UNDERSTANDING, PREDICTING, OPTIMIZING BUSINESS PROCESSES USING DATA

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by

BORIS WIEGAND

Saarbrücken, 2024

ABSTRACT

Companies face a disruptive digital transformation, which forces them to adapt their business model and innovate faster than ever before. Companies that do not transform rapidly enough risk to fall behind and fail in competition. On the other hand, changing existing business processes with complex behavior is highly risky. Analyzing process event logs promises to facilitate understanding, predicting and optimizing processes, and thus supports a successful transformation. As wrong transformation decisions impose an existential threat, understandable models in each of these steps are non-negotiable.

In this thesis, we propose novel approaches to discover inherently interpretable models from process event data. First, we explore how to summarize the actual behavior of complex processes in terms of control-flow and how event data changes throughout a process. Second, we study accurate yet interpretable event sequence prediction and learning queueing behavior. Third, we alleviate the effort of modelling optimization and AI planning problems by learning constraints from exemplary solutions.

ZUSAMMENFASSUNG

Viele Unternehmen sehen sich mit einer disruptiven digitalen Transformation konfrontiert. Diese zwingt sie dazu, ihr Geschäftsmodell schneller anzupassen und Innovationen schneller voranzubringen als jemals zuvor. Unternehmen, die sich nicht schnell genug transformieren, riskieren im Wettbewerb zurückzufallen und zu scheitern. Gleichzeitig birgt jede Veränderung an bestehenden Prozessen aufgrund ihres komplexen Verhaltens ein hohes Risiko. Die Analyse von Ereignisprotokollen verspricht, Prozesse leichter zu verstehen, vorherzusagen und zu optimieren, und unterstützt somit eine erfolgreiche Transformation. Weil falsche Transformationsentscheidungen eine existentielle Bedrohung darstellen, sind verständliche und nachvollziehbare Modelle für jeden dieser Schritte unabdingbar.

In dieser Arbeit schlagen wir neue Ansätze vor, um inhärent interpretierbare Modelle aus Prozessereignisdaten zu lernen. Zunächst untersuchen wir, wie sich das tatsächliche Verhalten komplexer Prozesse in Bezug auf ihren Kontrollfluss und sich im Prozessverlauf ändernde Ereignisdaten zusammenfassen lässt. Zweitens untersuchen wir präzise und dennoch interpretierbare Vorhersagen von Ereignissequenzen und das Erlernen von Warteschlangenverhalten. Drittens verringern wir den Modellierungsaufwand für Optimierungs- und Planungsprobleme, indem wir Einschränkungen aus Beispiellösungen lernen.

CONTENTS

1 INTRODUCTION

Companies face a continuous challenge of even further improving their business processes in terms of efficiency and effectiveness to remain competitive. The current dynamics of a disruptive digital transformation [73] force them to adapt their business model and innovate faster than ever before. Companies that do not transform rapidly enough risk to fall behind and fail in competition [56]. Existing business processes usually exhibit complex behavior of non-trivial interdependencies within a large set of process activities as well as humans, machines and business objects involved. This complexity impedes a rapid transformation and makes changing processes risky.

To mitigate the risk of transformation, process owners would ideally test any change to their process on a digital twin, i. e., an accurate digital representation of the process, before making expensive investments such as buying new machines. Due to the complex and non-trivial behavior, creating a digital twin of an existing process involves both a lot of manual effort and domain knowledge. Furthermore, manually created models are often idealized, do not fit the actual behavior well [1], and thus can lead to wrong decisions. Hence, process owners need data-driven information about the actual process behavior.

Fortunately, companies usually use information systems such as workflow engines or manufacturing execution systems to record the actual behavior of their business processes in the form of *event logs*. An event log is a set of traces, where each trace refers to a case, e. g., a single product in manufacturing or an order of a customer handled during the process. Each trace consists of numerical and categorical attributes and an event sequence. The attributes contain data about the case, e. g., the customer of an order or the type of product. Each event refers to an activity of the process such as a certain production step, and contains event data attributes such as a timestamp when the event occurred. *Process mining* [1] analyzes event logs to enable a better understanding of the actual process behavior, and thus supports identifying anomalies, inefficiencies and opportunities for automation.

By process mining, companies make data-driven decisions and create more accurate digital twins, which increases their chances to succeed in the digital transformation [51, 143].

As real-world processes exhibit complex and non-trivial behavior, so do their events logs. Furthermore, as any real-world data source, an event log usually contains noise, and we must expect it to only show an excerpt of all possible behavior in the process. Last but not least, despite the complexity of the actual process, any model we derive from an event log does not only need to be accurate, it also must be simple enough for domain experts and process owners to understand and trust the model. This makes process mining easier said than done.

