
```

population, and repeat the whole process. We introduce several modifications to its internal mechanisms, including redesigning the building blocks for generating process trees, evaluation, selection, and change.

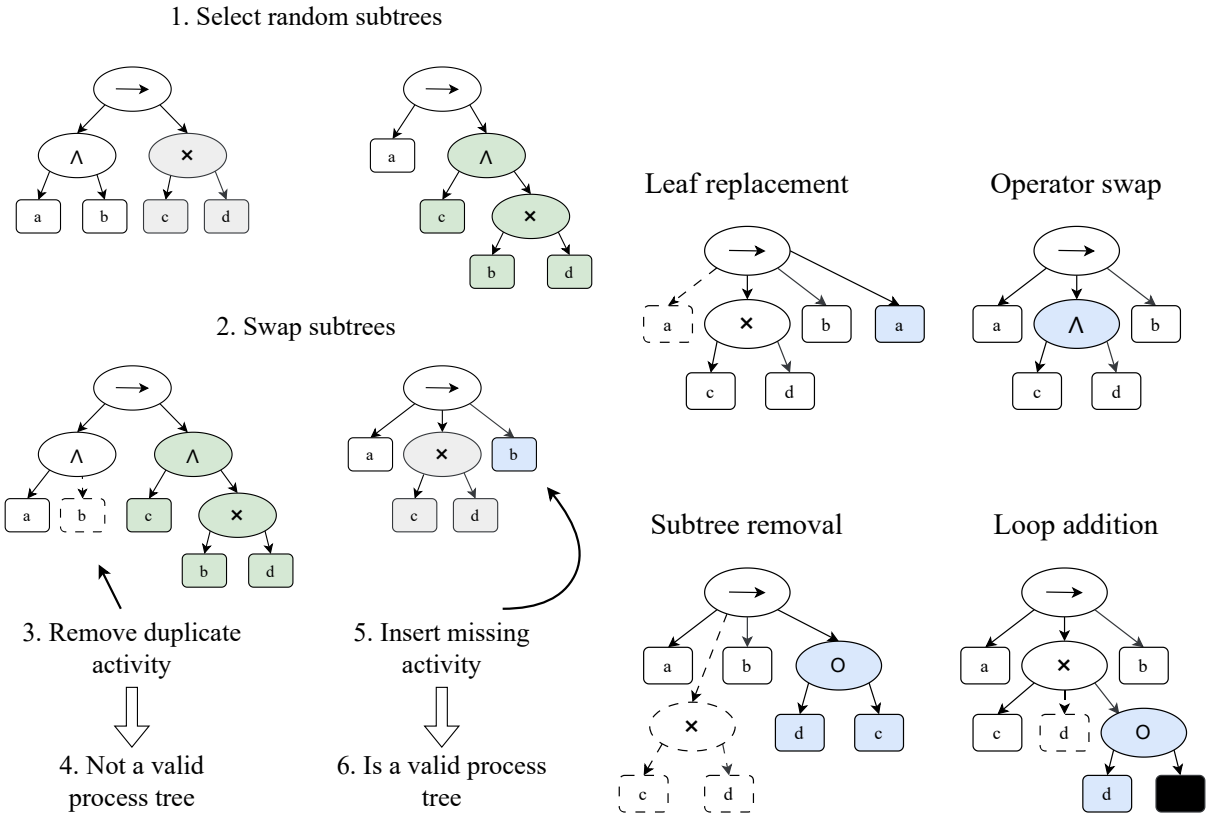
Algorithm 1 depicts the details of our genetic algorithm. It repeatedly selects a new generation by creating a new population of candidates from the previous generation. Based on a given objective function, we first select a percentage of the best-performing individuals as an elite set, which we move directly into the new population. A new population also includes some new randomly generated trees. The remaining candidates are created by a *tournament selection* [26] on the recently evaluated population: we randomly select a number of individuals for a tournament, pick two best individuals from the tournament based on the objective function and perform a crossover (see Section 3.2) followed by a random mutation (see Section 3.3). 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. The algorithm terminates either by reaching the given number of generations or if no significant improvement among a fixed number of last generations is observed.

3.2. Crossover

A crossover involves two process trees (called parents) 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 3a. For each child, we remove duplicate activities and randomly insert missing activities. Then we check if the children are valid process trees, i.e. satisfy Definition 2. If the first child or second child is a valid process tree, it is returned (left child has a priority). If none of the children are a valid process tree, we randomly return one of the parents.

3.3. Mutations

To apply a mutation operation to a process tree, we randomly perform one of the following four mutation strategies as depicted in Figure 3b, showing how the possible mutations transform the process tree of the



(a) Crossover example showing subtree exchange between two parent process trees. Grey and green indicate the selected subtrees; dashed lines show removed elements, and blue elements are inserted.

(b) Examples of mutation operations on the process tree resulting from Figure 3a. Dashed lines indicate removed elements, and blue color indicates inserted elements.

Figure 3: Crossover and mutation examples

valid child from the crossover example in Figure 3a. 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.

