universität wien

BACHELORARBEIT / BACHELOR’S THESIS

Titel der Bachelorarbeit / Title of the Bachelor’s Thesis

„Enhancement of the Process Mining Visualization Tool: Merge Projects, add Filtering, UI-Enhancements and Genetic Miner Implementation “

verfasst von / submitted by

David Kapfhammer

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science (BSc)

Wien, 2025 / Vienna, 2025

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 033 526
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Bachelorstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Ing. Dipl.-Ing. Dr.techn. Marian Lux

Abstract

Process mining enables the discovery of process models from event logs, offering organizations fact-based insights into how processes are executed in reality. While commercial tools such as *Disco* and *Celonis* provide advanced features, they are closed-source and limited for academic use. Open frameworks like *ProM* offer flexibility but remain difficult to use for non-experts. To address this gap, the *Process Mining Visualization Tool* was developed as an open, extensible, and user-friendly platform.

This thesis advances the tool in three key directions. First, previously fragmented implementations were merged into a unified architecture, consolidating Alpha, Heuristic, Fuzzy, and Inductive Miner algorithms together with global filtering capabilities based on the Search Process Model (SPM) metric. Second, the tool was enhanced with interactive features such as synchronized threshold sliders, node and edge filtering, clustering fixes, and click-based edge exploration, improving usability and interpretability of results. Third, and most importantly, the Genetic Miner was implemented and integrated into the Streamlit-based interface. As a global search approach based on genetic algorithms, it introduces robustness against noise and incompleteness by evolving populations of candidate models under a token-game fitness evaluation.

Evaluation confirmed functional alignment with the *ProM 5.2* reference implementation on simple logs, while also revealing current limitations: the absence of silent transitions, incomplete completion checks in the token-game logic, and the lack of large-scale benchmarks. Nevertheless, theoretical sensitivity analyses of parameters such as population size, crossover and mutation rates, and tournament selection provide a solid foundation for future experimentation.

In summary, this thesis strengthens the Process Mining Visualization Tool as a modular and extensible open-source platform, bridging academic accessibility with advanced discovery techniques. The integration of the Genetic Miner demonstrates the potential of evolutionary search in process discovery and lays the groundwork for further improvements, including enhanced fitness metrics, support for silent transitions, and large-scale empirical validation.

Contents

1 Motivation

In today's dynamic and data-driven world, organizations rely on increasingly complex processes to deliver their products and services. These processes often involve numerous activities, systems, and stakeholders, which makes them difficult to understand, optimize, and control. Process mining has emerged as a discipline that directly addresses this challenge by reconstructing process models from event logs and revealing how processes are executed in reality. By combining data-driven insights with graphical representations, it becomes possible to detect inefficiencies, uncover deviations, and guide targeted improvements. This ability to translate raw data into actionable knowledge is what makes process mining an indispensable technique in both academia and industry.

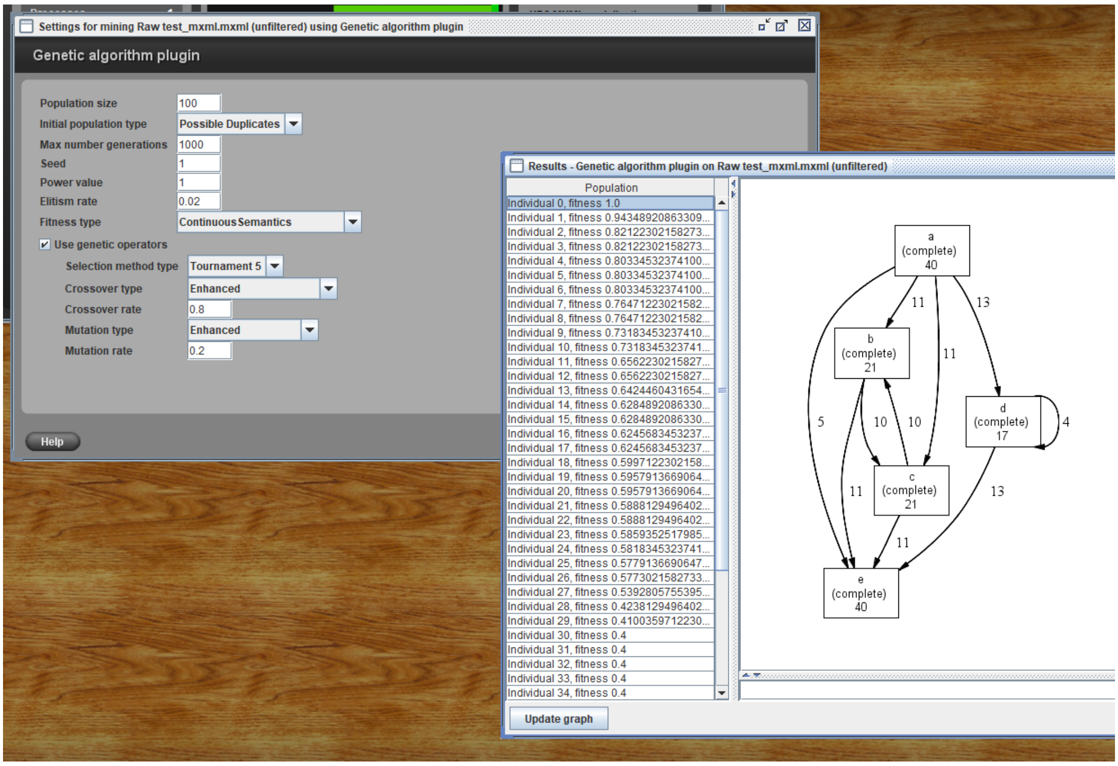
1.1 Problem Statement & Research Gap

Although the potential of process mining is widely acknowledged, the availability of accessible and effective tools is still limited. Commercial solutions such as *Disco* [5] or *Celonis* [3] provide powerful features but are proprietary and therefore less suitable for academic work, teaching, or community-driven research. On the other hand, open-source frameworks like *ProM* [17] offer a large set of algorithms and flexibility, but they are technically demanding and difficult to use for non-experts.

The *Process Mining Visualization Tool* was developed to bridge this gap by offering an open, user friendly, and extensible environment. However, the current state of the tool was fragmented across two separate projects: one version implemented the Alpha Miner together with the SPM filter in the legacy desktop UI-version [18], while another version provided a modern Streamlit-based interface and added the Inductive Miner [6]. So as a first step, these projects need to be merged into a single platform.

Even after merging, several limitations remain. The tool still lacks proper filters for nodes, edges, and the SPM metric [13], and edges are not yet clickable for interactive analysis. In addition, clustering is affected by a bug in the Heuristic Miner and not yet implemented in the Fuzzy Miner. Finally, the discovery algorithms already available — Alpha Miner, Heuristic Miner, Fuzzy Miner, and Inductive Miner — are known to be sensitive to noise and incompleteness in event logs, which often prevents them from producing robust and reliable process models [20].

The Genetic Miner is not a new concept, as it has already been implemented as a plugin in *ProM 5.2*. However, this implementation is outdated and difficult to use in practice. To apply the plugin to standard event logs such as CSV or XES, users must first import the data into a newer version of ProM (e.g., ProM 6.14), export it in the older MXML format, and then re-import it into ProM 5.2 to finally run the miner. This multi-step workflow makes the algorithm impractical for everyday use and discourages broader adoption. The plugin itself provides a wide range of configuration options, such as population size, crossover and mutation rates, or selection methods, and it can produce process models of reasonable quality. Nevertheless, the reliance on outdated file formats and cumbersome preprocessing steps significantly limits its accessibility for practitioners and students. Figure 1.1 illustrates the original interface of the Genetic Miner in *ProM 5.2* together with an example of a mined process model.



Settings for mining Raw test_mxml.mxml (unfiltered) using Genetic algorithm plugin

Genetic algorithm plugin

Population size: 100
Initial population type: Possible Duplicates
Max number generations: 1000
Seed: 1
Power value: 1
Elitism rate: 0.02
Fitness type: ContinuousSemantics
☑ Use genetic operators
Selection method type: Tournament 5
Crossover type: Enhanced
Crossover rate: 0.8
Mutation type: Enhanced
Mutation rate: 0.2

Results - Genetic algorithm plugin on Raw test_mxml.mxml (unfiltered)

Population
Individual 0, fitness 1.0
Individual 1, fitness 0.94348920863309...
Individual 2, fitness 0.82122302158273...
Individual 3, fitness 0.82122302158273...
Individual 4, fitness 0.80334532374100...
Individual 5, fitness 0.80334532374100...
Individual 6, fitness 0.80334532374100...
Individual 7, fitness 0.76471223021582...
Individual 8, fitness 0.76471223021582...
Individual 9, fitness 0.73183453237410...
Individual 10, fitness 0.7318345323741...
Individual 11, fitness 0.6562230215827...
Individual 12, fitness 0.6562230215827...
Individual 13, fitness 0.6424460431654...
Individual 14, fitness 0.6284892086330...
Individual 15, fitness 0.6284892086330...
Individual 16, fitness 0.6245683453237...
Individual 17, fitness 0.6245683453237...
Individual 18, fitness 0.5997122302158...
Individual 19, fitness 0.5957913669064...
Individual 20, fitness 0.5957913669064...
Individual 21, fitness 0.5888129496402...
Individual 22, fitness 0.5888129496402...
Individual 23, fitness 0.5859352517985...
Individual 24, fitness 0.5818345323741...
Individual 25, fitness 0.5779136690647...
Individual 26, fitness 0.5773021582733...
Individual 27, fitness 0.5392805755395...
Individual 28, fitness 0.4238129496402...
Individual 29, fitness 0.4100359712230...
Individual 30, fitness 0.4
Individual 31, fitness 0.4
Individual 32, fitness 0.4
Individual 33, fitness 0.4
Individual 34, fitness 0.4

Figure 1.1: The Genetic Miner plugin in ProM 5.2.