In this thesis, we propose new and noise-robust solutions to discover accurate yet simple models from event logs. While, as mentioned before, the overall goal is to optimize processes, the complexity of process behavior and the resulting challenges of process mining requires process analysts to deal with multiple research problems. Before we can start thinking about optimizing, we need a thorough understanding of the actual process behavior. This means, we must comprehend the control-flow of events and how events modify the attributes of process cases. From this, we derive the first research problem as

Problem 1 (Understanding Process Behavior) *Given an event log, discover models summarizing the control-flow of events and how event data changes throughout the process.*

Understanding the past and current process behavior enables us to identify anomalies and inefficiencies, which helps us to find potential areas for improvement. To avoid inefficiencies such as bottlenecks due to overloading resources of a process, and to decide on countermeasures such as changing the schedule of cases or investing into the capacity of resources, looking into the presence and past is not enough. Hence, we define the second research problem as

Problem 2 (Predicting Process Behavior) *Given an event log, find models to predict event sequences with their activities, event data and timestamps.*

By understanding and predicting actual process behavior, domain experts and process owners identify and validate process improvements. Modelling and solving optimization problems such as scheduling of process cases supports companies in facing a continuous challenge of improving their business processes in terms of efficiency and effective-

ness to remain competitive. Since process optimization is the final goal, we define our last research problem as

Problem 3 (Optimizing Process Behavior) *Given an event log, find measurements to improve and optimize process behavior.*

Next, we discuss our contributions to these research problems.

CONTRIBUTIONS AND OUTLINE

In this section, we give an overview of our contributions and present the outline of this dissertation.

CONTRIBUTION 1 Real-world event data, such as production logs, exhibit complex behaviors. These include sequences, choices, loops, optionals, and combinations thereof that make it hard to gain insight into what is going on, and how we can improve the process. We address the first part of Problem 1 in Chapter 2, in which we summarize the control-flow of events in an event log, and thus enable an initial understanding of the actual process behavior. As a model, we define a pattern graph, in which nodes correspond to event patterns such as sequences, choices or loops, and directed edges indicate the control-flow. We need a noise-robust approach to deal with real-world event logs, and we are interested in the best trade-off between model complexity and fitting the data. Therefore, we use the Minimum Description Length (MDL) principle for model selection, by which the best model is the one with the smallest lossless description of the data. As finding the best model is a NP-hard problem, we propose our greedy algorithm PROSEQO to discover good pattern graphs in practice. Since processes with very complex behavior may require a complex MDL optimal model, we additionally propose PROSIMPLE to find pattern graphs satisfying a user-defined complexity threshold, while using MDL to minimize the loss of information.

CONTRIBUTION 2 In Chapter 3, we tackle the second part of Problem 1. Finding rules on how data changes throughout a business process is a surprisingly understudied topic. To close this gap, we propose the MOODY algorithm to discover interpretable if-then rules that explain and predict data modifications throughout a process. Through

extensive experiments on both synthetic and real-world data, we empirically show MOODY finds succinct rules, needs little data for accurate discovery, is robust to sensible amounts of noise, and thus gives valuable insight into data modifications.

CONTRIBUTION 3 To approach Problem 2 and predict event sequences for new traces in an event log from trace attribute data, we propose CONSEQUENCE in Chapter 4 to learn a sparse event-flow graph over the training sequences, and statistically robust rules that use trace attributes to determine which paths to follow. Since both the event-flow graph and the decision rules are easily human-readable, CONSEQUENCE in contrast to deep neural networks enables truly interpretable event sequence prediction. In our experiments including a case study on real-world data, we show that CONSEQUENCE indeed produces compact, interpretable and accurate models, is robust against noise and has low empirical sample complexity.

CONTRIBUTION 4 Predicting time as well as detecting and preventing bottlenecks is particularly important in service-oriented and manufacturing processes. Queueing theory is mathematically well-founded and enables interpretable sojourn and waiting time prediction. Queueing models consist of servers with limited capacity and if the service capacity is exceeded, new jobs must wait, until a free server becomes available. Creating those models, however, requires both mathematical and domain knowledge, a lot of manual effort, and often results in idealized models not fitting the actual behavior well. Therefore, we propose CUEMIN in Chapter 5 to discover interpretable queueing models from data. By extensive experiments on both synthetic and real-world data, including a case study on call center data, we show that CUEMIN, in contrast to the state of the art, finds inherently interpretable models, which explain and predict behavior of waiting line processes.

CONTRIBUTION 5 Constraint programming and AI planning are powerful tools for solving assignment, optimization, and scheduling problems. They require, however, the rarely available combination of domain knowledge and mathematical modeling expertise. Learning constraints from exemplary solutions can close this gap and alleviate the effort of modeling. Existing approaches either require extensive

Publication	Basis of
Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Mining easily understandable models from complex event logs." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Virtual Event*. 2021, pp. 244–252.	Chapter 2
Marco Bjarne Schuster, Boris Wiegand, and Jilles Vreeken. "Data is Moody: Discovering Data Modification Rules from Process Event Logs." Under submission. arXiv: 2312.14571	Chapter 3
Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Discovering Interpretable Data-to-Sequence Generators." In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*. 2022, pp. 4237–4244.	Chapter 4
Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Why Are We Waiting? Discovering Interpretable Models for Predicting Sojourn and Waiting Times." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Minneapolis, MN*. 2023, pp. 352–360.	Chapter 5
Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "What are the Rules? Discovering Constraints from Data." In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI), Vancouver, Canada*. 2024.	Chapter 6

Table 1.1: [**Publications**] List of publications with reference to chapters.

user interaction, lack to find an accurate yet concise and simple constraint set or show high noise-sensitivity. In Chapter 6, we propose the UrPiLs algorithm to find constraints from potentially noisy solutions, without the need of user interaction. Extensive experiments on constraint programming and AI planning benchmark data show UrPiLs not only finds more succinct constraints, but also is more robust to noise, and has lower sample complexity than the state of the art.