3.4. 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 (denoted by the constant INITIAL_SAMPLING_RATE) of the traces in the event log. If an activity from the initial event log is not represented in the subset, we find a trace where the missing activity appears and add the trace to the subset. Thereby, such a subset of traces contains all unique activities from the initial event log. We treat the subset of traces as an event log and use it to discover a process tree by the IM. Repeating this process several times, we obtain a population of process trees discovered on different aspects of the event log. In this way, we obtain a population diversity in the initial population as well as a fast method to create the initial population.

3.5. Evaluation of Objective Function and Stopping Criteria

In the evaluation step, we measure the replay fitness, precision, and simplicity of each candidate process tree by an objective function, which is a weighted average of these measures (for precise definition

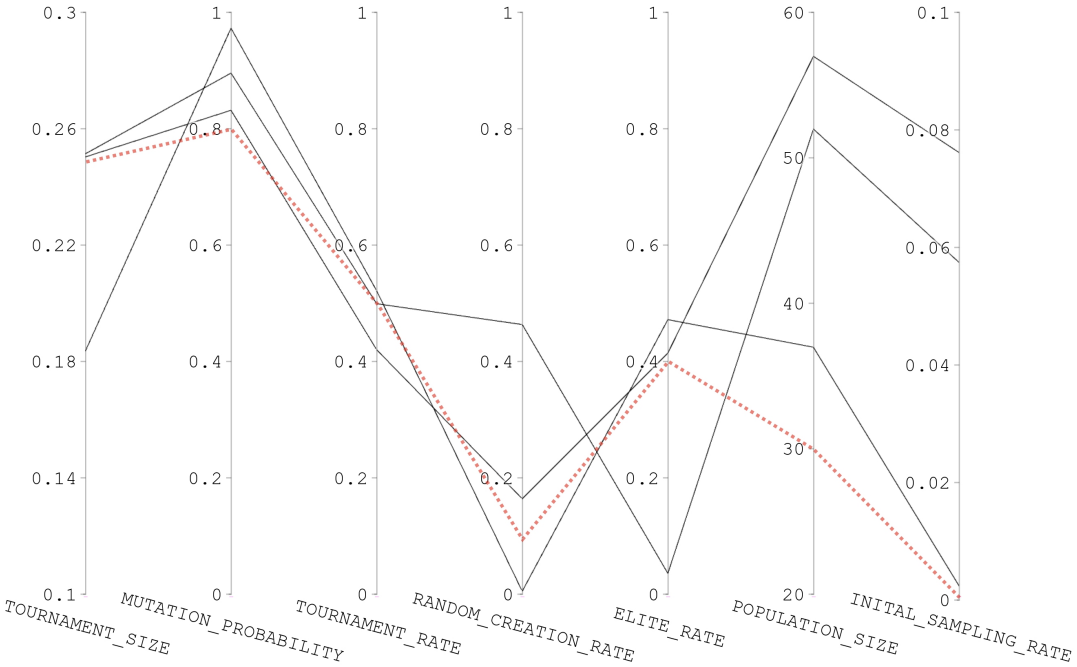


Figure 4: Bayesian optimization results on three event logs: 2013-cp, 2012 and Sepsis. The red dotted line highlights the proposed values for each hyperparameter.

see Section 5.1). The evaluation of fitness and precision is a time-consuming step in the algorithm, so minimizing the evaluation time is a major focus in our implementation. To speed up the evaluation of the objective function for every individual, we utilize a fixed subset of the event log, including only traces that appear most frequently. How large subset we choose depends on the number of unique traces and is dictated by $f(t) = 0.5987 \cdot e^{-2.251 \cdot 10^{-4} \cdot t}$, where t is the number of unique traces. The function is designed to balance the speed of the discovery algorithm and ensures that on the smaller logs we consider about 60% of unique traces and on the larger logs we consider only about 5%. 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. After an evaluation, we check whether one of two given stopping criteria is met. If so, the algorithm terminates and returns the best result (a process tree with the best objective function score in the last population). We stop once the algorithm stagnates for a given number of generations (no significant improvement of the objective function is observed) or after we reach a timeout.

