Production Systems and Information Engineering
Volume 12 (2), pp. 51–63.
https://doi.org/10.32968/psaie.2024.2.4

DISCOVERING PROCESS MODELS CONTAINING XOR BRANCHES

ERIKA BAKSÁNÉ VARGA
University of Miskolc, Hungary Institute of Information Technology
erika.b.varga@uni-miskolc.hu

ATTILA BAKSA
University of Miskolc, Hungary Faculty of Mechanical Engineering and Informatics
attila.baksa@uni-miskolc.hu

Abstract: This study aims to investigate the robustness of the process discovery algorithms implemented in the PM4Py library. We created synthetic event logs to serve as benchmark datasets for evaluating process discovery methods in terms of the complexity of the event logs. Specifically, we developed a test framework using process models containing XOR branches. For simple XOR branches, all the examined algorithms (Alpha Miner, Inductive Miner, and Heuristics Miner) effectively uncovered the underlying process models. However, for more complex scenarios with events occurring in varied structural positions, heuristic and inductive approaches proved to be more reliable.

Keywords: *process discovery, PM4Py, process models with XOR branches*

1. Introduction

Effective business process management is vital for organizational success, and various methodologies, including process mining, play a crucial role in achieving and maintaining optimal performance levels. Process mining offers valuable support across the entire business process management (BPM) life-cycle, from initial design to continuous optimization [1]. In practice, process mining becomes particularly valuable when traditional methods fail to provide formal process descriptions, or when existing documentation lacks reliability [2]. By analyzing event logs, practitioners can compare observed behavior with predefined or expected patterns, ensuring adherence to prescribed standards [3].

Process mining constitutes a set of methodologies and techniques utilized to delve into event data, aiming to comprehend and enhance operational processes within organizations. Positioned at the intersection of data analysis and process management, process mining relies on event logs generated by business information systems, which typically contain essential elements such as transaction identifiers (case IDs), activity descriptions, timestamps, and sometimes supplementary details like resource allocations and costs.

The objectives of process mining can be categorized into three main areas: process discovery, conformance checking and process performance improvement [4].

The present research concentrates on process discovery, which involves the automated construction of process models based on event log data. By employing this technique, organizations gain valuable insights into the actual sequences of activities executed within their operational processes. Through process discovery, previously hidden patterns and variations in process execution can be uncovered, shedding light on both expected and unexpected process behaviors [5].

Process models that can be discovered from event logs include low-level models, like state transition systems, and high-level models, such as Petri-nets, process trees and BPMN models. These workflow models are built-up from basic control-flow structures [6] and are systematic representations of the sequence of processes and tasks within the organization.

In recent years, the process mining group of the Fraunhofer Institute has initiated a number of process mining-related projects. A notable example is the PM4Py (Process Mining for Python) library [7], which implements a wide range of state-of-the-art process mining algorithms. Several studies have evaluated existing process mining tools, including PM4Py, ProM, and Disco. For instance, [8] benchmarks these tools based on metrics such as process discovery accuracy and conformance checking, while [9] emphasizes PM4Py's flexibility, customization options, and suitability for large-scale experimentation. These capabilities have inspired numerous studies in diverse domains, including healthcare [9], business process management [10], finance [11], education [12], and traffic management [13].

In our investigations, we applied the process discovery algorithms implemented in the PM4Py library. We have created synthetic event logs including sequential and conditional patterns with a web-based process graph creator and log generator application [14] and evaluated the output of the process discovery methods as compared with the expected process model.

2. Input Data

2.1. Event Log Formats

In information systems, activities executed sequentially are stored in event logs, which are detailed records of these activities. An event log is essentially a structured table where each record corresponds to a specific case or process instance. The data captured during the execution of the sequence of activities within a given case are organized into columns. Event logs can be stored in various formats, such as CSV, JSON, and XML, but the standard format for storing case-based event logs is XES (eXtensible Event Stream) [15]. In these logs, events are grouped under process instances, referred to as cases or traces. Each case contains a unique case ID, a timestamp, and the IDs of the activities that occurred. Additionally, XES allows for the inclusion of other relevant information, such as the originator of the event, event types, and data attributes, providing a comprehensive view of the process execution.

This detailed structure facilitates effective analysis and monitoring of business processes, aiding in process improvement and compliance tracking [16].