The motivation of this thesis is therefore twofold. First, it strengthens the tool by resolving existing issues and improving its technical foundation, making the tool more extensible for future use. Second, it integrates the Genetic Miner into the tool as a new capability for robust process discovery. While the implementation is nearly complete, some challenges remain, such as the need for a proper completion check in the token game logic and the integration of silent transitions. Even so, the addition of the Genetic Miner significantly expands the scope of the tool and provides a solid foundation for future work.

2 Related Work & Algorithms

Process mining bridges event logs and process models, enabling data-driven discovery of workflow structures. The following sections outline the theoretical foundations used in the tool and review the main discovery algorithms that have been implemented or extended with filters in this project. Special emphasis is given to the Genetic Miner, which was implemented and adapted to the project architecture.

2.1 Foundations

Process mining in Process Mining Visualization Tool The tool reads event logs (traces with frequencies), applies filtering to abstract noise and infrequent behavior, and runs a chosen discovery algorithm to derive a process model. Visualization is handled in a modular way: each discovery algorithm has its own dedicated graph class (extending a common `BaseGraph`) built on *Graphviz*. This ensures that the rendering matches the semantics of the underlying algorithm. For instance, Alpha and Genetic Miner use Petri-net–like graphs with explicit places, Heuristic Miner produces dependency graphs with weighted edges, Inductive Miner visualizes block-structured process trees including silent activities [11], and Fuzzy Miner generates clustered maps based on significance and correlation metrics. All miners inherit from the shared `BaseMining` class, which provides global filtering functionality. In particular, both SPM-based filtering and node frequency filtering are applied before discovery, and the absolute and normalized thresholds for these filters are synchronized to ensure consistent interaction in the UI. Frequencies are calculated over all traces in the event log. A log L is a multiset of traces σ, where each trace is a sequence of activities and has an associated frequency $f(\sigma)$. For an activity a, its absolute frequency is the weighted count of all its occurrences across the log:

$$\mathrm{freq}(a) = \sum_{\sigma \in L} \mathrm{count}_\sigma(a) \cdot f(\sigma),$$

where $\mathrm{count}_\sigma(a)$ is the number of times a appears in trace σ. Normalized frequencies are obtained by dividing through the maximum frequency of any activity:

$$\mathrm{freq}_{norm}(a) = \frac{\mathrm{freq}(a)}{\max_{b \in A} \mathrm{freq}(b)}.$$

Earlier project iterations contributed Alpha, Heuristic, Inductive, and Fuzzy Miner implementations plus the SPM filter and UI groundwork, which we merged and extended in this thesis [18, 6].

SPM Filtering (Search Process Model) The *Search Process Model (SPM)* quality metric was introduced to evaluate and filter event logs and models where user behavior is highly variable, such as search processes [13]. Its goal is to reduce complexity and highlight common, representative behavior, making models more interpretable and less overloaded with rare or noisy paths. This abstraction is particularly important for exploratory domains, where traditional miners otherwise produce spaghetti-like structures.

For a node $n \in N$, the metric is defined as

$$\text{spm}(n) = 1 - \frac{\deg(n) \cdot |A|}{\text{freq}(n) \cdot |L|},$$

where $\deg(n)$ is the (in+out) degree of n in the succession matrix, $|A|$ is the number of distinct activities in the log, $\text{freq}(n)$ the absolute frequency of n, and $|L|$ the total number of events in the log. The value is bounded between 0 and 1, with higher values indicating nodes that are both frequent and structurally central. Intuitively, the formula penalizes nodes with many incoming and outgoing connections (high degree), which typically add complexity, while rewarding those that occur frequently in the log, which are more representative of common behavior.

Nodes with SPM values below the chosen threshold are removed from the model before discovery. This ensures that only semantically strong and frequent activities are retained, leading to models that balance structural simplicity with behavioral relevance [13, 18]. This filtering is implemented in the shared `BaseMining` class, making it globally available across all discovery algorithms.

2.2 Related Work

2.2.1 Alpha Miner

The *Alpha Miner* was the first widely adopted discovery algorithm in process mining. It constructs a Petri net from event logs by systematically detecting causal, parallel, and choice relations between activities. The algorithm assumes that the log is complete and noise-free, making it particularly suited for structured processes, though less robust for real-world data that contain deviations or short loops [1] [2].

Core Relations The algorithm starts by extracting the set of unique events from the log, and then identifies the set of start and end events. From the log's traces, it computes the following relations:

- **Direct succession** ($>$): $a > b$ if activity a is directly followed by activity b in some trace.
- **Causality** ($\rightarrow$): $a \rightarrow b$ if $a > b$ and not $b > a$.
- **Parallelism** ($\|$): $a \| b$ if both $a > b$ and $b > a$ hold.
- **Choice** ($\#$): $a \# b$ if neither causality nor parallelism holds between a and b.

These relations are recorded in the so-called *footprint matrix*, which forms the basis of the model.

Set Generation To capture splits and joins, the Alpha Miner searches for pairs of activity sets (A, B) such that each $a \in A$ causally leads to each $b \in B$, and activities within A and within B are pairwise independent. The initial set XL is reduced to a maximal set YL by removing non-maximal pairs and self-loop artifacts. This ensures that only the most expressive causal relations are retained.

Petri Net Construction Using YL, the algorithm creates places in the Petri net that connect transitions corresponding to activities in A to those in B. Start and end places are added and linked to the identified start and end activities. The resulting Petri net is a workflow net that, under ideal conditions, perfectly reproduces the observed behavior.

Implementation in Process Mining Visualization Tool The implementation follows closely the steps introduced by van der Aalst [2] and Rustemi [18]. It computes the footprint relations from filtered event logs and then constructs the Petri-net–like graph:

1. Compute direct succession, causality, parallelism, and choice relations.
2. Generate candidate set XL and reduce it to YL by removing subsets and self-loop inconsistencies.
3. Create nodes for all activities (with associated frequencies and SPM scores).
4. For each $(A, B) \in YL$, add a connector place and link activities accordingly, supporting single–single, single–multiple, and multiple–multiple cases.
5. Ensure start and end places are properly connected, even if no transitions were discovered.

It also addresses edge cases: if no causal relations exist, each start activity is connected directly to the end, ensuring a valid minimal model.

Limitations and Filtering Like the theoretical algorithm, the implemented Alpha Miner is sensitive to noise, incompleteness, and short loops. To improve interpretability, the *SPM filter* [13] and node frequency thresholds are integrated. These filters abstract away infrequent or low-quality nodes and edges, reducing complexity while keeping the most representative behavior visible to the user. As a result, the discovered models are more concise and readable, especially when dealing with real-life event logs.

2.2.2 Heuristic Miner

The *Heuristic Miner* improves on the Alpha Miner's strictness by introducing frequency-based dependency measures that make it more robust against noise and incomplete logs [2]. Instead of assuming that every directly-follows relation is significant, it focuses on the most frequent and reliable relations in the log.

Dependency Metric For every pair of activities, a dependency score is calculated as:

$$\operatorname{dep}(a,b) = \begin{cases} \dfrac{f(a,b) - f(b,a)}{f(a,b) + f(b,a) + 1}, & \text{if } a \neq b, \\ \dfrac{f(a,a)}{f(a,a) + 1}, & \text{if } a = b, \end{cases}$$

where $f(a,b)$ is the number of times a is directly followed by b in the log. This captures how strongly a depends on b, discounting symmetrical or weak relations. Self-dependency captures **length-one loops**, activities that directly follow themselves.

Graph Construction The activities are added as nodes, annotated with their SPM value, normalized frequency, and absolute frequency. Edges between activities are drawn only if they satisfy the dependency and frequency thresholds, and are annotated with frequency values (absolute and normalized) and dependency scores. This yields a directly-follows graph that highlights dominant behavior while suppressing noise and outliers.

Filtering and Metrics In the implementation, the Heuristic Miner uses additional filtering to improve interpretability on noisy or incomplete logs. The SPM filter [13] removes nodes of low semantic quality by balancing frequency and structural connectivity. Node and edge frequency thresholds, defined both in absolute and normalized terms, discard infrequent activities and transitions while keeping the values synchronized for consistency. Finally, a dependency threshold ensures that only strong causal relations remain. Together, these filters ensure that only well-supported and relevant relations remain in the model. [4, 18].

2.2.3 Fuzzy Miner

The *Fuzzy Miner* was designed for highly unstructured processes, where enforcing exact control-flow semantics would result in overly complex models [9]. Instead of aiming for sound workflow nets, it uses significance and correlation metrics to abstract the process into a more readable structure. Based on these measures, it applies three rules: significant nodes are retained, less significant but strongly correlated nodes are clustered, and nodes with both low significance and low correlation are removed. In this way, the algorithm highlights dominant behavior while avoiding distracting details.

Metrics The Fuzzy Miner relies on several local metrics to determine which nodes and edges should be kept, clustered, or removed [10].

Unary significance measures how central an activity is within the process. It is calculated as:

$$unary\_sig(a) = \frac{freq(a)}{\max_{x \in A} freq(x)},$$

where $freq(a)$ is the frequency of activity a in the log. This value lies between 0.0 and 1.0 and expresses how prominent a node is relative to the most frequent activity. Numerically, this measure is identical to the normalized node frequency, but while node frequency is mainly used for filtering, unary significance directly influences clustering: nodes with low unary significance may be clustered with neighbors if they remain strongly connected.