4. Hyperparameters, Convergence and Stability

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 [27], on three event logs from the BPI Challenge [28]: 2013-cp [29], 2012 [30] and Sepsis [31]. BO optimizes the objective function which is a weighted sum of four quality metrics: replay fitness (weight 0.5), precision (weight 0.3), simplicity (weight 0.1), and refined simplicity measuring the number of places (weight 0.1). The results are depicted in Figure 4, and each black line represents the set of values that achieved the maximal objective score after 18 hours of running BO. A general tendency forms across the three datasets, leading to fixing the following hyperparameter values (depicted by the red line in the figure): 0.25 (tournament size rate), 0.8 (mutation probability), 0.5 (tournament rate), 0.1 (random creation rate),

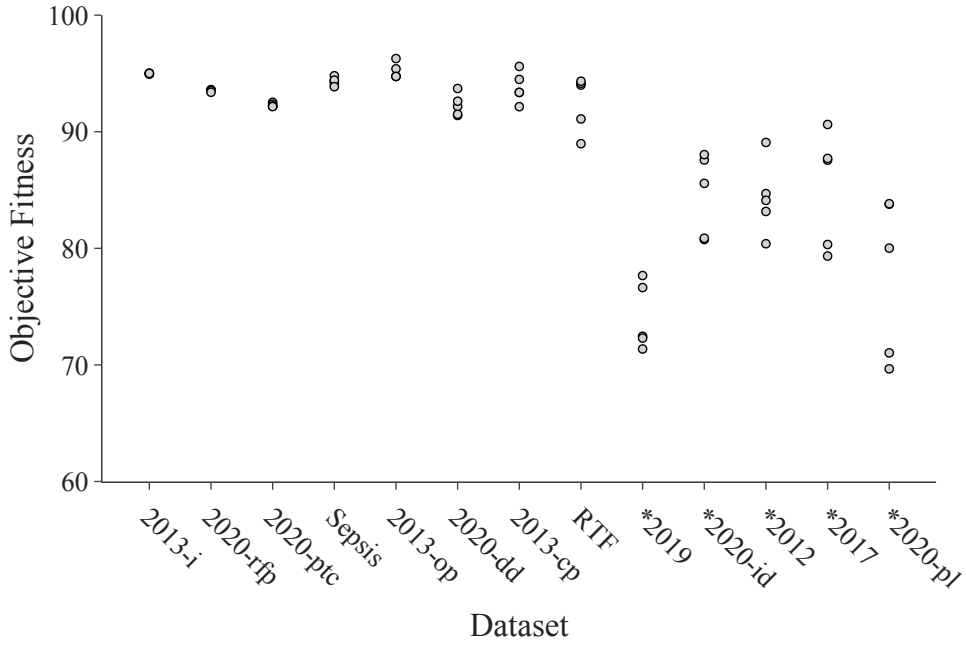


Figure 5: Objective fitness variance after five runs of GTM; the stars mark the datasets where stagnation was not achieved within the given one minute timeout

0.4 (elite rate), 30 (population size) and 0.001 (initial sampling rate). On purpose, we choose a lower population size and an initial sampling rate to speed up the computation.

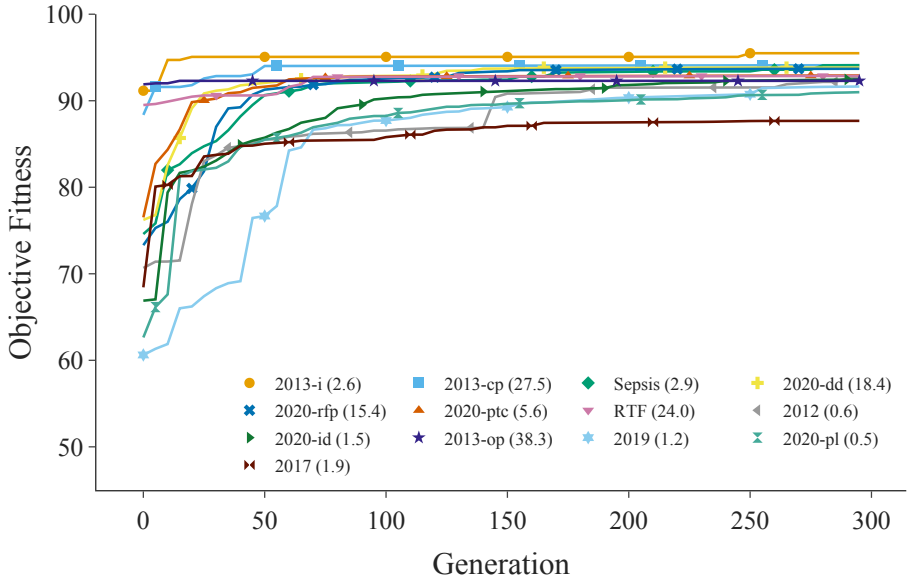
Next, we investigate the stability of GTM due to the stochastic nature of genetic algorithms. Figure 5 shows five distinct runs of GTM on each dataset with a stagnation limit of less than 1% improvement in 50 generations and a 60 second timeout. Each circle represents the objective fitness (value of the objective function) of the discovered model, showing the variance between different runs. In 8 out of 13 event logs, the variance is negligible; a larger variance can be seen for event logs on which the algorithm timed out (marked with a star), and greater stability can be achieved by increasing the timeout limit.

In addition to stability, we investigate the convergence speed of GTM. In Figure 6a, each line shows the evolution of the objective fitness over 300 generations. The convergence rate varies depending on the log size and complexity, however, GTM generally stabilizes mostly before 100 generations.