Event logs serve as inputs for process discovery algorithms. In our experimental workflow, the synthetically generated event log files are formatted in plain text (TXT), with each line encapsulating the description of a process instance. Each activity within these cases is denoted by its symbolic name, typically represented by a letter indicative of the corresponding activity as shown in *Figure 1*. To process this data format, we developed a parser program specifically designed to convert these TXT files into CSV and XES event logs. During the transformation process, essential details such as the case ID and timestamps are automatically incorporated, ensuring the integrity and completeness of the event logs. Subsequently, these logs are integrated into the process discovery pipeline, enabling the use of a diverse set of algorithms available within the PM4Py library [7].

```
a,b,e,d,e
a,b,e,d,c,e,d,c,e,d,e
a,b,e,d,c,b,e,d,e
a,b,e,d,c,b,e,d,c,b,e,d,e,
a,b,e,d,c,b,e,d,c,e,d,c,b,e,d,e
```

Figure 1. Process instances in S_IR dataset

2.2. Control Structures in Event Logs

Typical workflow patterns [6] include sequential, parallel, conditional and loop structures as shown in *Figure 2*. In the simplest workflow structure, activities are performed one after the other in linear sequence. In a parallel pattern, some tasks can be executed simultaneously. An AND-split indicates the point where parallel paths are created, while an AND-join synchronizes these paths back into a single sequence. There are two types of conditional structures: exclusive and inclusive. In an exclusive choice, one of several possible paths is chosen at an XOR-split point based on a condition. If the conditional point is an OR-split, multiple paths can be taken simultaneously. In both cases, a join point is required to merge the alternative paths back into a single flow. Last, but not least, loop structures allow for the repetition of activities.

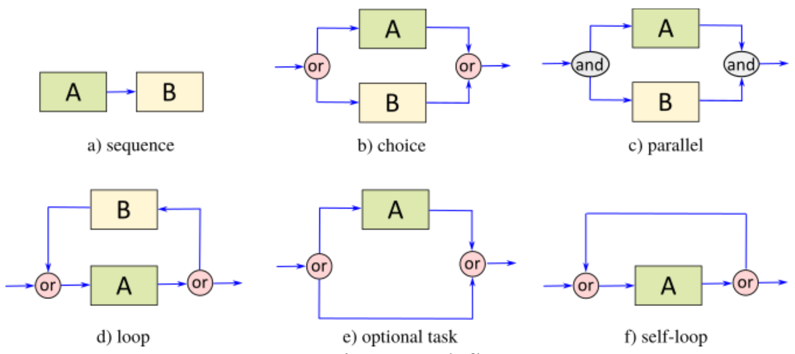


Figure 2. Basic control-flow patterns

For the present study, we created process graphs containing only sequential and exclusive choice workflow patterns. We then used these models to generate event logs, ensuring that all cases correspond to the predefined workflows.

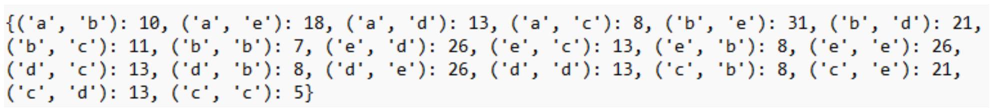
3. Methods

3.1. Process Discovery Methods in PM4Py

The algorithms available in the PM4Py library describe the discovered process in various ways. We examined the following algorithms:

- discover_eventually_follows_graph()
- discover_log_skeleton()
- discover_dfg_typed()
- discover_heuristics_net()
- discover_prefix_tree()
- discover_transition_system()
- discover_process_tree_inductive()
- discover_bpmn_inductive()
- discover_petri_net_alpha()
- discover_petri_net_heuristics()
- discover_petri_net_inductive()
- discover_petri_net_ilp()

The text representation of an Eventually Follows Graph (EFG) involves a list of pairs indicating which activity eventually follows the other. In the PM4Py implementation of the method, the frequency of each pair of subsequent activities is also given in the output file (see *Figure 3*).