To discuss the connections between our contributions, we conduct a short case study on the event log of the rolling mill process of a steel producing company in Chapter 7. More specifically, we demonstrate how we can combine our contributions to create a discrete-event simulation of the rolling mill. In Chapter 8, we finally draw a conclusion, and discuss limitations as well as potential future work.

The chapters of this thesis are based on publications. We list in which chapter we used which publication in Table 1.1. The author of this thesis is first author in all publications, contributed to the main ideas, theoretical work, experiments, writing of the manuscript and implementation for Chapters 2, 4, 5 and 6. The second publication is based on shared first authorship. While the author of this thesis focused on writing the manuscript, both first authors equally contributed to the main ideas, theoretical work and experiments, and Marco Bjarne Schuster provided the implementation.

2

MINING UNDERSTANDABLE MODELS FROM COMPLEX EVENT LOGS

Before we can start to think about improving a process, we need to understand the events and possible behaviors in that process. Databases of event sequences encode the actual process behavior. However, existing approaches for discovering models from event sequences either cannot capture complex behavior and find over-simplified models not fitting the actual process well, or they return complex and hardly understandable models. In this chapter, we propose the Proseqo algorithm to discover accurate models from potentially noisy event sequences. We select a graph-based model summarizing the event sequences using the Minimal Description Length (MDL) principle, by which the best model gives the shortest lossless description of the data. Whenever simplicity is more important than accuracy, we propose the Prosimple algorithm to remove edges with the least loss of information, until we satisfy a user-defined graph density threshold.

2.1 INTRODUCTION

Suppose we are given a database of event sequences. How can we discover a high-quality yet easily understandable model of the data generating process? For instance, consider the event log of an industrial plant, in which sequences correspond to manufactured products and events to steps in their production. While every self-respecting plant of course has some process models, these are idealized and do not necessarily match reality in which human decisions, machine breakdowns, bottlenecks, etc. all have their effects. A high-quality model of the *actual* behavior hence does not only allow valuable insight, the chance to optimize, but also answering what-if questions.

This chapter is based on [158]: Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Mining easily understandable models from complex event logs." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Virtual Event*. 2021, pp. 244–252.

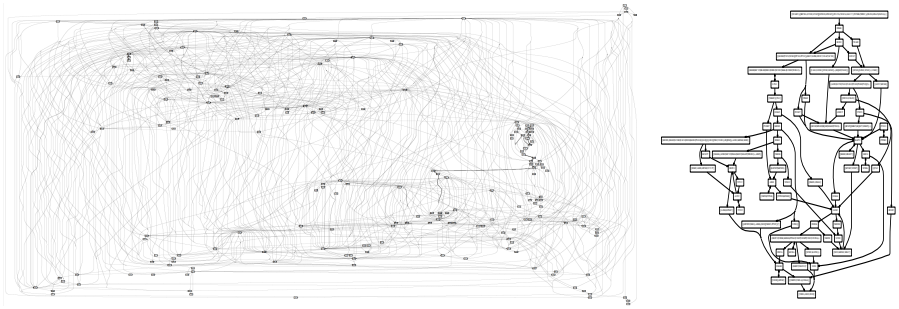


Figure 2.1: **[Example on real data]** Directly-follows-graph representing the raw data (left) versus the result of ProSimple (right) for the *Rolling Mill* production log of steel producer Dillinger.

Real-world event data exhibits complex behavior patterns, such as sequences, branches, loops, optionals, and combinations thereof. Thus, gaining insight from such data is easier said than done. Simply looking at the data does not bring us far. As an example, consider the left-hand side of Figure 2.1, where we plot the production log of steel manufacturer Dillinger as a directly-follows-graph – in which nodes correspond to events, and directed edges from a to b represent that somewhere in the log event b happened right after event a. Although we recognize some structure, the graph is heavily cluttered: we and our domain experts could barely make out the bigger picture, let alone gain any non-trivial understanding.

Making sense of event data is a classic problem, which is studied by both the pattern mining and process discovery communities. Existing solutions, however, do not satisfactorily solve the problem. While pattern mining methods [49, 63, 148] effectively discover and summarize non-trivial behavior, they only return loose collections of local structures, rather than a global model for the data. Process discovery [11, 82] on the other hand creates global models, but these tend to look similarly complex and hard-to-understand as the directly-follows-graph in Figure 2.1. We combine the best of both worlds. To this end, we propose to discover easily understandable models from complex event logs in the form of *pattern graphs*. These are directed graphs with event patterns as nodes, that together form a global model for the data.

We formulate the problem in terms of the Minimum Description Length (MDL) principle, by which we identify the best pattern graph

as the one providing the shortest description of the data. As the resulting optimization problem is NP-hard, we propose the greedy ProSeqo algorithm to discover good models from data. Starting from the directly-follows-graph, we iteratively remove nodes and edges, as well as replace them with patterns, until MDL tells us to stop. For whenever this result is still too complex, i. e., has too many edges, we propose ProSimple, which additionally removes those edges that minimally harm our score until we satisfy a user-specified threshold.

We validate our methods through an extensive set of experiments. In particular, we show they reconstruct the ground truth well with little data needed, are robust against various types of noise, and scale well. On real-world data and through a case study we confirm that, unlike the state of the art, we discover models that are easily understandable *and* fit the data well. As an example, we show the model that ProSimple discovers on the steel production data as Figure 2.1. The model is uncluttered, easy to understand, and our domain experts confirmed it matches the production process well, while providing them novel insight regarding anomalies.