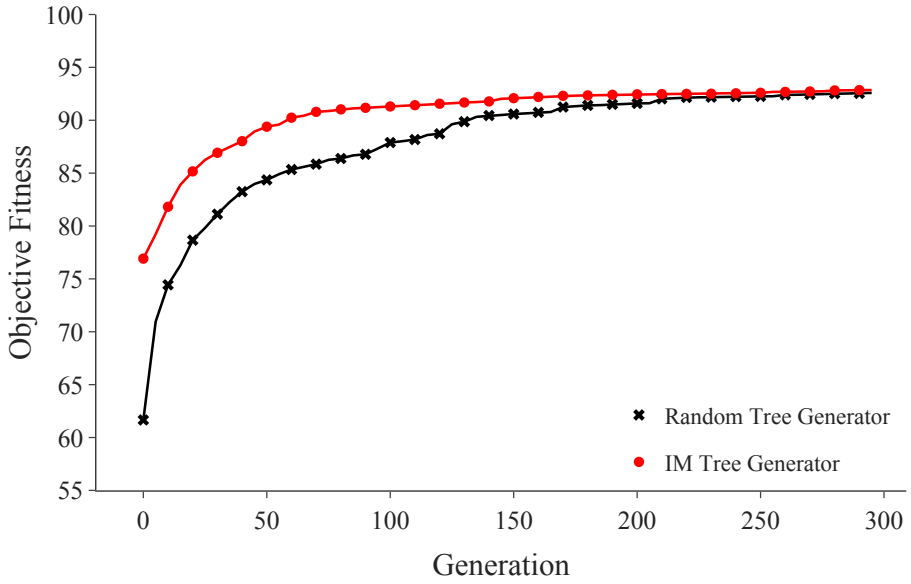
Finally, we compare the convergence rate of GTM when using random trees in its initial population (randomly generated trees containing all unique activities in the event log), instead of the default initial population method based on IM as described in Section 3.4. The results in Figure 6b highlight the advantage of the GTM method over the random initial population, as it achieves better objective scores in the early generations, which is important for larger logs, where the algorithm terminates due to a timeout. On the other hand, it also demonstrates that we propose well-performing crossover and mutation changes that guarantee convergence to optimal objective scores even without injecting IM generated process trees into the initial generation.

5. Implementation and Evaluation

GTM is implemented in Python, which allows for an integration to the PM4py framework [21]. However, key evaluation algorithms, specifically the replay-based fitness and precision, are implemented in C++ for performance reasons. To enable their use within the Python-based implementation, we develop Python bindings that make these C++ evaluation components callable directly from Python. The code is released under open-source license [32].



(a) Convergence rate of GTM on all datasets. The time in parentheses after each log is an average number of generations per second.



(b) Convergence rate of GTM using IM initial population vs. random initial population. Each line represents the mean objective fitness across all datasets in our benchmark.

Figure 6: Convergence results

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 from the BPI Challenge [28], commonly used for benchmarking in process discovery. No preprocessing or filtering is applied on the event logs, the only exception is the 2019 log where 10% filtering was applied in order to be able to execute the SM algorithm. Event log statistics is displayed in Table 1.

We compare the performance of the process discovery algorithms using established variants of the four classic performance measures: replay fitness [23], precision [24], generalization [25] and simplicity [18], all evaluated on the complete logs and computed using the standard implementations available

Event log	Unique Traces	Traces	Average Trace Length	Unique Activities
2020-rfp [33]	89	6886	5	19
2020-dd [34]	99	10500	5	17
2013-op [35]	108	819	2	3
2013-cp [29]	183	1487	4	4
2020-ptc [36]	202	2099	8	29
RTF [37]	231	150370	3	11
2020-id [38]	753	6449	11	34
Sepsis [31]	846	1050	14	16
2020-pl [39]	1478	7065	12	51
2013-i [40]	1511	7554	8	4
2012 [30]	4366	13087	20	23
2019 [41]	11028	226561	6	42
2017 [42]	15930	31509	38	26

Table 1
Overview of event log benchmark

Log Name	Discovery Method	Accuracy			Generalization	Simplicity	Objective Fitness	Time (s)
		F1-score	Fitness	Precision				
2013-cp	GTM	**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	-
	X-Proc	0.94	0.95	0.97	0.92	0.76	0.91	10.98
2012	GTM	**0.78**	0.94	0.67	0.90	0.66	**0.84**	60.05
	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	-
	X-Proc	-	-	-	-	-	-	$>$ 3600
Sepsis	GTM	**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	-
	X-Proc	0.76	0.65	**1.00**	0.90	0.68	0.77	72.52
Average over all eventlogs	GTM	**0.91**	0.94	0.89	0.93	0.74	**0.91**	34.38
	IM	0.54	**0.99**	0.42	0.91	0.63	0.69	7.25
	SM	0.72	0.60	0.97	**0.94**	**0.87**	0.77	-
	X-Proc*	0.85	0.80	**0.99**	0.91	0.72	0.84	$>$ 1228

Table 2
Performance comparison of process discovery methods on the three logs used by GTM for parameter identification; the averages marked as X-Proc* are only over the two computed values

in the PM4Py library [21]. We also report the value of the objective function, which is a weighted sum of the C++ replay fitness (0.5 weight), C++ precision (0.3 weight), PM4Py simplicity (0.1 weight) and our own refined simplicity (weight 0.1). The refined simplicity is defined as $max\{0, 1 - \frac{\#places}{100}\}$. This is the objective function that our GTM algorithm optimizes for in order to obtain well-balanced models that put higher weights to the fitness and precision. However, note that the values for the four performance measures listed in all our result tables are computed using the standard PM4Py library.