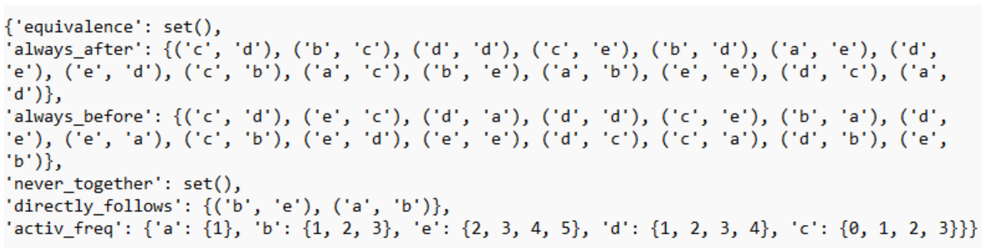


```
{('a', 'b'): 10, ('a', 'e'): 18, ('a', 'd'): 13, ('a', 'c'): 8, ('b', 'e'): 31, ('b', 'd'): 21,
('b', 'c'): 11, ('b', 'b'): 7, ('e', 'd'): 26, ('e', 'c'): 13, ('e', 'b'): 8, ('e', 'e'): 26,
('d', 'c'): 13, ('d', 'b'): 8, ('d', 'e'): 26, ('d', 'd'): 13, ('c', 'b'): 8, ('c', 'e'): 21,
('c', 'd'): 13, ('c', 'c'): 5}
```

Figure 3. EFG graph of S_IR dataset

A log skeleton is again a textual representation (see *Figure 4*) used to describe the underlying structure and constraints of an event log in the following grouping:

- **equivalence:** the order of the two activities can be swapped.
- **always_after:** activity pairs where the first activity is always followed by the second.
- **always_before:** activity pairs where the first activity always precedes the second.
- **never_together:** activity pairs that never occur simultaneously in any process instance.
- **directly_follows:** activity pairs where the first activity is immediately followed by the second.
- **activ_freq:** the occurrence frequencies of the given activity in the process instances.



```
{'equivalence': set(),
'always_after': {('c', 'd'), ('b', 'c'), ('d', 'd'), ('c', 'e'), ('b', 'd'), ('a', 'e'), ('d',
'e'), ('e', 'd'), ('c', 'b'), ('a', 'c'), ('b', 'e'), ('a', 'b'), ('e', 'e'), ('d', 'c'), ('a',
'd')},
'always_before': {('c', 'd'), ('e', 'c'), ('d', 'a'), ('d', 'd'), ('c', 'e'), ('b', 'a'), ('d',
'e'), ('e', 'a'), ('c', 'b'), ('e', 'd'), ('e', 'e'), ('d', 'c'), ('c', 'a'), ('d', 'b'), ('e',
'b')},
'never_together': set(),
'directly_follows': {('b', 'e'), ('a', 'b')},
'activ_freq': {'a': {1}, 'b': {1, 2, 3}, 'e': {2, 3, 4, 5}, 'd': {1, 2, 3, 4}, 'c': {0, 1, 2, 3}}}
```

Figure 4. Log skeleton of S_IR dataset

The algorithms producing a Directly Follows Graph (DFG) and a Heuristic Net yield the same graph for the small-sized randomly generated S_IR dataset as can be seen in *Figure 5*. These graphs describe the relationships and frequencies of event sequences within a process. Each node represents a unique activity and its frequency in the given position of the process instances. Directed edges between the nodes show the frequency and direction of sequences between the activities.