Binary significance (also referred to as edge significance) assesses the importance of a transition between two activities. It combines the frequency of the source activity with the frequency of the edge itself. In practice, it is computed from the significance matrix as

$$bin\_sig(a,b) = \frac{freq(a \to b)}{\max_{x \in A} freq(x)},$$

which yields values between 0.0 and 1.0. This measure expresses how relevant a transition is in relation to the most frequent activity in the log. Edges with binary significance below a user-defined threshold are ignored in subsequent utility calculations.

Correlation quantifies how strongly two activities are related. It is computed as:

$$corr(a,b) = \frac{freq(a \to b)}{\sum_{x \in A} freq(a \to x)},$$

the proportion of times a is followed by b relative to all direct successors of a. This measure ranges from 0.0 (no correlation) to 1.0 (perfect correlation).

Utility ratio defines the balance between significance and correlation when evaluating an edge. A ratio of 1.0 means that only significance is considered, while 0.0 means that only correlation is considered.

Utility value is the combined score of significance and correlation, defined as:

$$util(a,b) = ur \cdot sig(a,b) + (1 - ur) \cdot corr(a,b),$$

where $sig(a,b)$ is the binary significance of the edge and ur is the chosen utility ratio.

Edge cutoff then applies Min–Max normalization to all incoming edges of a node and removes those with normalized utility below the cutoff:

$$util_{norm}(a,b) = \frac{util(a,b) - \min(util)}{\max(util) - \min(util)}.$$

Based on these measures, the abstraction follows three rules:

1. Nodes with high unary significance are always retained,

2. Less significant but strongly correlated nodes are clustered, and

3. Weak and insignificant nodes are removed entirely.

This adaptive simplification emphasizes dominant behavior while suppressing noise, producing a process model that resembles a “roadmap” rather than an exhaustive map of all possible paths. [10, 9]

Implementation in Process Mining Visualization Tool In the tool, the Fuzzy Miner extends this principle with multiple filtering options. Alongside the core metrics of unary and binary significance, correlation, utility ratio, and edge cutoff, the implementation also integrates SPM filtering as well as node and edge frequency thresholds. The SPM filter [13] removes low-quality nodes by balancing frequency with structural connectivity, while frequency thresholds discard infrequent activities and transitions. All sliders can be adjusted interactively, enabling the user to emphasize dominant paths, merge secondary behavior into clusters, or prune edges below a desired relevance.

Filtering and Limitations Unlike procedural miners, the Fuzzy Miner does not guarantee sound Petri-nets, but instead aims for models that are faithful abstractions of noisy and weakly structured logs. In practice, this means that accuracy in terms of formal semantics is traded for clarity of representation. By combining the *SPM-filter*, frequency-based thresholds, and fuzzy significance–correlation metrics, the algorithm incrementally suppresses low-value behavior while preserving dominant paths. This adaptive filtering prevents the generation of "spaghetti-like" diagrams and instead produces simplified and interpretable models that emphasize the main process structure. As a result, the discovered models are particularly well suited for exploratory analysis and teaching scenarios, where clarity and adaptability are prioritized over strict completeness. [10, 9]

2.2.4 Inductive Miner

The *Inductive Miner* constructs block-structured process trees by recursively detecting cuts in the event log. Unlike Alpha or Heuristic Miner, which directly connect activities based on relations, Inductive Miner ensures that the discovered models are sound and block-structured. The algorithm repeatedly checks whether the log matches a *base case*, attempts to find a valid *cut*, and otherwise applies a *fallthrough* strategy. Each recursion yields a subprocess tree, and the trees are combined into a complete process model. [6, 11]

Base Cases Two base cases stop the recursion: if the log contains only an empty trace, the miner returns a silent activity, and if it contains a single trace with exactly one activity, that activity is returned as a leaf node. These base cases represent the smallest process trees.

Cut Detection and Log Splitting If the log is not a base case, the miner tries to detect a cut in the directly-follows graph. Cuts partition the log into sublogs and correspond to process tree operators:

- **Exclusive choice (*xor*):** traces are assigned to exactly one partition (*exclusive split*).
- **Sequence (*seq*):** traces are split into ordered segments that follow one another (*sequence split*).

- **Parallelism (*par*):** traces are projected onto partitions that can be executed concurrently (*parallel split*).
- **Loop (*loop*):** one partition forms the "do" part, others form "redo" parts that can repeat (*loop split*).

If a valid cut is found, the miner recurses on each sublog and combines the results under the corresponding operator. By construction, only valid partitions are accepted, which guarantees that the resulting process tree is sound and block-structured.

Fallthroughs If no cut can be found, the miner applies fallthrough behavior. If the log contains an empty trace, an exclusive choice between a silent activity and the remainder of the log is returned. If the log contains only one activity repeated multiple times, a loop between that activity and a silent activity is created. Otherwise, the miner constructs a *flower model*, represented as a loop that allows any order of the remaining activities. These fallthroughs guarantee termination and rediscoverability for all logs.

Implementation in Process Mining Visualization Tool The implementation closely follows the theoretical algorithm, but introduces frequency-based filtering before cut detection to improve readability and cut quality. Infrequent events are removed using *SPM filtering* and node frequency thresholds, and infrequent traces are removed using a configurable *trace threshold*. The trace threshold is defined as:

$$tr(trace) = \frac{freq(trace)}{\max_{t \in L} freq(t)},$$

and removes all traces below a chosen ratio of the maximum trace frequency. The recursive procedure then applies base cases, cut detection, and fallthroughs as described above, returning a process tree. The tree is visualized as a graph: events are shown as boxes with frequency and SPM values, operators are rendered as gateways (exclusive, parallel, loop), and silent activities are displayed as small point nodes. This ensures that even logs with noise or outliers are reduced to clear block-structured models. [20, 11, 6]

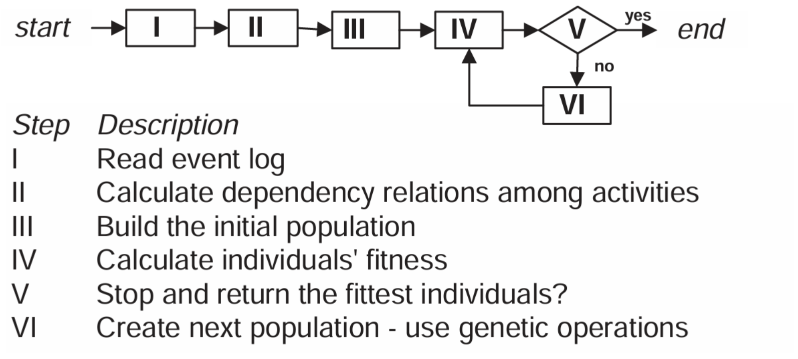
Filtering and Limitations In contrast to the other discovery algorithms, the Inductive Miner guarantees soundness and block-structuredness of the discovered models by design. Specifically, the *SPM filter* [13] removes semantically weak nodes, node frequency thresholds discard rare activities, and a configurable trace threshold eliminates infrequent traces. These preprocessing steps suppress outliers and noise, allowing the algorithm to identify cleaner directly-follows relations and more meaningful cuts during recursion. As a consequence, the resulting models remain rediscoverable and theoretically correct with respect to the filtered log, while at the same time being more concise, readable, and practically usable in real-world settings.

2.3 Genetic Miner

The *Genetic Miner* is a global process discovery algorithm based on genetic algorithms (GAs). Unlike local miners, which rely on directly-follows statistics, it searches over the space of possible process models. Candidate models (individuals) are evolved using selection, crossover, and mutation, guided by a fitness function that measures how well the model parses the log. This global evaluation approach makes the Genetic Miner particularly effective for logs that are noisy, incomplete, or weakly structured. In contrast to local miners, whose dependency on directly-follows relations can be easily distorted by noise or missing data, the Genetic Miner evaluates candidate models against the log as a whole, searching for structures that optimize both activity parsing and trace completion. This robustness to incomplete or inconsistent data comes at the cost of higher computational effort, but enables the discovery of more reliable models in challenging real-world scenarios. [14].

2.3.1 Base Flow

Figure 2.1 illustrates the main flow of the algorithm. 1) Starting from the event log, 2) dependency relations are computed, 3) an initial population of candidate models is built, and 4) fitness is evaluated. 5) If the stopping criterion is not met, 6) genetic operations are applied to produce the next population. The loop continues until a sufficiently fit model is found or threshold limits are reached.



Step	Description
I	Read event log
II	Calculate dependency relations among activities
III	Build the initial population
IV	Calculate individuals' fitness
V	Stop and return the fittest individuals?
VI	Create next population - use genetic operations

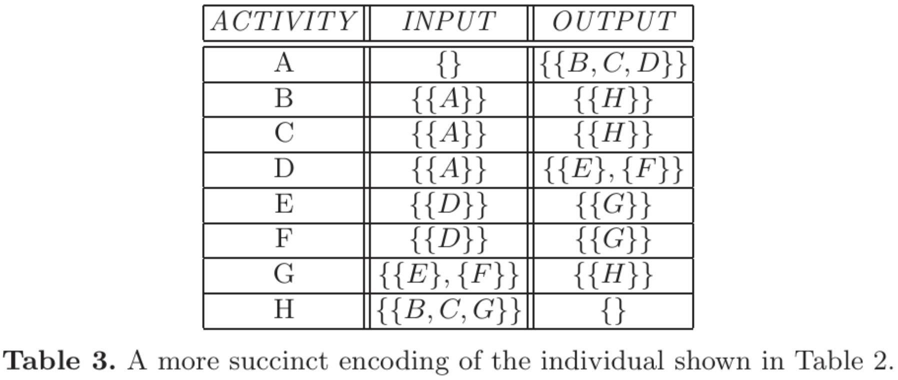
Fig. 4. Main steps of our genetic algorithm.