To evaluate the effectiveness of GTM, we compare it against two widely-used algorithms, Inductive Miner (IM) [6] and Split Miner (SM) [5], as well as the state-of-the-art genetic miner X-Processes [19] (abbreviated as X-Proc). 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 (http://apromore.org/) with its default parameters. These two methods are chosen due to their strong performance and frequent use in both academic and commercial settings. The X-Proc algorithm [19] is also run using the default parameters, while using the harmonic mean of all four measures as the optimization criterion.

All experiments are run on a macbook with Apple M2 3.49 GHz CPU, 16 GB of RAM with MacOS

Sequoia 15.4.1 using Python 3.13 and PM4Py 2.7.15.2. No multiprocessing is utilized in any of the experiments, and GTM, X-Proc and IM are run on the same hardware. SM is run on Apromore servers, and no timing information is provided, however, other studies already showed that process discovery using SM is in general faster than by IM [5].

Our GTM is run with a maximum runtime of 60 seconds and a stagnation limit of less than 1% improvement during the last 50 generations. As X-Proc has often problems with convergence, we let it run for a full hour. To account for stochasticity in GTM and X-Proc, each configuration is executed five times, and we report the median values. The source code, datasets, and instructions for reproducing the results are publicly available in [32].

5.2. Results

Tables 2 and 3 present comprehensive comparisons of GTM, IM, SM and X-Proc across the benchmark. We note that we identified the default parameters of our genetic algorithm only by using the three logs from Table 2 and our algorithm was not optimized for any of the logs in Table 3 where the default parameters (identified earlier) are used. The quality of the discovered model of each method is evaluated on the complete event log using the quality metrics computed using PM4Py: F1-score, fitness, precision, generalization, simplicity. We also report on the objective fitness (value of the objective function) and the runtime.

Overall, GTM in general outperforms all other methods on F1-score and achieves the best balance between fitness and precision, the most critical evaluation aspect of any discovery algorithm. Generally, the IM exhibits high replay fitness but tends to yield low precision. In contrast, the SM often achieves significantly higher precision than IM, albeit with reduced replay fitness. GTM provides a more balanced trade-off, demonstrating strong performance on both precision and replay fitness in most scenarios. However, on one of the largest event logs, namely 2019, GTM produces a process model with a lower F1-score than SM. If GTM is instead run for 5 minutes on 2019, the achieved F1-score of 0.99 now exceeds the result achieved by SM. The competing X-Proc genetic algorithm achieves on average the second best F1-score, confirming the findings from [19] where the authors argued that it produces better quality models than IM and SM. The downside of the X-Proc approach is longer running time; in particular the algorithm did not converge on the four largest logs from our benchmark within one hour. Three of these logs were also used in X-Proc own evaluation paper [19] where the authors had to run their algorithm for 24 hours (on a different hardware) before usable process models were obtained.

With respect to generalization, all methods exhibit comparable performance, as also reflected by similar average scores. On the simplicity dimension, SM performs very well as demonstrated by the best simplicity on most of the event logs. This is also reflected in the average, where SM is noticeably better than the three other methods, however, often at the expense of low fitness.

All discovered Petri nets can be seen in the appendix. Here we discuss in detail two relatively smaller logs with a lower number of different activities, namely 2013-op and RTF. The reason is that the process models of these event logs can be more easily visualized in the figures. If necessary, the discovered BPMN or process tree models are converted to Petri nets by using PM4Py library and the nets are annotated with the F1-score and the four performance metrics.

Figure 7 depicts the discovered models for the 2013-op event log. The log contains only three activities and therefore the discovered nets are simple, except for IM that discovered a rather complex model, however, with the maximum fitness (as expected). SM and X-Proc both achieve the highest precision but with lower fitness (in particular SM). GTM and IM have the highest F1-scores but our GTM model is clearly preferred due its simplicity.

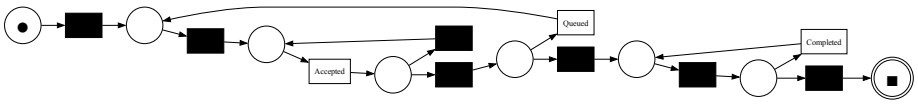
Similarly, Figure 8 shows the discovered workflow nets for the RTF log with 11 unique activities. On this log, GTM achieves significantly higher F1-score than the remaining three competing methods. The second best performing method is this time SM which provides better replay fitness than X-Proc. IM is again excelling in the maximum replay fitness but suffers from poor precision and high complexity of the discovered net.