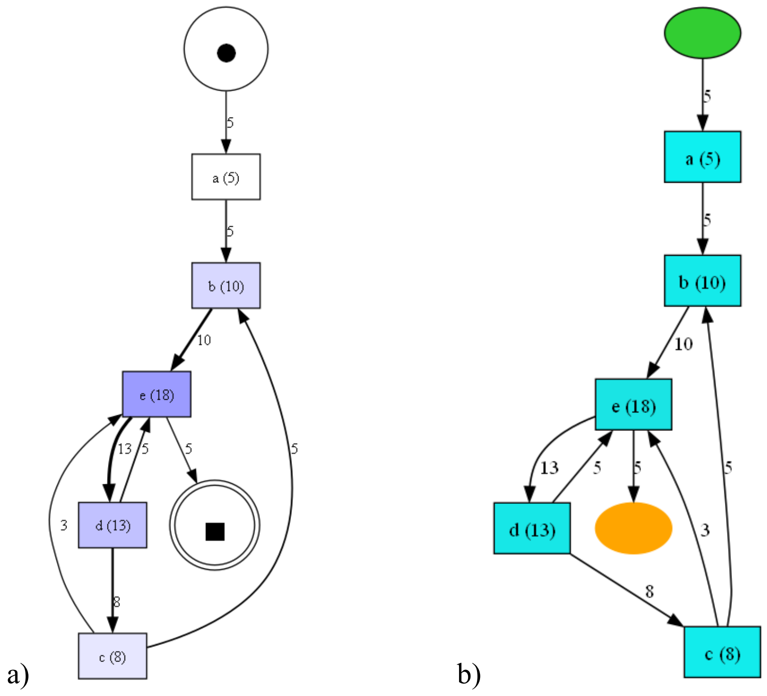


Figure 5. a) DFG Graph and b) Heuristic Net of S_IR dataset

So, for example the edge from node ‘a’ to node ‘b’ ($a \rightarrow b$) indicates that activity 'b' directly follows activity ‘a’ five times in the observed process instances. On the other hand, the absence of edge between node ‘a’ and ‘c’ suggests that this sequence does not occur or is infrequent in the observed data. The most frequent start and end activities are explicitly denoted in both graphs.

Another notation used to describe a process model is prefix tree. This is a tree-like data structure used for efficiently storing and retrieving a set of event sequences that share a common prefix. Each node represents an activity, and traversing from the root to a node along the edges forms a case that corresponds to the event sequence represented by that path (see *Figure 6*).

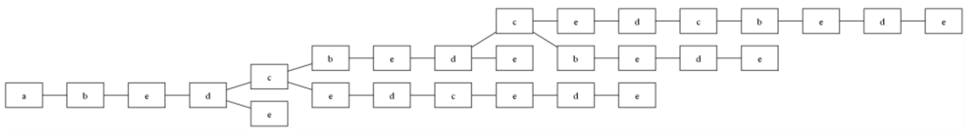


Figure 6. Prefix Tree of S_IR dataset

A state-transition system graph visually represents the states of the process instances and the possible transitions between these states (see *Figure 7*). States are denoted by nodes including pairs of subsequent events. Edges stand for transitions labeled with activities that trigger these transitions.

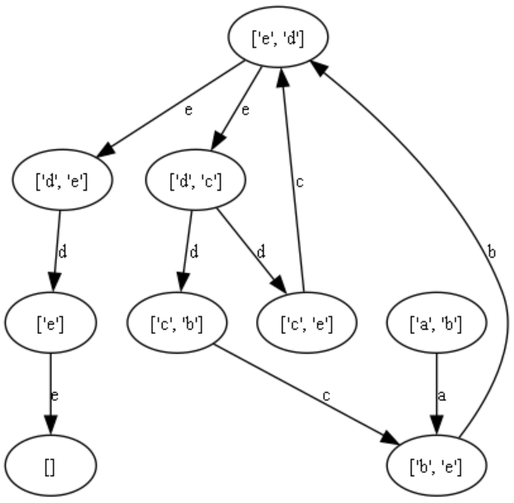


Figure 7. Transition System of S_IR dataset

Beside the above mentioned process representations, there are different types of process tree notations applied in the PM4Py program library. We can create the process tree, the BPMN or Petri-net graph of the discovered process model.

The algorithm that generates the process tree in *Figure 8* can uncover the following structures:

- **seq:** sequential execution
- **and:** joining parallel branches
- **xor:** executing one branch (exclusive or)
- **xor loop:** loop (repeating one branch of the process). The end of the repeating section is marked by a black dot.

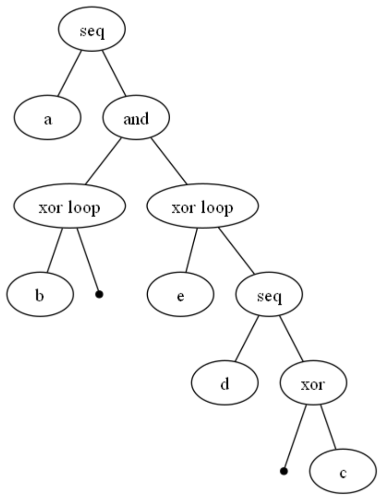


Figure 8. Process tree of S_IR dataset

In a BPMN graph activities are represented by rectangles. Circles denote the most frequent start and end event of the process instances, and diamonds are for denoting exclusive (X) and parallel (+) gateways (see *Figure 9*).