Our contributions are as follows. We

(a) propose to discover pattern graphs,
(b) formalize the problem in terms of MDL,
(c) give the ProSeqo and ProSimple algorithms to respectively discover good and simple models from data,
(d) evaluate via a large set of experiments,
(e) and make all code and data available.

Next, we introduce the notation of our pattern graph model in Section 2.2. Then, we give a short introduction to the MDL principle in Section 2.3. In Section 2.4, we formalize the problem of discovering a pattern graph in terms of MDL. Afterward, we propose our heuristic discovery algorithms ProSeqo and ProSimple in Section 2.5. In Section 2.6, we give an overview of related work, before we empirically evaluate ProSeqo and ProSimple in Section 2.7. We discuss potential future work in Section 2.8 and conclude this chapter in Section 2.9.

2.2 NOTATION FOR EVENT SEQUENCES AND PATTERN GRAPHS

First, we introduce the notation we use in this chapter. As input data, we consider databases of *event sequences*. Such a database D consists of

$n = |D|$ sequences. A sequence $y \in D$ consists of $m = |y|$ events drawn from a finite length alphabet $\Omega = \{a, b, \ldots\}$. To refer to the i^{th} event in y, we write $y[i]$. We denote the empty string as ϵ.

We model event sequences by *pattern graphs*. A pattern graph is a directed and possibly cyclic graph $G = (V, E)$, where each node corresponds to a pattern. The simplest patterns are *singletons* $e \in \Omega$. Based on this base case, patterns are recursively defined as *sequences* of patterns. For example, $[a]$ expresses that event a happens, whereas $[a, b]$ models that event a happens before event b. Patterns can be *optional*, denoted by a question mark. For example, $[a, b?]$ models that b may, but does not necessarily have to happen after a. We also allow for *choices* within a pattern, and denote these by parentheses and vertical bars. With $[a, (b|c)]$, for example, we model that b *or* c happens after an a. To model repetitions, we allow for *loops*, which we denote with a plus symbol. For example, $[a, (b|c)]+$ specifies that the pattern of a followed by either b or c repeats itself.

In addition, a valid pattern graph has two special nodes, the empty string ϵ and an end-of-sequence character $\hookleftarrow$ that respectively serve as source v_s and sink v_e. Since we are specifically interested in easily understandable models, we require that every singleton event $e \in \Omega$ appears in at most one node $v \in V$.

We can then describe any given event sequence $y \in \Omega^m$ with a pattern graph G, simply by traversing G from v_s to v_e, and emitting events according to the nodes that we visit. To determine what path to take and which choices to make, we have to read codes from the *code stream* C that corresponds to how the model explains, or *covers* the sequence. Conceptually, we can split C into two parts: the *model stream*, C_m, which encodes how to traverse the model, and the *disambiguation* stream, C_d, which encodes the necessary details to decode the sequence.

We both give an example sequence y and the covers of it for three different models M_1, M_2, M_3 in Figure 2.2. The first model, M_1, consists of a graph over just singletons. To decode the data, we start at source node v_s and read the first code from C_m. This is a $\boxed{\rightarrow}$ code, which means we emit the current symbol of the current pattern, i. e., ϵ, and move on. As v_s has only one outgoing edge we unambiguously arrive at node a. We read the next code from C_m, which tells us to emit a, and move onward. This time we can either go to node b or to node c. To determine which path to take, we read the next code from the

disambiguation stream, C_d. As we read the code corresponding to the path to b, we take this path and carry on in similar fashion, until we arrive at the sink v_e and have decoded y without loss.

In the second example, we start again at v_s, emit ϵ after reading [→], and arrive at a *sequence* pattern. We emit its first element (a) after reading the next [→], and arrive at its second element ($b|c$). To decide whether to emit b or c, we read from C_d. To determine which edge to follow, we read from C_d, and arrive at e. We emit e and follow the edge to *loop pattern* $d+$. We emit the looped pattern (here, d) as often as we read a [↺] from C_d, and one final time when we read a [×]. As there is only one outgoing edge from $d+$, we unambiguously arrive at *optional* pattern $f?$. To decide whether to emit f, we read a [+] or [−] from C_d. We then unambiguously arrive at g, which we emit, and are done.

In the third example, the pattern graph consists of a single large pattern. We decode the a and b just like above, but then encounter a [?] code in C_m. This code tells us that the next event is not captured by the model. We decode this event by reading the code for a singleton event $e \in \Omega$ from C_p, which is the code for e, which we then emit. Next, after decoding the two d's via the loop, we encounter a [↷] code in C_m, which means that we move to the next element but do not emit. The final [→] takes care of g, and we have again losslessly decoded y.

2.3 THE MINIMUM DESCRIPTION LENGTH PRINCIPLE

For a given dataset, we want to find a pattern graph that both fits the data well and is easily understandable by being as simple as possible. Additionally, since real-world data usually contains noise, we need a robust model selection criterion. Therefore, we use the Minimum Description Length (MDL) principle [52, 123] for model selection. MDL identifies the best model as the one with the shortest lossless description of the given data. Formally, given a set of models $\mathcal{M}$, the best model is defined by $\arg\min_{M \in \mathcal{M}} L(M) + L(D \mid M)$, in which $L(M)$ is the length in bits of the description of M, and $L(D \mid M)$ is the length in bits of the data encoded with the model. This form of MDL is known as two-part or crude MDL. Although one-part or refined MDL provides stronger theoretical guarantees, it is only computable in specific cases [52]. Hence, we use two-part MDL.