In conclusion, GTM achieves in general the most balanced performance at the expense of longer but

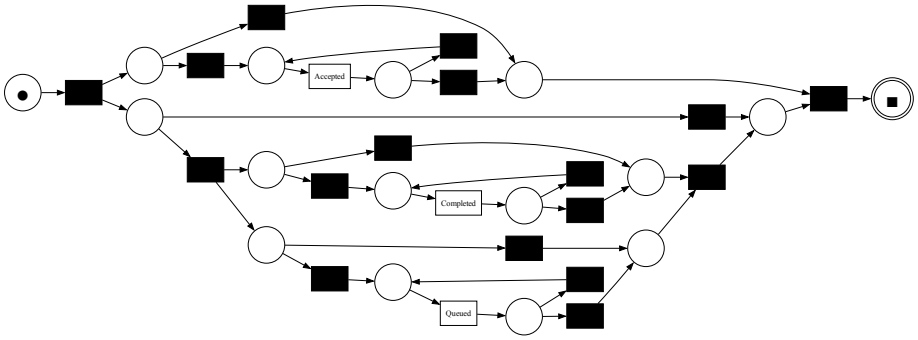
Log Name	Discovery Method	Accuracy			Generalization	Simplicity	Objective Fitness	Time (s)
		F1-score	Fitness	Precision				
2020-rfp	GTM	**0.99**	0.99	**1.00**	**0.89**	0.62	**0.94**	10.17
	IM	0.50	**1.00**	0.33	0.87	0.60	0.73	0.07
	SM	0.82	0.83	0.82	0.81	**0.66**	0.81	-
	X-Proc	0.85	0.82	0.90	0.81	0.62	0.81	12.18
2020-dd	GTM	**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	-
	X-Proc	0.93	0.92	0.92	0.79	0.62	0.87	10.86
2013-op	GTM	**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	-
	X-Proc	0.91	0.86	**1.00**	0.95	0.88	0.90	10.04
2020-ptc	GTM	**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	-
	X-Proc	0.87	0.83	0.95	0.65	0.53	0.79	32.50
RTF	GTM	**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	-
	X-Proc	0.76	0.60	0.99	0.97	0.82	0.77	123.19
2020-id	GTM	**0.91**	0.92	0.90	**0.91**	0.65	**0.86**	60.01
	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	-
	X-Proc	0.89	0.88	0.89	0.74	0.52	0.75	226.79
2020-pl	GTM	**0.92**	0.86	**0.99**	**0.88**	0.59	**0.80**	60.09
	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	-
	X-Proc	-	-	-	-	-	-	$>$ 3600
2013-i	GTM	**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	-
	X-Proc	0.94	0.91	**1.00**	0.93	0.73	0.90	62.13
2019	GTM	0.55	0.97	0.38	**0.94**	0.59	**0.72**	60.02
	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	-
	X-Proc	-	-	-	-	-	-	$>$ 3600
2017	GTM	**0.92**	0.94	**0.90**	**0.95**	0.66	**0.88**	60.02
	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	-
	X-Proc	-	-	-	-	-	-	$>$ 3600
Average over all eventlogs	GTM	**0.92**	0.95	0.91	**0.93**	0.69	**0.89**	33.18
	IM	0.50	**1.00**	0.38	0.90	0.62	0.65	8.66
	SM	0.82	0.73	0.94	0.90	**0.76**	0.81	-
	X-Proc*	0.88	0.83	**0.95**	0.83	0.67	0.83	$>$ 1611

Table 3
Performance comparison of process discovery methods; GTM is run with the default parameters and is not trained on any of the logs in this table; the averages marked as X-Proc* are only over the computed values

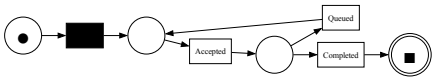
acceptable running times with a timeout stopping criterium of at most one minute. The state-of-the-art X-Proc genetic method is also capable of discovering well-balanced models but is lacking behind GTM both in the overall performance and in particular in the running times where X-Proc clearly struggles with larger event logs in the BPIC benchmark.