Figure 2.1: Main steps of the Genetic Algorithm. [14]

2.3.2 Representation (Individuals)

Each individual encodes a candidate model as a *causal matrix* $CM = (A, C, I, O)$:

- A: the list of activities,
- $I(a)$: a dictionary mapping each activity a to a list of input sets,
- $O(a)$: a dictionary mapping each activity a to a list of output sets,
- $C = \{(a, b) \mid b \in O(a)\}$: the set of causal relations represented as tuples.



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

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

Figure 2.2: Representation of an individual with I and O sets, adopted from [14]

2.3.3 Dependency Measures

Before initialization, the algorithm computes directly-follows and loop statistics from the log:

$$\begin{aligned} &follows(a, b) = \#(a \to b), \\ &L1L(a) = \#(a \to a), \\ &L2L(a, b) = \#(a \to b \to a). \end{aligned}$$

From these, the dependency measure is defined as:

$$D(a, b) = \begin{cases} \dfrac{L2L(a, b) + L2L(b, a)}{L2L(a, b) + L2L(b, a) + 1}, & a \neq b,\ L2L(a, b) > 0 \\ \dfrac{follows(a, b) - follows(b, a)}{follows(a, b) + follows(b, a) + 1}, & a \neq b,\ L2L(a, b) = 0 \\ \dfrac{L1L(a)}{L1L(a) + 1}, & a = b. \end{cases}$$

Additionally, start and end measures are calculated:

$$S(t) = D(start, t), \quad E(t) = D(t, end).$$

2.3.4 Heuristic Initialization

Initial individuals are created probabilistically in several steps:

1. Keep each causal relation (a, b) with probability $D(a, b)^p$, where p is an odd integer (power parameter).
2. Apply *start wipe*: remove all incoming edges of a with probability $S(a)^p$.
3. Apply *end wipe*: remove all outgoing edges of a with probability $E(a)^p$.
4. Partition predecessors of each activity to form input sets $I(\cdot)$.
5. Partition successors to form output sets $O(\cdot)$.
6. Derive causal relations C from $O(\cdot)$.

The power parameter controls sparsity: larger values yield fewer initial edges.

2.3.5 Fitness and Token Game

Fitness is evaluated using a token-game simulation:

- An activity fires if all its input sets are satisfied (AND across sets) and each set has at least one provider with a token (OR within sets).
- Tokens are consumed from inputs and produced in outputs.
- Self-loops are explicitly supported.
- A trace is *parsed* when all activities fire successfully.
- A trace is *completed* if no tokens remain at the end.

The continuous fitness measure is:

$$F(PM, L) = 0.40 \cdot \frac{\text{parsed activities}}{\text{total activities}} + 0.60 \cdot \frac{\text{completed traces}}{\text{total traces}}, \quad F \in [0, 1].$$

2.3.6 Genetic Operations

Elitism

The best $elitism\_rate \times population\_size$ individuals are copied unchanged into the next generation. This guarantees that the current best solution is preserved.

Selection

Tournament selection is used: randomly sample k individuals and select the fittest as a parent. This balances exploration and exploitation by giving weaker individuals a small chance of being selected.

Crossover

For crossover, a random activity t is chosen, and its input/output sets are partially swapped between two parents:

1. Select crossover points in $I(t)$ and $O(t)$ of both parents.
2. Exchange suffixes to create new partitions.
3. Resolve overlaps by merging or splitting subsets.
4. Update related activities: ensure consistency such that $b \in O(a) \Leftrightarrow a \in I(b)$.

This operator recombines building blocks of process models while maintaining structural validity. Figure 2.3 illustrates this procedure, showing how two parent individuals generate two consistent offspring.

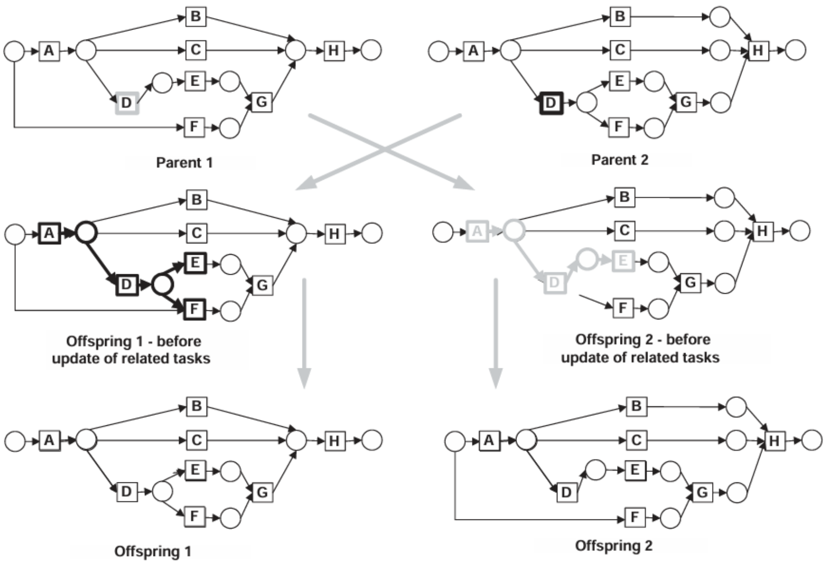

Parent 1
Parent 2
Offspring 1 - before update of related tasks
Offspring 2 - before update of related tasks
Offspring 1
Offspring 2
A
B
C
D
E
F
G
H

Fig. 6. Example of the crossover operation for the two individuals in Figure 5. The crossover point is activity D.

Figure 2.3: Example of the crossover operation. [14]

Mutation

Each activity may undergo mutation with probability *mutation_rate*. Mutation operates by:

- **Merge:** two input (or output) subsets are merged into one.

- **Split:** a non-trivial subset is split into two smaller subsets.

After mutation, a consistency check updates all related activities to repair I/O mismatches. This ensures that causal relations remain symmetric and the model valid.

Termination

The algorithm stops when:

1. An individual reaches the fitness threshold (default 1.0),
2. The generation count is reached, or
3. Fitness stagnates for half the generations.

2.3.7 From Individual to Petri Net

After termination, the best individual is transformed into a Petri-net–like graph:

- Start and End nodes are connected through dedicated places.
- Each input set generates an intermediate place connecting its predecessors to the activity.
- Each output set generates a place connecting the activity to its successors.
- Self-loops are handled by introducing a dedicated place that connects the activity to itself.
- Redundant singletons already covered by larger sets are skipped to avoid duplication.

Silent transitions In the original Genetic Miner design [14], *silent transitions* are introduced when translating causal sets into a Petri net. These connectors have no corresponding activity in the log but are essential to preserve correct semantics for:

- **Choices:** where a place enables multiple alternative successors,
- **Parallel splits and joins:** where execution branches or synchronizes,
- **Loops:** where an activity can repeat without an explicit event in between.

Silent transitions ensure that routing constructs are modeled precisely without introducing artificial edges.

In the current implementation, however, silent transitions are *not yet included*: routing is approximated using intermediate places only. While this yields a simplified graph that is easier to render, it can result in models that are structurally less accurate. Adding proper silent transitions remains an open improvement for faithfully representing the semantics described in the original algorithm.

2.3.8 Parameters

The implementation exposes all key parameters as interactive sliders:

- **Genetic sliders:** Population Size, Max Generations, Crossover Rate, Mutation Rate, Elitism Rate, Tournament Size, Power Value, Fitness Threshold.
- **Filtering sliders:** SPM threshold, Node frequency threshold.

This design makes the search behavior transparent to users and allows fine-tuning of exploration, exploitation, and abstraction levels.

Pseudocode: Default Genetic Algorithm

```
INPUT:
    TARGET string
    POPULATION_SIZE
    GENES (alphabet)

INITIALIZATION:
    population = random Strings (length = TARGET)

EVOLUTION LOOP:
    WHILE best individual has fitness > 0:
        1. COMPUTE fitness of each individual
           Fitness = Count of wrong symbols
        2. SORT population after fitness (beste first)
        3. ELITISM: copy the best 10 % unchanged
        4. REPRODUCTION:
           - Pick parents random from the best 50 %
           - Crossover: take over genes from Parent1 / Parent2
           - Mutation: with probability 10 % replace a random
   symbol
        5. Population = new generation

OUTPUT:
    best individual (exact string)
```

Listing 2.1: Default Genetic Algorithm

Pseudocode: Genetic Miner

```
INPUT:
    EVENT LOG (Traces + Frequencies)
    GA-Parameters:
       population_size, max_generations,
       crossover_rate, mutation_rate,
       elitism_rate, tournament_size,
       power_value, fitness_threshold

INITIALIZATION:
    1. COMPUTE dependency-matrix (follows, L1L, L2L)
    2. COMPUTE S(t) = D(start,t), E(t) = D(t,end)
    3. CREATE heuristic individuals:
       - representation: activities,
         INPUT-Sets, OUTPUT-Sets, causal Relations
       - power_value controls sparsity of edges

EVOLUTION LOOP:
    WHILE (best fitness < fitness_threshold)
          AND (generation_count < max_generations)
          AND (no stagnation reached):
        1. COMPUTE fitness of all individuals
           fitness = 0.40 * (parsed events / total events)
                   + 0.60 * (completed traces / total traces)
        2. SORT population after fitness (best first)
        3. ELITISM: copy best (elitism_rate * pop_size) unchanged
        4. REPRODUCTION:
           - Parent selection with tournament (size = k)
           - Crossover: change I/O-Sets of an activity
           - Mutation: split/merge I/O-Sets with prob.
   mutation_rate
           - Consistency check: b in O(a) <=> a in I(b)
        5. Population = new generation

OUTPUT:
    best process model (individual)
    -> translated into Petri net
```

Listing 2.2: Genetic Miner

2.3.9 Comparison of Default GA and Genetic Miner

The pseudocodes (Listings 2.1 and 2.2) show the key differences. While the Default GA evolves strings of characters, Genetic Miner adapts GA principles to complex process models:

- Representation: simple strings vs. Petri-net–like causal structures.
- Initialization: random vs. dependency-based heuristic initialization.
- Fitness: Hamming distance vs. token-game parsing with trace completion.
- Operators: character-wise swap/mutation vs. structural changes in I/O sets.
- Termination: reaching target string vs. achieving process-model fitness.