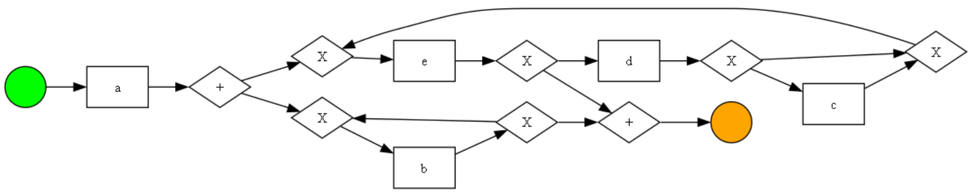


Figure 9 BPMN model of S_IR dataset

Petri-nets are popular output models of process discovery algorithms. In these diagrams, places are represented as circles denoting the state of the system where tokens (representing resources, tasks, or conditions) can reside. Transitions are represented as rectangles denoting events that can change the state of the system. Transitions may fire (activate) when certain conditions (token availability in input places) are met, which results in the consumption of tokens from input places and the production of tokens in output places. Arcs are directed connections between places and transitions indicating the flow of tokens between the elements. See an example in *Figure 10*.

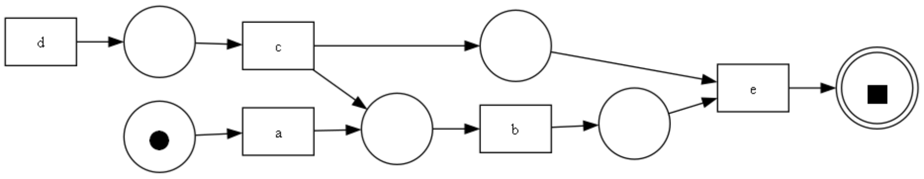


Figure 10. Petri-net model of S_IR dataset yielded by Alpha Miner

3.2. Evaluation of Process Discovery Methods

When assessing the models produced by the above algorithms, we use the functions of the pm4py.algo.evaluation package and consider four criteria:

1. ***Fitness*****:** This measures how well the discovered model aligns with the cases recorded in the event log, indicating the percentage of cases it accurately represents.
2. ***Precision*****:** This criterion ensures that the discovered model does not generate cases that are not present in the event log. It measures what percentage of the cases generated by the model actually appear in the original event log.
3. ***Generalization*****:** This assesses the extent to which the discovered model can generalize beyond the specific cases in the event log.
4. ***Simplicity*****:** This evaluates how straightforward and uncomplicated the discovered model is.

3.3. Test Data

We generated synthetic datasets for benchmark purposes in two stages. In the first step, a process model graph was drawn with a web-based graph creator tool. This application allows for setting the frequencies of events and directly-follows-relations to be used when producing the process instances in the generated event log. For the present study, we constructed workflow graphs containing sequences and XOR branches following this systematic structural layout:

- Process models containing single-depth XOR branches (X) (3 files)
- Process models containing double-depth XOR branches (XX) (2 files)
- Process models containing triple-depth XOR branches (XXX) (2 files)

The generated event logs are stored in TXT format and their statistical features are listed in *Table 1*.

Table 1. Statistical data of the test event logs

Event log	Process instances	All events	Event types
S_X_01	1000	5000	7
S_X_02	1000	7000	10
S_X_04	1000	11000	16
S_XX_01	1000	3932	7
S_XX_10	1000	4534	6
S_XXX_01	1000	4026	9
S_XXX_10	1000	3911	9

4. Results

This section summarizes the results gained when executing the process discovery methods for the examined synthetic event logs.

4.1. *Conformance of Expected and Discovered Process Models*

Logs containing single-depth XOR branches (S_X_01, S_X_02, S_X_04) differ only in the number of involved events and types of events. In these simple cases, the Alpha Miner, Inductive Miner and Heuristics Miner algorithms discover the same process models, which exactly correspond to the ones we have created to generate the event logs.

We have received the same results when increasing the number of XOR branch embeddings to three (S_XXX_01, S_XXX_10). So we can conclude, that one can rely on all three algorithms when discovering process models containing only sequences and XOR branches where events do not occur in multiple structural positions. The number of event types and the length of the process instances do not make any difference. These graphs are displayed in *Figures 11* and *12*.