(a) **GTM**: $F1 = 0.96, \quad F = 0.95, \quad P = 0.97, \quad G = 0.96, \quad S = 0.82$



(b) **IM**: $F1 = 0.95, \quad F = 1.00, \quad P = 0.91, \quad G = 0.93, \quad S = 0.69$



(c) **SM**: $F1 = 0.82, \quad F = 0.70, \quad P = 0.99, \quad G = 0.96, \quad S = 1.00$

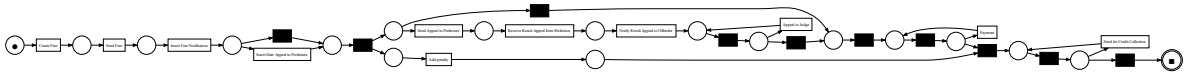
(d) **X-Proc**: $F1 = 0.91, \quad F = 0.86, \quad P = 1.00, \quad G = 0.95, \quad S = 0.88$

Figure 7: Discovered Petri nets for the 2013-op event log

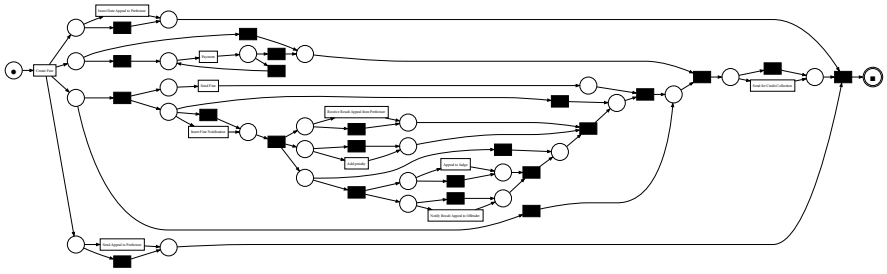
6. Conclusion and Future Work

We introduced GTM, a genetic process discovery algorithm that operates on process trees, thereby guaranteeing the generation of sound workflow models. Like other evolutionary approaches, our method offers high flexibility by allowing optimization against arbitrary user-defined objective functions. Compared to earlier genetic algorithms, we achieved significant performance improvements through a combination of algorithmic strategies and implementation optimizations. These include a dedicated C++ library for computing fitness and precision, a strategic population initialization, and enhanced convergence via tournament selection together with a diverse set of crossover and mutation operators. Furthermore, we observed that evaluating individuals on a subset of the most frequent trace variants suffices to guide the search effectively.

Empirical evaluations demonstrate that GTM is competitive with state-of-the-art techniques in terms of model quality. Considering the F1 score, GTM outperformed all competing methods on 12 out of 13 benchmark event logs using no more than 60 seconds of CPU time on a standard laptop. While on one of the largest logs, split miner outperformed our method in F1 score, GTM still surpassed the other two miners, and with a 5-minute timeout on a single core, we outperform split miner also on this event log. Given the inherently parallel nature of genetic algorithms, we anticipate near-linear speedups with multi-core executions. This positions our method as a practical and scalable solution for process discovery tasks.

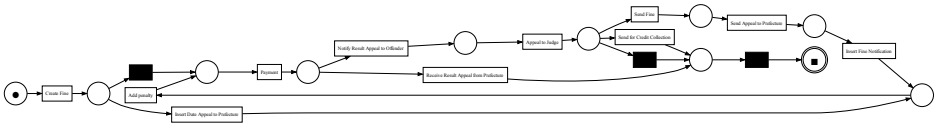


(a) **GTM**: $F1 = 0.97, \quad F = 0.95, \quad P = 1.00, \quad G = 0.99, \quad S = 0.80$



(b) **IM**: $F1 = 0.77, \quad F = 1.00, \quad P = 0.63, \quad G = 0.97, \quad S = 0.62,$

(c) **SM**: $F1 = 0.88, \quad F = 0.79, \quad P = 0.99, \quad G = 0.99, \quad S = 0.92$



(d) **X-Proc**: $F1 = 0.76, \quad F = 0.60, \quad P = 0.99, \quad G = 0.97, \quad S = 0.82$

Figure 8: Discovered Petri nets for the RTF event log

To support the research community, we plan to contribute the GTM implementation, including the optimized fitness and precision evaluation routines, to the PM4Py framework. As future work, we aim to explore the post-processing of the discovered Petri nets by leveraging genetic algorithms to add places capturing long-term dependencies, further improving the accuracy of the resulting models.

Declaration on Generative AI

During the preparation of this work, the authors used different LLMs to improve the writing style in nontechnical parts of the paper. The authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] J.-R. Rehse, S. J. J. Leemans, P. Fettke, J. M. E. M. van der Werf, On process discovery experimentation: Addressing the need for research methodology in process discovery, ACM Trans. Softw. Eng. Methodol. 34 (2024). URL: https://doi.org/10.1145/3672447. doi:10.1145/3672447.

[2] W. van der Aalst, Process Mining: Data Science in Action, Springer, 2016. URL: http://dx.doi.org/10.1007/978-3-662-49851-4. doi:10.1007/978-3-662-49851-4.

[3] A. K. A. de Medeiros, A. J. M. M. Weijters, W. M. P. van der Aalst, Genetic process mining: An experimental evaluation, Data Mining and Knowledge Discovery 14 (2007) 245–304. doi:10.1007/s10618-006-0061-7.

[4] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, A. Soo,

Automated discovery of process models from event logs: Review and benchmark, IEEE Transactions on Knowledge and Data Engineering 31 (2019) 686–705. doi:10.1109/TKDE.2018.2841877.

[5] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, A. Polyvyanyy, Split miner: automated discovery of accurate and simple business process models from event logs, Knowledge and Information Systems 59 (2019). doi:10.1007/s10115-018-1214-x.

[6] S. Leemans, D. Fahland, W. Aalst, Process and deviation exploration with inductive visual miner, CEUR Workshop Proceedings 1295 (2014) 46.