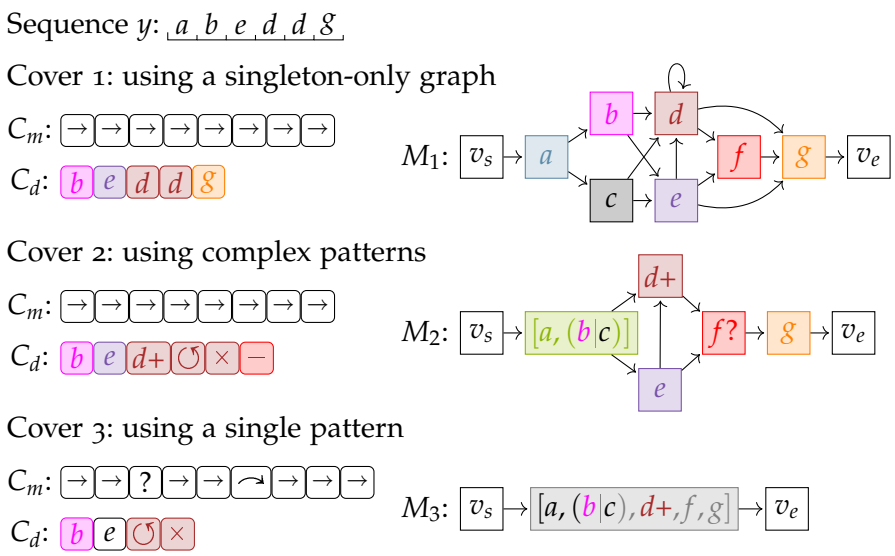


Figure 2.2: **[Cover toy example]** Sequence y as covered by three different pattern graph models M_1, M_2, M_3.

Note that MDL requires the compression $(L(M) + L(D \mid M))$ to be lossless in order to allow for fair comparison between different $M \in \mathcal{M}$, and that we are only concerned with code lengths, not actual code words. Since we measure the encoded length in bits, all logarithms are to base 2, and we use $0 \log 0 = 0$.

Assume we want to compute the encoded length of a code stream C, which is the sum of the encoded lengths of the individual codes in that stream. According to Shannon Entropy, the length in bits of the optimal prefix-free code for $x \in C$ is $-\log P(x)$, which follows the intuition that the more frequent a code the shorter its encoded length should be. However, this requires knowledge of the distribution of codes beforehand. Prequential codes [52] are asymptotically optimal *without* having to know the distribution of messages. The idea is remarkably simple. Starting with a uniform distribution, we update the counts after every received message, which means we have a valid probability distribution at every point in time, which permits optimal prefix codes [29].

To encode how often the model contains a certain structure, we must encode natural numbers, where we do not know an upper bound beforehand. The MDL-optimal encoding for integers $z \geqslant 1$ [124] is defined as $L_{\mathbb{N}}(z) = \log^* z + \log c_0$, where $\log^* z = \log z + \log\log z + \ldots$,

and we sum only the positive terms, and $c_0 = 2.865064$ is set such that we satisfy the Kraft-inequality – i. e., ensure it is a lossless code. Whenever we must compute the encoded length of a real number $z \in \mathbb{R}$, we encode z up to a user-specified precision p by the smallest integer shift s such that $z \cdot 10^s \geqslant 10^p$. We then encode shift, shifted digit and sign, i. e., $L_{\mathbb{R}}(z) = L_{\mathbb{N}}(s) + L_{\mathbb{N}}(\lceil z \cdot 10^s \rceil) + 1$ [92].

2.4 MDL FOR PATTERN GRAPHS

The examples in Section 2.2 illustrate how we can model event sequences y using a pattern graph M, and, importantly, in what contexts to expect what codes. We will now formalize these intuitions into a lossless MDL score, such that we can identify the best model $M^* \in \mathcal{M}$ for given data D. We start by defining $L(D \mid M)$, the encoded cost of a given sequence database D for a given model M.

2.4.1 *Encoded Length of the Database*

At a high level, the encoded length of the data given a model is

$$L(D \mid M) = L_{\mathbb{N}}(n) + L(C_m) + L(C_d) \,,$$

where we first encode the number n of the sequences in D, and then proceed to encode the code streams C_m and C_d. To avoid any arbitrary choices in the model encoding, we use prequential codes (see Section 2.3) to encode the model and disambiguation streams. To make maximum use of the available information, we should use codes that are conditioned both on *where in the model* and *where in the data* we are. For the model stream C_m, which is a sequence over $\{\rightarrow, \rightsquigarrow, ?\}$, we have

$$L(C_m) = -\sum_{i=1}^{|C_m|} \log \frac{\mathrm{usg}_i(C_m[i] \mid e_i) + \epsilon}{\sum \mathrm{usg}_i(\,\cdot \mid e_i) + \epsilon} \,,$$

where we encode whether we have to emit, skip, or fill a gap, conditioned on what event $e_i \in \Omega$ we encoded right before message $C_m[i]$. Initializing with standard choice $\epsilon = 0.5$, and $\mathrm{usg}_0(\cdot|\cdot) = 0$, we increment the *usage counts* upon receiving messages.