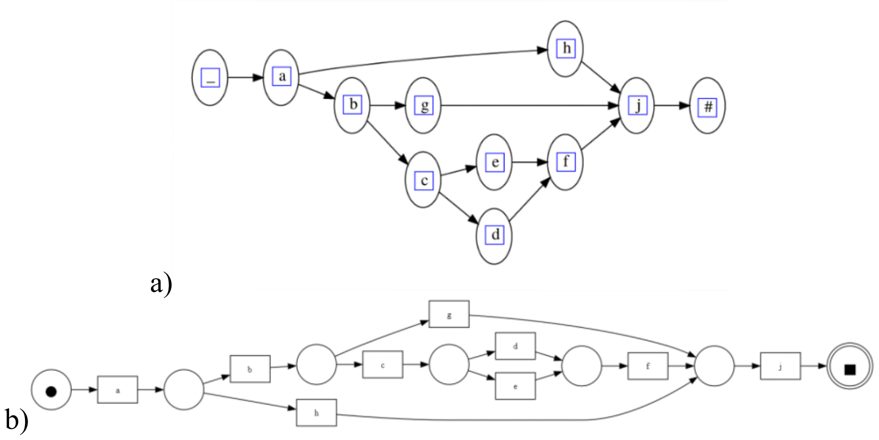


Figure 11. a) Expected and b) discovered process model for S_XXX_01 dataset

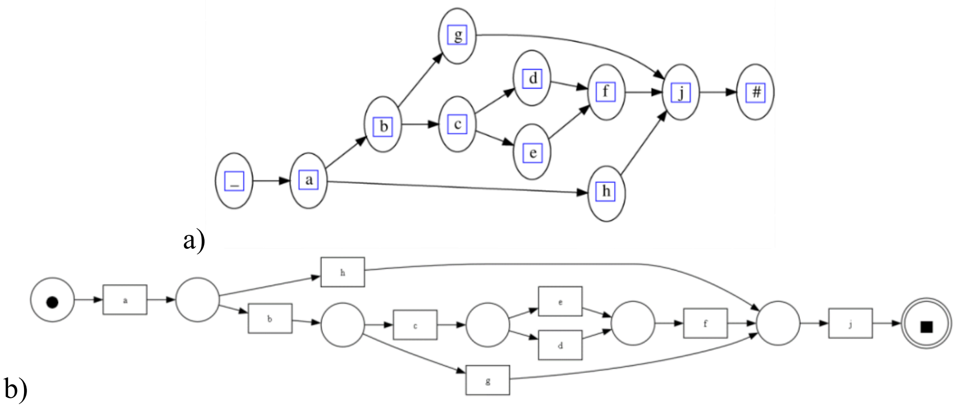


Figure 12. a) Expected and b) discovered process model for S_XXX_10 dataset

On the other hand, the two event logs containing double-depth XOR branches (S_XX_01, S_XX_10) are similar in event size and process length, but the second dataset is more complex. Activity ‘b’ and ‘e’ may occur in two different structural positions: as part of a sequence and as part of the XOR branch. This difference is the cause of the diverging models explored by the different algorithms shown in *Figures 13* and *14*.

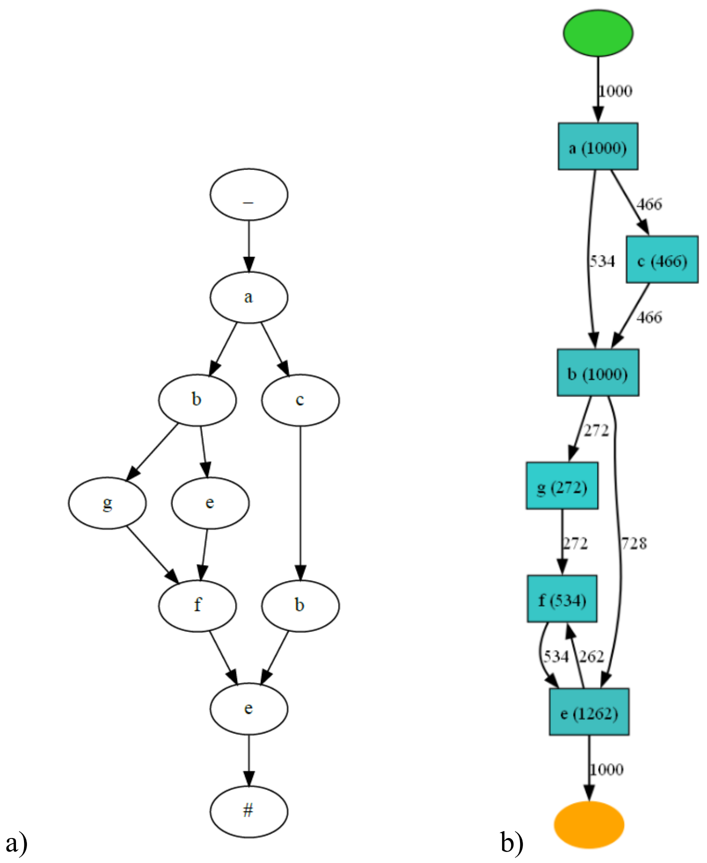


Figure 13. Process model for S_XX_10 dataset:
a) expected, b) produced by Heuristics Miner