This illustrates how Genetic Miner generalizes the GA framework from trivial strings to realistic process discovery tasks.

3 Implementation

This chapter documents the technical realization of the project. It first introduces the overall architecture of the tool, then outlines the project setup, implemented improvements, and the integration of the Genetic Miner. Finally, practical aspects of working with the project are discussed. The full source code of the implementation is openly available at: `https://github.com/kapfi02/ProcessMiningVisualization_Kapfhammer`.

3.1 Architecture

The Process Mining Visualization Tool follows a modular *Model–View–Controller (MVC)* architecture [8], which clearly separates data handling, algorithmic logic, and user interaction. This design was already applied in earlier contributions [6, 4, 18], and was maintained and extended for this thesis.

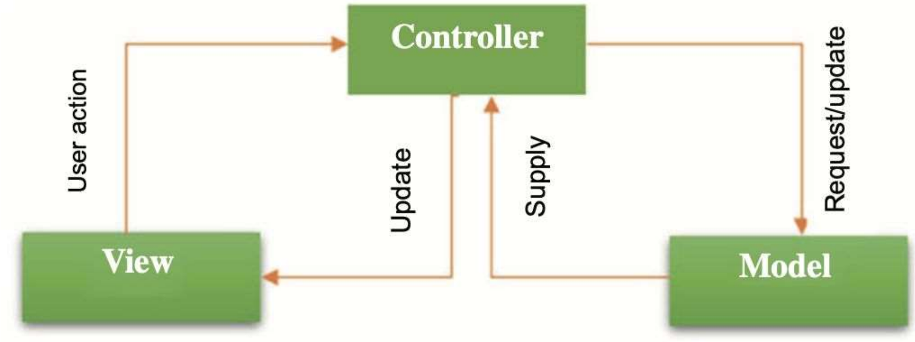


Figure 3.1: MVC architecture (adopted from [8, 6]).

- **Model** — Encapsulates the event log, filtering mechanisms, and process discovery algorithms. Each miner (e.g., `GeneticMining`, `InductiveMining`) extends the shared base class `BaseMining` [6], which centralizes essential functionality such as log preprocessing (SPM, node and edge filtering), succession matrix construction, frequency statistics, and threshold synchronization, alongside other core utilities.

- **View** — Implemented with `Streamlit` [19], providing an interactive user interface. It constructs process graphs using `Graphviz` [7] and renders them through interactive Streamlit components. The view also exposes sliders for filter thresholds, displays parameter controls for algorithm configuration, and supports user interaction through buttons and click handling on nodes and edges.

- **Controller** — Centralizes the interaction logic between model and view. It processes user inputs such as slider adjustments, button clicks, and session-state changes, updates algorithm and filter parameters, and triggers rediscovery of process graphs. It also manages error handling, ensuring smooth coordination between user actions and model execution in real time.

This strict separation increases maintainability and extensibility: new algorithms, filters, or UI features can be integrated with minimal changes to existing components. Figure 3.2 illustrates the class-level organization of the MVC pattern, highlighting how `BaseController` serves as the abstraction through which concrete controllers coordinate multiple `BaseView` and `BaseModel` instances.

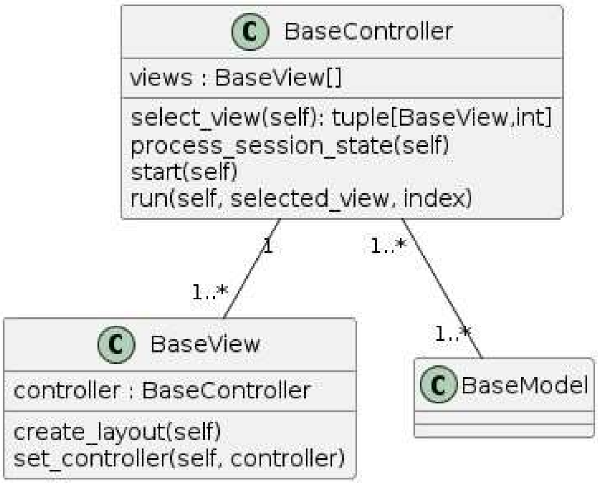

C
BaseController
views : BaseView[]
select_view(self): tuple[BaseView,int]
process_session_state(self)
start(self)
run(self, selected_view, index)
1
1..*
1..*
1..*
C
BaseView
controller : BaseController
create_layout(self)
set_controller(self, controller)
C
BaseModel

Figure 3.2: Simplified class diagram emphasizing the MVC architecture. [6]

3.2 Project Setup

The Process Mining Visualization Tool is implemented in Python and relies on well-established libraries, most prominently `Streamlit` [19], `Graphviz` [7], and `NumPy` [16]. Additional dependencies are listed in the attached `requirements.txt` file, which can be installed with `pip`. The application requires Python ≥ 3.10 and can be started locally via:

```
pip install -r requirements.txt
streamlit run app.py
```

A more detailed description of the installation steps, configuration options, and usage examples is provided in the project's README file on GitHub[1].

[1] `https://github.com/kapfi02/ProcessMiningVisualization_Kapfhammer`

File Structure

The project is organized into modular packages that separate core functionalities and follow the principles of the *MVC* architecture. The central entry point is the `streamlit_app.py` file, which launches the Streamlit interface and orchestrates the interaction between user input, filtering, and discovery algorithms.

Basic configurations are maintained in `config.py`, while reusable logging utilities are defined in `logger.py`. The `ui` module contains all Streamlit pages, with each page separated into its own folder that typically holds a controller and at least one view. Shared interface elements are collected in the `components` module.

All process discovery algorithms are contained in the `mining_algorithms` package. Each algorithm (Alpha Miner, Heuristic Miner, Inductive Miner, Fuzzy Miner, and Genetic Miner) is implemented in its own class file, extending the abstract base class `BaseMining`. This ensures a uniform interface for applying filters, computing succession matrices, and other core utilities.

The `graphs` module contains all graph-related classes. These include both generic visualization utilities and algorithm-specific graph classes such as `GeneticGraph`, which is responsible for rendering the output of the Genetic Miner. Supporting transformations, such as conversion from event logs into succession or dependency matrices, are encapsulated in the `transformations` module.

To support advanced log analysis, the `clustering` module integrates the Density Distribution Cluster Algorithm (DDCAL) [12]. This module enables clustering of event frequencies and provides abstractions that are reused by different miners.

Unit tests are located in the `tests` directory and cover the essential components of the system, including initialization of dependency matrices, filtering behavior, and the correctness of crossover and mutation operators in the Genetic Miner. Templates for the user interface are stored in the `templates` directory, and all documentation files, including UML diagrams and prior project reports, are maintained in the `docs` directory. Presentations and algorithm-specific documentation are collected there for reference.

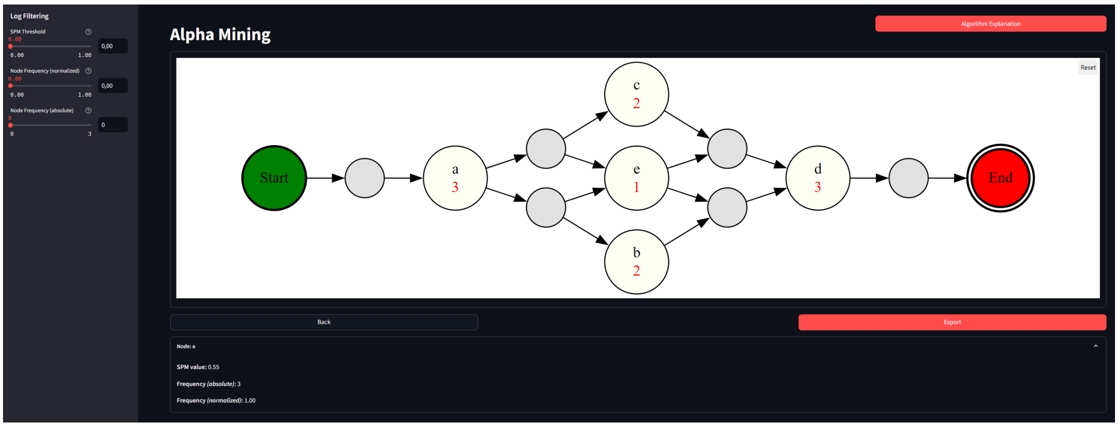
Together, this structure ensures a clean separation of concerns: user-facing pages are strictly separated from the underlying mining logic, graph rendering, and data transformations. This design supports extensibility, as new miners, filters, or visualization features can be integrated with minimal disruption to the existing codebase.

3.3 Implemented Improvements

During this thesis, the Process Mining Visualization Tool was extended and adapted in several ways. The work focused on consolidating existing features, extending filters, refining the user interface, and integrating the Genetic Miner as a new discovery algorithm. The following subsections highlight these improvements in more detail.

Merging of Projects and Alpha Miner Integration

The first major step was the consolidation of previously separate codebases into a unified repository. In this process, the Alpha Miner and its SPM filter were migrated from the legacy interface into the new Streamlit UI. By aligning these components with the shared `BaseMining` class, SPM filter can now be applied consistently across all algorithms. In addition, the `BaseAlgorithmController` and `BaseAlgorithmView` classes were extended to make the SPM filter globally available in the interface. This ensures that filter parameters are consistently exposed and processed in the same way for every miner. Figure 3.3 shows the Alpha Miner in the modernized Streamlit interface, demonstrating seamless interaction with sliders and updated visualization.