We encode the disambiguation stream C_d analogously. For the definition of its encoded length it is helpful to consider it as three independent parts, namely stream C_p of pattern codes that we expect after

reading a ⤳ or →, stream C_g of codes we expect after reading a ?, and stream C_s of codes that we need to disambiguate loops, optionals, and choices. That is, $L(C_d) = L(C_p) + L(C_g) + L(C_s)$.

The codes in C_p refer to nodes in the model, and which nodes $v \in G$ are possible depends on that node v_i we are currently at. We have

$$L(C_p) = -\sum_{i=1}^{|C_p|} \log \frac{\mathrm{usg}_i(C_p[i] \mid v_i) + \epsilon}{\sum \mathrm{usg}_i(\,\cdot \mid v_i) + \epsilon} \,.$$

The messages in C_g correspond to singletons $e \in \Omega$, but only those that we cannot directly reach from current node v_i – we are after minimal descriptions, after all. We hence have

$$L(C_g) = -\sum_{i=1}^{|C_g|} \log \frac{\mathrm{usg}_i(C_g[i] \mid v_i) + \epsilon}{\sum \mathrm{usg}_i(\,\cdot \mid v_i) + \epsilon} \,.$$

Finally, the messages in C_s are dependent both on the last decoded symbol e_i and what node v_i we are at, i. e.,

$$L(C_s) = -\sum_{i=1}^{|C_s|} \log \frac{\mathrm{usg}_i(C_s[i] \mid e_i, v_i) + \epsilon}{\sum \mathrm{usg}_i(\,\cdot \mid e_i, v_i) + \epsilon} \,.$$

This gives us a lossless encoding of a database.

2.4.2 *Model Encoding*

Next, we define how we encode a model M in bits. Because we make use of prequential codes in encoding the data, encoding the model is relatively straightforward. Formally, we have

$$\begin{aligned} L(M) = L_{\mathbb{N}}(|\Omega|) + \log(|\Omega| + 1) + \sum_{v \in V} (\log |\mathcal{T}| + L(v)) \\ + \log(|V|^2 + 1) + \log \binom{|V|^2}{|E|} \,, \end{aligned}$$

where we first encode the size of the alphabet. This gives an upper bound that we use to encode the number of pattern nodes in G (i.e. excluding v_s and v_e). We then encode the type ($\mathcal{T} = \{\mathit{singleton}, \mathit{sequence}, \mathit{choice}, \mathit{loop}, \mathit{optional}\}$) and content of each pattern node. Finally, we encode the graph among them by first encoding the number of edges,

and then their layout. This we do via a data-to-model code [87], which is an index over a canonically ordered set of all directed graphs of $|V|$ nodes and $|E|$ edges. The only further details left are how to encode the different types of nodes. Singletons are the base case, with

$$L_{\text{singleton}}(v) = \log |\Omega| \ .$$

Optionals and loops are a wrapper for one subpattern v', i. e.,

$$L_{\text{optional}}(v) = L_{\text{loop}}(v) = \log |\mathcal{T}| + L(v') \ ,$$

whereas sequences and choices consist of up to $|\Omega|$ subpatterns, i. e.,

$$L_{\text{sequence}}(v) = L_{\text{choice}}(v) = \log |\Omega| + \sum_{v' \in v} (\log |\mathcal{T}| + L(v')) \ ,$$

by which we have a lossless encoding of a model M.

2.4.3 *Formal Problem Definition*

With the above definitions, we now formally define our problem.

Minimal Pattern Graph Problem *Let D be a sequence database over alphabet Ω, find the minimal pattern graph $M \in \mathcal{M}$ and cover C of D, such that the total encoded cost $L(M) + L(D \mid M)$ is minimal.*

The minimal pattern graph problem is a rather difficult problem. For a given database Ω, there exist exponentially many models M, and the score does not exhibit trivial structure, such as submodularity or monotonicity, that we can exploit for efficient search. Moreover, there exist exponentially many covers for a given model, and finding the optimal cover is equivalent to aligning Petri nets and sequences which has known to be NP-hard [83]. Hence, we resort to heuristics.

2.5 ALGORITHM

To find good solutions to the Minimal Pattern Graph Problem in practice, we split the problem, and propose greedy algorithms to discover a good cover of the data for a given model, and for iteratively discovering good models from data. We discuss these algorithms in turn.

Algorithm 1: COVER

input : sequence y, model M, cover C
output: cover C

$v \leftarrow v_s$;
foreach event $e \in y$ **do**
  if pattern v can cover e **then**
    add codes to C to encode e with v;
  else if $\exists$ pattern $u \in M : e \in u$ **and** $\exists$ path ϕ from v to u **then**
    add codes to C to skip remainder of v;
    add codes to C to skip all $w \in \phi$ up to u;
    add codes to C to encode e with u;
    $v \leftarrow u$;
  else
    add codes to C to encode e as [?];
return C

2.5.1 *Computation of a Good Cover*

To compute $L(D \mid M)$ we need a good cover C for a given sequence $y \in D$. As computing the optimal cover is NP-hard, we take a greedy approach. Intuitively, we want to follow the model as much as we can, and hence want to maximize the number of [→] codes in C_m. To do so, we iteratively cover the events in y with patterns in the model, by which we ensure a linear runtime regarding the sequence length $|y|$.