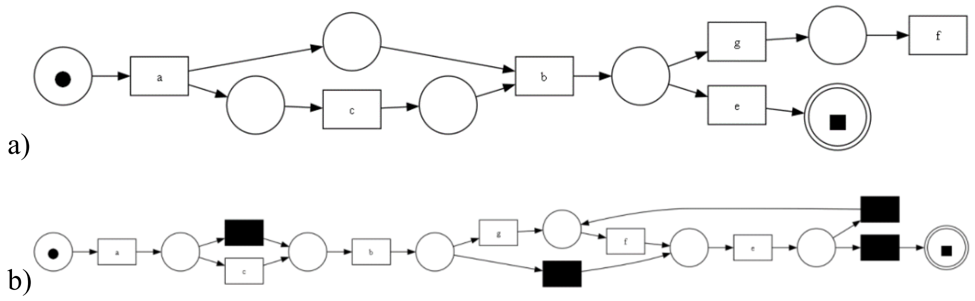


Figure 14. Process model for S_XX_10 dataset discovered by a) Alpha Miner, b) Inductive Miner

Among these graphs, the heuristic net and the Petri-net produced by the Inductive Miner fit the original process model. We can see that Alpha Miner has limitations handling infrequent behavior.

4.2. Comparative Evaluation of Process Discovery Methods

Table 2 gives a summary of the evaluation metrics calculated by means of the following functions:

- pm4py.fitness_token_based_replay()
- pm4py.fitness_alignments()
- pm4py.precision_token_based_replay()
- pm4py.precision_alignments()
- pm4py.algo.evaluation.generalization.algorithm.apply()
- pm4py.algo.evaluation.simplicity.algorithm.apply()

Table 2. Evaluation of the discovered process models

Dataset	**Method**	**Fitness**	**Precision**	**Generali-zation**	**Simplicity**
S_X_01	All methods	1.0	1.0	0.96	0.87
S_X_02	All methods	1.0	1.0	0.96	0.82
S_X_04	All methods	1.0	1.0	0.96	0.78
S_XX_01	All methods	1.0	1.0	0.95	0.87
S_XX_10	**Alpha Miner**	**0.83**	**0.93**	**0.96**	**1.0**
S_XX_10	Heuristics M.	1.0	0.95	0.96	0.81
S_XX_10	ILP Miner	1.0	0.87	0.96	0.68
S_XX_10	Inductive M.	1.0	0.83	0.96	0.77
S_XXX_01	All methods	1.0	1.0	0.94	0.80
S_XXX_10	All methods	1.0	1.0	0.94	0.80

Our analysis revealed a negative correlation between fitness and simplicity, indicating that as fitness improves, model simplicity tends to decrease ($r = -0.76$). From the evaluation of the methods, we can conclude that they aim to maximize fitness and generalization. In simpler cases, they generate process models that align closely with the datasets while maintaining a balance between generalization and simplicity. However, with higher event log complexity, differences between the process discovery methods become apparent.

The Alpha Miner algorithm tends to prioritize simplicity and generalization over fitness, which can result in models that may not accurately represent all observed behaviors in the event log. In contrast, Heuristics Miner, ILP Miner, and Inductive Miner strive to maximize fitness and generalization, even at the cost of producing more complex models.

5. Conclusion

In this study, we explored process discovery techniques for event logs containing XOR branches, utilizing various algorithms implemented in the PM4Py library. Our synthetic datasets, designed with single, double, and triple-depth XOR branches, provided a controlled environment to evaluate the efficacy and limitations of different process discovery methods.

Our findings revealed that the Alpha Miner, Inductive Miner, and Heuristics Miner algorithms consistently discovered accurate process models for event logs with XOR branches. These models matched the predefined structures used to generate the event logs, indicating the robustness of these algorithms in handling straightforward sequences and XOR branches where events do not repeat in multiple structural positions.

However, the analysis of double-depth XOR branches highlighted challenges in process discovery, particularly when events occur in multiple structural positions. The resulting models from different algorithms varied, demonstrating that certain algorithms, like Alpha Miner, struggle with infrequent behaviors and complex branching structures. In contrast, the models generated by Inductive Miner and Heuristics Miner showed better alignment with the original process models, suggesting their enhanced capability in handling more complex XOR branching scenarios.