[7] W. van der Aalst, T. Weijters, L. Maruster, Workflow mining: discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 1128–1142. doi:10.1109/TKDE.2004.47.

[8] A. Weijters, W. Aalst, A. Medeiros, Process mining with the heuristics miner-algorithm, Cirp Annals-manufacturing Technology - CIRP ANN-MANUF TECHNOL 166 (2006).

[9] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, A. Serebrenik, Process discovery using integer linear programming, in: K. M. van Hee, R. Valk (Eds.), Applications and Theory of Petri Nets, Springer, 2008, pp. 368–387.

[10] H. M. W. Verbeek, W. M. P. van der Aalst, Decomposed process mining: The ILP case, in: Business Process Management Workshops, Springer International Publishing, 2015, pp. 264–276.

[11] A. Augusto, M. Dumas, M. L. Rosa, Automated discovery of process models with true concurrency and inclusive choices, 2021. URL: https://arxiv.org/abs/2105.06016. arXiv:2105.06016.

[12] W. Aalst, A. Medeiros, A. Weijters, Genetic process mining, An experimental valuation, Data Mining and Knowledge Discovery 14 (2005) 48–69. doi:10.1007/11494744_5.

[13] J. Buijs, B. van Dongen, W. van der Aalst, A genetic algorithm for discovering process trees, in: 2012 IEEE Congress on Evolutionary Comp., 2012, pp. 1–8. doi:10.1109/CEC.2012.6256458.

[14] M. Eck, van, J. Buijs, B. Dongen, van, Genetic process mining: Alignment-based process model mutation, in: Business Process Management Workshops (BPM 2014 International Workshops), Lecture Notes in Business Information Processing, Springer, Germany, 2015, pp. 291–303. doi:10.1007/978-3-319-15895-2_25.

[15] T. Molka, D. Redlich, W. Gilani, X.-J. Zeng, M. Drobek, Evolutionary computation based discovery of hierarchical business process models, in: Business Information Systems, Springer International Publishing, 2015, pp. 191–204.

[16] T. Molka, D. Redlich, M. Drobek, X.-J. Zeng, W. Gilani, Diversity guided evolutionary mining of hierarchical process models, in: GECCO'15, Association for Computing Machinery, New York, NY, USA, 2015, p. 1247–1254.

[17] S.-Y. Jing, Set-based differential evolution algorithm based on guided local exploration for automated process discovery, Complexity 2020 (2020).

[18] B. Vázquez-Barreiros, M. Mucientes, M. Lama, Prodigen: Mining complete, precise and minimal structure process models with a genetic algorithm, Information Sciences 294 (2015) 315–333.

[19] M. Fantinato, S. M. Peres, H. A. Reijers, X-processes: Process model discovery with the best balance among fitness, precision, simplicity, and generalization through a genetic algorithm, Inf. Syst. 119 (2023).

[20] S. J. van Zelst, S. J. J. Leemans, Translating workflow nets to process trees: An algorithmic approach, Algorithms 13 (2020). URL: https://www.mdpi.com/1999-4893/13/11/279. doi:10.3390/a13110279.

[21] A. Berti, S. van Zelst, D. Schuster, PM4Py: A process mining library for Python, Software Impacts 17 (2023) 100556. URL: https://www.sciencedirect.com/science/article/pii/S2665963823000933. doi:https://doi.org/10.1016/j.simpa.2023.100556.

[22] J. Snoek, H. Larochelle, R. P. Adams, Practical bayesian optimization of machine learning algorithms, in: Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, NIPS'12, Curran Associates Inc., Red Hook, NY, USA, 2012, p. 2951–2959.

[23] A. Berti, W. M. 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.

[24] J. Muñoz-Gama, J. Carmona, A fresh look at precision in process conformance, in: R. Hull, J. Mendling, S. Tai (Eds.), Business Process Management. BPM 2010, volume 6336 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 211–226. URL: https://doi.org/10.1007/978-3-642-15618-2_16. doi:10.1007/978-3-642-15618-2_16.

[25] J. C. A. M. Buijs, B. F. van Dongen, W. M. 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.

[26] B. L. Miller, D. E. Goldberg, Genetic algorithms, tournament selection, and the effects of noise, Complex systems 9 (1995) 193–212.

[27] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2019.

[28] Task Force on Process Mining, Event logs - task force on process mining, n.d. URL: https://www.tf-pm.org/resources/logs, accessed: 2025-05-22.

[29] W. 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.

[30] B. 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.

[31] F. 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.

[32] Genetic tree miner, https://github.com/jaxels20/genetic-tree-miner, 2025. Accessed: 2025-05-20.

[33] B. 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.

[34] B. 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.

[35] W. 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.

[36] B. 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.

[37] M. M. de Leoni, F. 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.

[38] B. 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.

[39] B. 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.

[40] W. 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.

[41] B. 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.

[42] B. 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.