We give the pseudocode as Algorithm 1. In a nutshell, whenever there exists a pattern $u \in M$ that can cover the next event $e \in y$, we see if there exists a path from the last-used pattern v to u, such that we minimize the number of [↝] codes in C_m. Whenever there exists no such a pattern, or no such path, we cannot use a pattern to cover e, and instead encode it with a [?] in C_m and a code for e in C_d.

Since every edge has equal cost, finding the shortest path between two pattern nodes reduces to a breadth-first-search with runtime complexity $O(|V| + |E|)$. As we allow an event to be part in at most one pattern node, the maximal number of nodes in the graph is bounded by $|\Omega|$. Therefore, covering all n sequences in D with sequence length m has runtime complexity $O(n \cdot m \cdot (|\Omega| + |E|))$. The cover encoding

Algorithm 2: PROSEQO

input : sequence database D
output: model M for D

```
1 M ← initialize with trivial model for D;
2 Q ← create transformations based on M;
3 while Q is not empty do
4 |   t ← pop first element of Q;
5 |   if L(D, t(M)) < L(D, M) then
6 |   |   M ← t(M);
7 |   |   Q ← update Q based on M;
8 return M;
```

using prequential codes is order-invariant, hence, the cover algorithm is sequence-order-invariant.

2.5.2 *Discovering Good Models with* PROSEQO

To discover good models we propose the PROSEQO algorithm, which greedily improves the current model top-down until convergence. We give the pseudocode as Algorithm 2. We start from the directly-follows-graph (ln. 1), an overfit pattern graph where all singleton events are nodes, i. e., $V = \Omega$, and $(v, u) \in E$ whenever in any sequence in D, u happens right after v. Then, we generate candidate transformations of the current model (described below), and store these in a priority queue (ln. 2). We evaluate the candidates in descending order of their individual gain, and update model and candidate transformations whenever we improve over the current model (ln. 3 – 7).

As candidate model transformations we consider the following three types. We consider *removing edges*, and regard every edge $(v, u) \in E$ whose removal does not cut the path from v_s to v_e as a candidate. We consider *removing nodes*, and consider every node $v \in M$ whose removal does not cut the path from v_s to v_e as a candidate. Finally, we consider *growing patterns*, by which we replace current patterns $v \in M$ with a new pattern v'. We create a candidate sequence $[a, b]$ for every edge $(a, b) \in E$. Nodes with a common predecessor generate choice candidates, loop candidates are created from loops in the pat-

tern graph, and optional patterns are from nodes whose predecessors are also connected to ancestors.

The main bottleneck is the computation of the cover both during evaluation and to rank candidates, which requires a pass over the whole sequence database. To gain efficiency, we do not re-generate the entire candidate set Q in every iteration, but rather update it: we remove those transformations from Q that are no longer possible, and only add and compute the gains for new candidates – i. e., we do not re-compute gains of previously generated transformations.

The number of generated candidates increases with the number of edges in the pattern graph. A maximally dense graph with $|E| = |\Omega|^2$ generates $|E|$ edges, $|E|$ sequences, $\binom{|\Omega|}{2}$ choices, $|\Omega|$ optionals and $|\Omega|$ (self-)loops. In the worst case, each generated transformation improves our score and leads to $O(|\Omega|)$ new candidates, which makes $O((|\Omega| + |E|) \cdot |\Omega|)$ candidates in total. Considering the repetitive cover computation, ProSeqo has a runtime complexity of $O\left(n \cdot m \cdot (|\Omega| + |E|)^2 \cdot |\Omega|\right)$. This means, ProSeqo scales linearly with the number of sequences n and sequence length m.

2.5.3 *Discovering Simple Models with* ProSimple

What if the MDL-optimal model, or its approximation by ProSeqo, is still too complex for a human? This happens if the underlying data generating process is inherently complex and any simplification results in a huge loss of information. In such a setting, how can we discover models that are easily understandable, while ensuring they do explain the data as well as possible?

The main complexity of a process model, as confirmed by our domain experts, comes from its number of edges: it is hard to keep track of all possible paths in a directed graph with many edges. This suggests that we can ensure understandability by controlling the number of edges in a model. How can we do so in a principled manner? Let $\mathcal{M}^{(r)} \subseteq \mathcal{M}$ be the set of all models over Ω that have a degree ratio $|E|/|V|$ of at most r. It is trivial to re-write the Minimal Pattern Graph problem accordingly: we are now after that model $M^* \in \mathcal{M}^{(r)}$ that minimizes the total encoded length.