Log Filtering

SPM Threshold: 0.00 (0.00 – 1.00) 0,00

Node Frequency (normalized): 0.00 (0.00 – 1.00) 0,00

Node Frequency (absolute): 0 (0 – 3) 0

Node: a

SPM value: 0.55

Frequency *(absolute)*: 3

Frequency *(normalized)*: 1.00

Figure 3.3: Alpha Miner merged into the new Streamlit interface.

Extended Filtering

Filtering capabilities were extended and standardized across all miners:

- *SPM filter*: implemented globally to remove semantically weak events [13].
- *Node filtering*: both absolute and normalized frequencies are supported, with synchronized sliders to prevent inconsistencies.
- *Edge filtering*: introduced for Heuristic and Fuzzy Miner to reduce visual complexity, likewise supporting absolute and normalized frequencies with synchronized sliders.

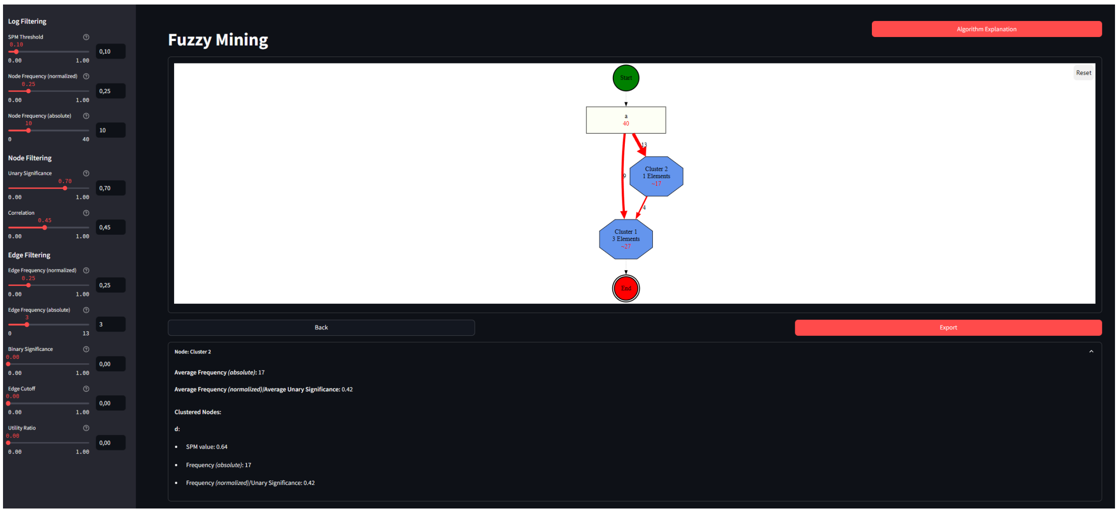
This ensures that models remain interpretable even with noisy or large logs. The synchronization logic was placed in `BaseMining` to work around Streamlit's limitations, where the session state is also used to keep normalized and absolute thresholds aligned.

UI-Enhancements

Several UI-related enhancements were realized:

- Absolute and normalized threshold sliders are synchronized.
- Edge-click handler was introduced to allow inspection of transition properties.
- Edge clustering was debugged for the Heuristic Miner and integrated into the Fuzzy Miner.
- Unary (node) and binary (edge) significance metrics were separated in the Fuzzy Miner, giving users better control over abstraction.
- Clustered edges in the Fuzzy Miner now display their average frequency, making it easier to judge the representativeness of aggregated connections.
- For clustered nodes, the Fuzzy Miner now provides an enhanced expander view with much more detail, including the average frequency of the cluster and statistics for each contained node (see Figure 3.4).

These changes make the visualization environment more transparent and adaptable. Figure 3.4 illustrates the updated Fuzzy Miner interface, where additional sliders provide finer control over filtering and clustering, and nodes and edges are visualized with more details.



Log Filtering

SPM Threshold 0.10 (0.00–1.00) 0,10

Node Frequency (normalized) 0.25 (0.00–1.00) 0,25

Node Frequency (absolute) 10 (0–40) 10

Node Filtering

Unary Significance 0.70 (0.00–1.00) 0,70

Correlation 0.45 (0.00–1.00) 0,45

Edge Filtering

Edge Frequency (normalized) 0.25 (0.00–1.00) 0,25

Edge Frequency (absolute) 3 (0–13) 3

Binary Significance 0.00 (0.00–1.00) 0,00

Edge Cutoff 0.00 (0.00–1.00) 0,00

Utility Ratio 0.00 (0.00–1.00) 0,00

Fuzzy Mining

Node: Cluster 2

Average Frequency *(absolute)*: 17

Average Frequency *(normalized)*/Average Unary Significance: 0.42

Clustered Nodes:

d:

- SPM value: 0.64
- Frequency *(absolute)*: 17
- Frequency *(normalized)*/Unary Significance: 0.42

Figure 3.4: Enhanced Fuzzy Miner.

3.4 Integration of Genetic Miner

The central contribution of this thesis is the integration of the Genetic Miner, a discovery algorithm designed to handle noise and incompleteness by treating process discovery as a global search problem. The implementation mostly follows Alves de Medeiros et al. [14], adapted to the tool's architecture.

Adaptations and Limitations

While the overall workflow follows the theoretical design, two important deviations remain:

- **Silent transitions:** Unlike the original design, the current implementation does not yet insert silent transitions. This omission simplifies the visualization but leads to incomplete semantics in certain edge cases.
- **Completion semantics:** The token-game simulation used for fitness evaluation is functional, but its handling of completion is still marked as a workaround. Some unit tests (notably for length-two loops) include a `TODO` marker and will need adjustment once the final semantics are implemented.

These points are documented and form the basis for future improvements.

Testing

The Genetic Miner is covered by a comprehensive set of unit and integration tests. In addition to the general tests shared by all miners (filtering and log parsing), the following aspects were specifically validated:

- Initialization of dependency, start, and end measures.
- Creation of heuristic individuals from dependency matrices.
- Correctness of the fitness evaluation on perfect and imperfect models.
- Validity of crossover and mutation operators, including consistency repair of input/output sets.
- Graph integrity, ensuring that `Start` and `End` nodes are always present and reachable.
- Parsing semantics for OR, AND, loops of length one and two (L1L/L2L), and combinations.
- Integration tests with CSV and TXT logs, verifying that generated models remain structurally sound.

One specific test case for L2L semantics is currently set to `assertTrue` by default but marked with a `TODO` comment. Once the completion workaround in the token game is resolved, this assertion must be corrected to properly reflect expected behavior.

3.5 Working with the Project

The Process Mining Visualization Tool is designed not only as a research prototype but also as a living open-source project that can be continuously extended. This section summarizes the most important practices for developers who want to contribute new features or maintain existing ones. The modular *MVC* architecture ensures that improvements can usually be made by extending existing base classes rather than rewriting large parts of the code.

The following guidelines outline how to add new pages, extend views, define new column selections, integrate further mining algorithms, and generally extend the application. This provides future students and contributors with a practical entry point into the codebase.

3.5.1 Adding a Page

Pages are organized around the MVC pattern. To create a new page, at least two classes need to be implemented:

- a **controller** (subclassing `BaseController`) that manages the logic, and
- a **view** (subclassing `BaseView`) that handles layout and rendering.

Both classes are placed in their own folder inside the `ui` package. The controller is responsible for reacting to user inputs (e.g., slider adjustments or button clicks), while the view provides the layout elements. Logic that belongs to data processing or mining is encapsulated in models and should not be placed inside the view.

Controllers implement methods such as `run()` (executed at every Streamlit reload) and `process_session_state()` (to read and update Streamlit's session state). For multi-view pages, `select_view()` can be implemented to switch between layouts dynamically.

3.5.2 Adding a View to a Page

New views can be added if alternative layouts or specialized visualizations are needed. Views must either subclass `BaseView` directly or extend an existing page-specific view. Switching between views is coordinated in the controller via `select_view()`.

3.5.3 Adding New Column Selections

Some algorithms may require more information than just case ID, activity, and timestamp. Examples include resource-aware miners that need a *resource* or *role* column.

To add such columns, templates are available in the `templates/column_selection_template` folder. Developers can either start from a minimal template and define all needed columns, or extend the provided template that already includes the basic three. New selection boxes must be registered with keys that match the expected parameters of the algorithm's controller, ensuring smooth transformation of the DataFrame into event logs.

If the new columns affect the log transformation, the method `transform_df_to_log()` should be overridden in the corresponding controller.

3.5.4 Adding a Mining Algorithm

Adding a new discovery algorithm requires three components:

1. **MiningModel** – the algorithm itself, implemented in `mining_algorithms`. This should extend `BaseMining` to reuse core functionality such as filtering, succession matrix calculation, and SPM scoring. A dedicated Graph class (extending `BaseGraph`) should also be created.

2. **AlgorithmController** – a subclass of `BaseAlgorithmController`, responsible for reading parameter values from Streamlit's session state, triggering the mining run, and redisplaying the graph when parameters change.

3. **AlgorithmView** – a subclass of `BaseAlgorithmView`, defining the sidebar sliders, buttons and controls. Slider keys must match the session state keys used by the controller.

The Genetic Miner, for instance, was integrated this way: the `GeneticMining` model implements the algorithm, `GeneticMinerController` manages population, crossover, mutation, and other parameters, and `GeneticMinerView` provides the UI sidebar elements.