The evaluation metrics, including fitness, precision, generalization, and simplicity, provided a comprehensive assessment of the discovered models. The results indicated that while the algorithms generally achieved high fitness, maintaining a balance between generalization and simplicity sometimes compromised precision, leading to the generation of cases not found in the event logs. These findings underscore the importance of choosing the appropriate algorithm based on the complexity and characteristics of the event logs being analyzed.

References

[1] Lamghari, Z., Radgui, M., Saidi, R., and Rahmani, M. D. (2018). A set of indicators for BPM life cycle improvement, *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, Fez, Morocco, pp. 1–8. https://doi.org/10.1109/ISACV.2018.8354057.

[2] Dumas, M., La Rosa, M., Mendling, J., and Reijers, H. A. (2013). Introduction to Business Process Management. In: *Fundamentals of Business Process Management.* Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33143-5_1

[3] Di Francescomarino, C., Burattin, A., Janiesch, C., and Sadiq, S. (eds.) (2023). *Business Process Management.* 21st International Conference, BPM 2023, Utrecht, The Netherlands, September 11–15, 2023, Proceedings. Vol. 14159, Springer Nature, https://doi.org/10.1007/978-3-031-41620-0

[4] van der Aalst, W. M. P. (2016). *Process Mining – Data Science in Action*, Springer Berlin, Heidelberg, 2nd edition. https://doi.org/10.1007/978-3-662-49851-4

[5] van der Aalst, W. M. P., Carmona, J. (2022). *Process Mining Handbook*, Springer, Cham. https://doi.org/10.1007/978-3-031-08848-3

[6] Russell, N., ter Hofstede, A. H. M., van der Aalst, W. M. P., and Mulyar, N. (2006). *Workflow Control-Flow Patterns: A Revised View*. BPM Center Report BPM-06-22, BPMcenter.org, 2006

[7] Berti, A., van Zelst, S., and Schuster, D. (2023). PM4Py: A process mining library for Python. *Software Impacts*, 17, p. 100556. https://doi.org/10.1016/j.simpa.2023.100556.

[8] Parente, C., and Costa, C. J. (2022). Comparing Process Mining Tools and Algorithms. *17th Iberian Conference on Information Systems and Technologies (CISTI)*, Madrid, Spain, pp. 1-7. https://doi.org/10.23919/CISTI54924.2022.9820570.

[9] Gomes, A. F. D., Wanzeller, C., and Fialho, J. (2021). Comparative Analysis of Process Mining Tools. *CAPSI 2021 Proceedings*, 4. https://aisel.aisnet.org/capsi2021/4.

[10] Masyuk, M., and Dorrer, M. (2024). Using PM4Py for Process Mining in an Educational Organization. In: *Advances in Automation V*. Proceedings of the International Russian Automation Conference, RusAutoCon2023, September 10–16, 2023, Sochi, Russia, pp. 234–242. https://doi.org/10.1007/978-3-031-51127-1_23.

[11] Tripathi, A., Rai, A., Singh, U., Vyas, R., and Vyas, O. P. (2024). Unveiling AI Efficiency: Loan Application Process Optimization Using PM4PY Tool. *Advanced Computing*, IACC 2023, Communications in Computer and Information Science, Vol. 2053. https://doi.org/10.1007/978-3-031-56700-1_39.

[12] Baksáné Varga, E., and Baksa, A. (2025). Application of Process Discovery Methods for Learning Process Modeling, accepted in *Infocommunications Journal.*

[13] Kovács, L., and Jlidi, A. (2024). Navigating Process Mining: A Case study using pm4py. *Production Systems and Information Engineering*, Vol. 12, No. 1. https://doi.org/10.32968/psaie.2024.1.5.

[14] Mileff, P. (2024). Design and development of a web-based graph editor and simulator application. *Production Systems and Information Engineering – ERPA Project*, Vol. 12, No. 2. https://doi.org/10.32968/psaie.2024.2.1.

[15] Wynn, M. T., van der Aalst, W. M. P., Verbeek, E., and Stefano, B. D. (2024). The IEEE XES Standard for Process Mining: Experiences, Adoption, and Revision [Society Briefs]. In: *IEEE Computational Intelligence Magazine*, Vol. 19, No. 1, pp. 20–23, Feb. 2024. https://doi.org/10.1109/MCI.2023.3333141.

[16] van der Aalst, W. M. P., Weijters, A. J. M. M. (2004). Process mining: a research 5. https://doi.org/10.1016/j.compind.2003.10.001.