To register the new algorithm, the route and mappings must be added in `config.py`. Optionally, a Markdown documentation file can be placed in `docs/algorithms` and linked in the mapping dictionary.

3.5.5 Extending the Application

Beyond mining algorithms, many other extension points exist:

- **Filters:** Additional filters can be added to `BaseMining` or specialized miners.

- **Import/Export:** Support for more formats (e.g., XES, JSON logs) can be added by extending the I/O operations.

- **Testing:** New unit tests should be placed in the `tests` directory to ensure correctness and reproducibility.

Future contributors are encouraged to consult the `Contributing.md` file in the GitHub repository[1], which provides detailed guidance on how to extend the project.

4 Evaluation & Discussion

This chapter evaluates the effectiveness of the extended *Process Mining Visualization Tool* with the newly integrated *Genetic Miner*. The focus lies on comparing the implementation with reference approaches, examining the sensitivity of its parameters in theory, and documenting current limitations together with directions for improvement. As the Genetic Miner is the centerpiece of this thesis, particular attention is given to how its design addresses noise, incomplete logs, and parameter variation in principle, even though a full empirical study was not yet possible.

4.1 Sensitivity Analysis

The Genetic Miner relies on a number of hyperparameters that strongly influence the quality of discovered models: population size, mutation rate, crossover rate, tournament size, elitism rate, and the dependency-matrix exponent (`power_value`). Each parameter steers the exploration–exploitation balance of the search.

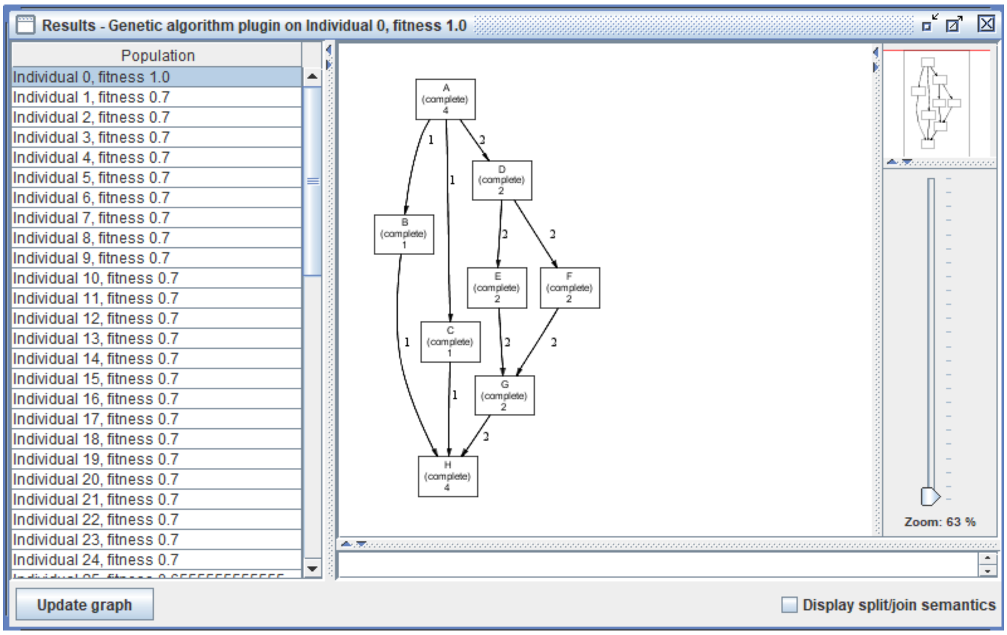
Theoretical sensitivity Large populations and many generations generally increase solution quality, but at a quadratic cost in runtime. High mutation rates promote diversity but risk disrupting promising structures, while too low mutation leads to premature convergence. Tournament size and elitism balance exploration against the preservation of high-quality models. The `power_value` parameter directly affects initialization sparsity: odd values above 1 prune weak causal relations, which is useful for noisy logs but can overly restrict search space when event data is sparse. [14, 15]

Status of implementation While parameter sliders are exposed in the Streamlit UI, a systematic empirical study was not yet possible. The current implementation lacks a complete trace completion check in the token-game simulation (see Section 4.3), which means that sensitivity with respect to completion-related fitness cannot yet be properly assessed. Once fixed, controlled experiments on synthetic and real-life logs can measure robustness against incomplete or inconsistent data, thereby validating the theoretical expectations.

4.2 Reference Implementations

To validate correctness, the Genetic Miner was compared against the ProM reference implementation [17]. Both share the same conceptual basis [14, 2], including individual representation, GA operators, and fitness evaluation. However, differences were observed:

- On simple logs like `genetic_test.csv` in the `testcsv` folder in the project, the Process Mining Visualization Tool and ProM produced equivalent models, confirming the correct integration of initialization, crossover, mutation logic, and parsing.



Results - Genetic algorithm plugin on Individual 0, fitness 1.0

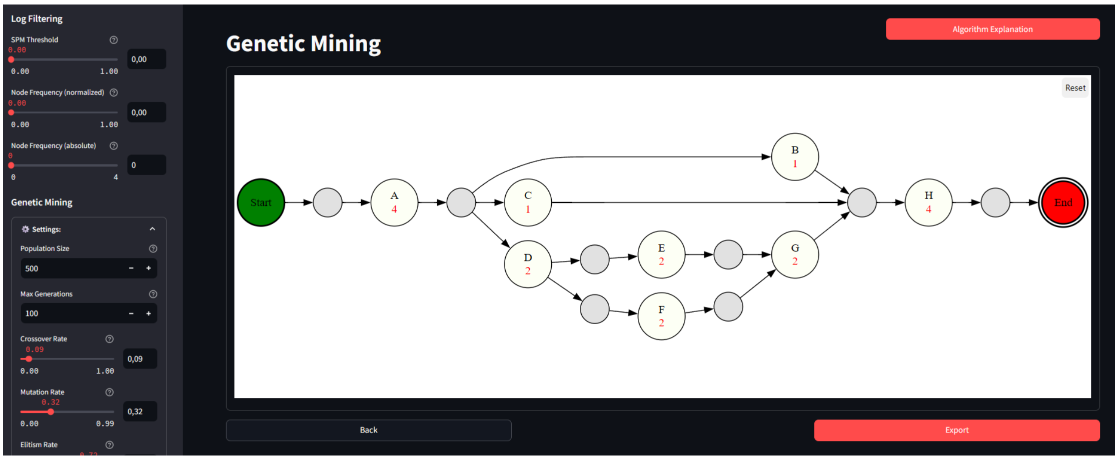
Population

Individual 0, fitness 1.0
Individual 1, fitness 0.7
Individual 2, fitness 0.7
Individual 3, fitness 0.7
Individual 4, fitness 0.7
Individual 5, fitness 0.7
Individual 6, fitness 0.7
Individual 7, fitness 0.7
Individual 8, fitness 0.7
Individual 9, fitness 0.7
Individual 10, fitness 0.7
Individual 11, fitness 0.7
Individual 12, fitness 0.7
Individual 13, fitness 0.7
Individual 14, fitness 0.7
Individual 15, fitness 0.7
Individual 16, fitness 0.7
Individual 17, fitness 0.7
Individual 18, fitness 0.7
Individual 19, fitness 0.7
Individual 20, fitness 0.7
Individual 21, fitness 0.7
Individual 22, fitness 0.7
Individual 23, fitness 0.7
Individual 24, fitness 0.7

Zoom: 63 %

☐ Display split/join semantics

Figure 4.1: Genetic Miner in ProM (fitness = 1.0).



Log Filtering

SPM Threshold 0.00 (0.00 – 1.00) 0,00

Node Frequency (normalized) 0.00 (0.00 – 1.00) 0,00

Node Frequency (absolute) 0 (0 – 4) 0

Genetic Mining

Settings:

Population Size 500

Max Generations 100

Crossover Rate 0.09 (0.00 – 1.00) 0,09

Mutation Rate 0.32 (0.00 – 0.99) 0,32

Elitism Rate

Figure 4.2: Genetic Miner in Process Mining Visualization Tool (fitness = 1.0).

- On more complex or noisy logs, deviations occurred. The reason lies in the incomplete *completion check* of the parse-token-game in the tool: currently, traces can be marked as finished even if tokens remain. As a result, fitness values diverge and model evolution may converge to different solutions.

- Silent transitions are not supported in the tool at the moment, whereas ProM inserts them to resolve duplicate edges and properly represent cut structures. This leads to duplicate connection or missing synchronization points in outputs of the tool.

These deviations highlight the importance of aligning the token semantics and silent transition handling with the ProM standard to ensure rediscoverability.

4.3 Limitations

The current implementation demonstrates the feasibility of integrating a GA-based discovery algorithm into the Process Mining Visualization Tool but has still clear limitations and deviations from the theoretical algorithm. [14] Documenting them precisely serves as a guide for future contributors:

- **Incomplete parsing logic** The token-game simulation omits a *completion check*: if tokens remain in the net after firing all observed events, traces are incorrectly considered completed. This inflates fitness values and biases selection. *Required fix:* Extend the `parse_trace_token_game()` method to include a final check verifying that the marking is empty after parsing and adjust the marking logic accordingly. This method is located in `genetic_mining.py`.

- **Absence of silent transitions** The current implementation does not support *silent transitions*, which are crucial for representing invisible routing behavior in Petri nets. These transitions do not correspond to any event in the log but serve as routing constructs to merge parallel edges, synchronize branches, and resolve structural patterns such as non-free-choice constructs, i.e., situations where choices between branches depend on additional synchronization with other parts of the process. [14, 2]

  Without them, the discovered models may exhibit artifacts such as *duplicate edges* or misleading direct connections. Figure 4.3 illustrates this issue: between activities `b` and `c`, the tool generates two parallel edges instead of a single connection mediated by a silent transition. *Required fix:* Extend the *graph generation* to insert silent transitions whenever duplicate edges or synchronization points appear. This ensures that the Petri net visualization is structurally correct and avoids artifacts such as multiple parallel edges.

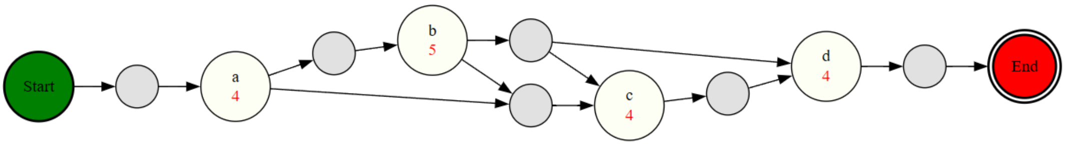


Figure 4.3: Duplicate connection between b and c in the implementation due to missing silent transitions.

- **Evaluation scope** Because of the above issues, systematic benchmarking against complex reference logs (e.g., from the BPI Challenges) could not be carried out. Only simpler logs yield comparable results to ProM. This limitation reduces the strength of conclusions about scalability and robustness.

Together, these restrictions explain the observed deviations and should be addressed in future work to bring the implementation in line with the theoretical approach. For a more detailed discussion of the underlying design choices and handling of silent transitions, see the original description of the Genetic Miner by Alves de Medeiros et al. [14].

4.4 Discussion

The integration of the Genetic Miner demonstrates that the tool can host not only local, statistics-based discovery techniques (Alpha, Heuristic, Inductive, Fuzzy), but also global search-based approaches. This broadens the tool's applicability to noisy, incomplete, and weakly structured event data. The sensitivity analysis underscores the flexibility of GA-based discovery, but also the necessity of correct semantics in the completion logic. Without proper completion checks and silent transitions, discovered models show structural inaccuracies in some cases, even if activity parsing is correct.

In sum, the evaluation highlights both the potential of the Genetic Miner integration for handling noisy and incomplete logs, and the unfinished aspects of the current implementation. Once the identified limitations are resolved, the tool will be able to serve as a robust, open-source alternative for teaching, experimentation, and research in genetic process mining. Until then, users should be aware of the semantic differences compared to ProM when interpreting results, as illustrated in Figures 4.2 and 4.3.

5 Conclusion & Future Work

This chapter summarizes the contributions of this thesis, reflects on the challenges encountered, and outlines directions for future work. The work presented here has expanded the Process Mining Visualization Tool into a more robust, extensible, and user-friendly platform, while also introducing the Genetic Miner as a new mining algorithm.

5.1 Summary of Contributions

The main contributions of this thesis can be grouped into three areas:

- **Merging of codebases** Earlier versions of the tool existed in fragmented form: one desktop-based implementation including the Alpha Miner and SPM filter [18], and a Streamlit-based version with the Inductive Miner implemented [6]. These were merged into a single coherent platform.
- **Enhancements to filtering and visualization** The SPM filter was made globally available across all algorithms, node- and edge-based filtering were introduced, and synchronized slider controls were implemented. Interactive visualization was improved through edge click-handlers and corrected clustering logic, making models more interpretable even for noisy or large logs.
- **Integration of the Genetic Miner** The implementation exposes key parameters (population size, crossover and mutation rate, elitism, tournament size, and power value) directly in the UI-sidebar. It covers initialization, fitness evaluation via a token-game simulation, and evolutionary operators, producing Petri-net–like models robust against noise and incomplete data. While functionally aligned with the theoretical algorithm [14] in most aspects, the integration also revealed important open points such as missing silent transitions and incomplete completion checks inside the token-game simulation.

5.2 Challenges

The development process revealed several challenges, both technical and conceptual:

- **Balancing usability and correctness** Streamlit provided a modern and interactive interface, but also introduced state-management issues (e.g., synchronized thresholds across filters). Ensuring consistent and interpretable outputs required additional synchronization logic in the controllers and `BaseMining` class.

- **Handling of token semantics and silent transitions** While ProM inserts silent transitions to resolve duplicate connections and structural conflicts, the current implementation omits them, leading to limitations in rediscoverability. Similarly, the token-game logic currently lacks a full completion check, allowing fitness values to diverge from the reference.

- **Evaluation constraints** A full-scale benchmark study across real-life logs was not yet possible, as the current implementation lacks a complete completion check in the token-game logic. This prevents reliable fitness evaluation and, consequently, systematic parameter sensitivity experiments. For this reason, only theoretical analysis and limited empirical testing on small synthetic logs could be performed within the scope of this thesis.

5.3 Future Work

While the thesis has established a solid foundation, several important extensions remain open for future work. The most relevant directions are:

- **Completion checks for token-game logic** Extending the fitness evaluation with a full trace completion check will align the implementation with the theoretical algorithm and the ProM Plug-in reference, ensuring that traces are only marked as completed when no tokens remain.

- **Support for silent transitions** Introducing silent transitions will enable correct handling of duplicate edges, synchronization points, and non-free-choice constructs, thereby improving rediscoverability.

- **Enhanced fitness metrics** Beyond the current parsing and completion ratio, additional dimensions such as precision or generalization could be incorporated, leading to more balanced evolutionary search. [15]

- **Integration of further algorithms** The tool's modular design allows future contributors to add additional discovery methods (e.g., region-based miners, alignment-based approaches). This would position the tool as an even more comprehensive educational and research platform.

Overall, this thesis demonstrates that the Process Mining Visualization Tool can serve as an accessible and extensible environment for process mining research and teaching. By merging earlier fragmentation, enhancing filtering and visualization, and integrating the Genetic Miner, the tool has been brought significantly closer to parity with established frameworks such as ProM, while remaining more approachable for students and practitioners. Future work can build on these foundations to further improve correctness, scalability, and breadth of supported algorithms.

Bibliography

[1] Wil Aalst and Boudewijn Dongen. Discovering petri nets from event logs. *Lecture Notes in Computer Science*, 01 2013.

[2] W.M.P. {Aalst, van der}. *Process mining : discovery, conformance and enhancement of business processes.* Springer, Germany, 2011.

[3] Celonis. Process intelligence and process mining, 2025. `https://www.celonis.com/`, Accessed on 2025-09-23.

[4] A. Chen. A python desktop app for business. process mining and visualization. Bachelor thesis, University of Vienna, 2023.

[5] Fluxicon. Process mining and automated process discovery software for professionals – fluxicon disco, 2025. `https://fluxicon.com/disco/`, Accessed on 2025-09-23.

[6] F. Fraunberger. Enhancing the process mining visualization tool by adding the inductive miner and improving the ui. Bachelor thesis, University of Vienna, 2024.

[7] Graphviz. Graphviz, 2025. `https://graphviz.org/`, Accessed on 2025-09-24.

[8] Ralph F. Grove and Eray Ozkan. The mvc-web design pattern. In *Proceedings of the 7th International Conference on Web Information Systems and Technologies - Volume 1: WEBIST,*, pages 127–130. INSTICC, SciTePress, 2011.

[9] C.W. Günther and W.M.P. {Aalst, van der}. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In G. Alonso, P. Dadam, and M. Rosemann, editors, *Proceedings of the 5th International Conference on Business Process Management (BPM 2007) 24-28 September 2007, Brisbane, Australia*, Lecture Notes in Computer Science, pages 328–343, Germany, 2007. Springer. 5th International Conference on Business Process Management (BPM 2007), BPM 2007 ; Conference date: 24-09-2007 Through 28-09-2007.

[10] A. Krasniqi. Fuzzy miner as an additional process mining algorithm for a process mining visualization tool in python. Bachelor thesis, University of Vienna, 2024.

[11] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering block-structured process models from incomplete event logs. In G. Ciardo and E. Kindler, editors, *Application and Theory of Petri Nets and Concurrency (PETRI NETS 2014)*, volume 8489 of *Lecture Notes in Computer Science*, pages 91–110. Springer, Cham, 2014.

[12] M. Lux. ddcal, 2023. `https://pypi.org/project/ddcal/`, Accessed on 2025-09-24.

[13] Marian Lux, Stefanie Rinderle-Ma, and Andrei Preda. Assessing the quality of search process models. In Marco Montali, Ingo Weber, Mathias Weske, and Jan vom Brocke, editors, *Business Process Management, 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9–14, 2018, Proceedings*, Lecture Notes in Computer Science, pages 445–461. Springer International Publishing, September 2018. Publisher Copyright: © Springer Nature Switzerland AG 2018.; Business Process Management : 16th International Conference on Business Process Management, BPM 2018 ; Conference date: 09-09-2018 Through 14-10-2018.

[14] A Medeiros, A. Weijters, and Wil Aalst. Using genetic algorithms to mine process models: representation, operators and results. *Systems Research and Behavioral Science - SYST RES BEHAV SCI*, 01 2005.

[15] A. Medeiros, A. Weijters, and Wil Aalst. Genetic process mining: An experimental evaluation. *Data Mining and Knowledge Discovery*, 14:245–304, 04 2007.

[16] NumPy-team. Numpy, 2025. `https://numpy.org/`, Accessed on 2025-09-24.

[17] ProMTools. Prom tools, 2025. `https://promtools.org/`, Accessed on 2025-09-23.

[18] A. Rustemi. Process mining visualization tool in python. Bachelor thesis, University of Vienna, 2024.

[19] Streamlit. Streamlit, 2025. `https://streamlit.io/`, Accessed on 2025-09-24.

[20] Wil van der Aalst. *Process Mining: Data Science in Action.* Springer, Berlin; Heidelberg; New York; Dordrecht; London, 2nd edition, 2016. Online resource, XIX, 467 p., 250 illus., 13 illus. in color.