We build upon ProSeqo to find a good solution for this new problem. We give the pseudocode as Algorithm 3. First, we simply run

Algorithm 3: PROSIMPLE

input : sequence database D, degree ratio r
output: model M for D with degree ratio $\leqslant r$

1 $M \leftarrow$ PROSEQO(D);
2 **while** $\frac{|E|}{|V|} > r$ **do**
3 $\quad e^* \leftarrow \arg\min_{(u,v)\in E \mid \deg^-(v)>1 \wedge \deg^+(u)>1} L(D, M \ominus (u,v))$;
4 $\quad M \leftarrow M \ominus e^*$;
5 $M \leftarrow$ PROSEQO(D, M);
6 **return** M;

PROSEQO on D, which gives us a $M \in \mathcal{M}$ (ln. 1). If M satisfies the degree ratio threshold r, we are done. If it does not, we iteratively remove those edges from the model that 'harm' the MDL score least, until we satisfy threshold r (ln. 2–5). To retain connectivity in the graph, we do not remove edges, which would lead to dead ends or unreachable nodes. After removing one, or multiple edges it is possible that PROSEQO can further optimize the MDL score – for example by applying patterns or removing nodes. While ideally we would do this in every iteration, for efficiency we do this only once, after we ensured $M \in \mathcal{M}^{(r)}$ (ln. 5). We refer to this method as PROSIMPLE.

In each iteration, we have up to $|E|$ edges, which we can remove. For each edge, we compute a cover to evaluate the score after removing this edge. Since we have up to $|E|$ iterations, the runtime complexity of the edge removal part of PROSIMPLE is $O(|E|^2 \cdot n \cdot m \cdot (|\Omega| + |E|))$.

2.6 RELATED WORK

Discovering structure from event sequences is a classic research topic [6, 91]. Earlier proposals focused on efficient discovery of all frequent subsequences with or without gaps [108, 164], resulting in overly many and highly redundant patterns: the pattern explosion. Attention hence shifted to reducing redundancy via closures [145, 146], statistical testing [109, 144], or a pattern set mining approach [48, 147].

Subsequences and serial episodes can model interesting behavior, but only have limited expressive power. There exist proposals that can additionally model parallel behavior [146], choices [63], or periodicity

[49], but each only extends in one direction. Petri nets [110] can jointly model loopy, alternative as well as concurrent behavior, but existing approaches [148] rely on frequency-based interestingness measures, and hence suffer from the pattern explosion.

While pattern mining is great for discovering interesting local behavior, the above methods only result in a set of disconnected patterns, rather than a coherent model that explains the generating process in terms of patterns. Towards this goal, the closest related field is process discovery, which is a subfield of process mining [1], and deals with the extraction of process models from event logs. The mainstream of process discovery algorithms infers model structure from the directly-follows-graph of the log [11, 82, 156]. The application of these state-of-the-art approaches on complex real-world event logs leads, however, either to highly complex models that are difficult to understand or models that over-generalize and obtain only low precision [1, 11].

In contrast to the above, with ProSeqo we discover compact, non-redundant, and coherent models that explain the process behind the data in terms of rich patterns. We do not only consider removing edges (i.e. directly-follows-relations) but also nodes (symbols) to reduce model complexity. Moreover, if desired by the user, we provide an easily interpretable hyperparameter that allows to further reduce model complexity in a principled way.

2.7 EXPERIMENTS

In this section, we empirically evaluate ProSeqo and ProSimple on synthetic and real-world data. To ensure reproducibility, we make both code and synthetic data generators publically available for research purposes.[1] We executed all experiments single-threaded on an Intel i7-6700 CPU, with 16 GB of memory, running Windows 10. We report wall-clock running times.

We compare to three state-of-the-art methods. SplitMiner [11] and IMf [82] are process discovery algorithms, whereas Squish [63] discovers models that consist of sequential patterns. ProSeqo and Squish have no hyperparameters. We run IMf and SplitMiner with the parameters set as recommended by the authors [11, 82].

1 https://eda.rg.cispa.io/prj/proseqo/

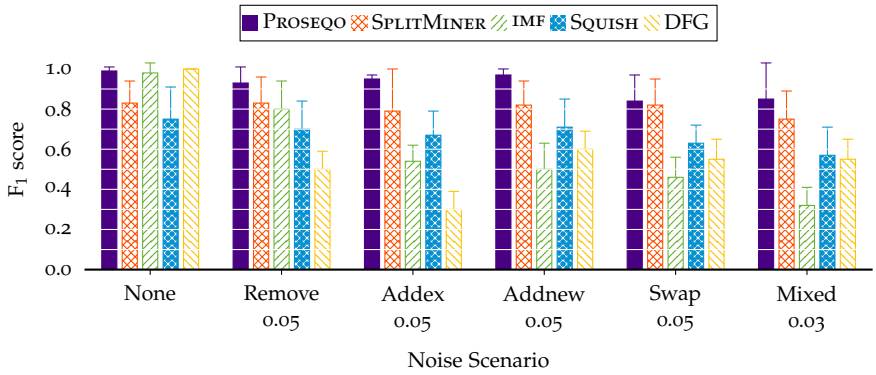


Figure 2.3: **[Proseqo handles different types of noise]** Mean F_1 scores on directly-follows-relations for respectively no, 5% Remove, 5% Addex, 5% Addnew, 5% Swap, and 3% of all noise types simultaneously, for Proseqo, SplitMiner, IMF, Squish, and the trivial directly-follows-graph that Proseqo departs from. Error bars indicate standard deviation.

2.7.1 *Synthetic Data*

First, we consider synthetic data, where we both know and can control the ground truth. We start with a sanity check, in which we evaluate on data without structure. To this end, we sample 100 sequences of length ten uniformly at random from $\Omega = \{a_0, \ldots, a_9\}$, and add a fixed START and END symbol to all sequences. Proseqo is the only method recovering the ground truth, returning the model $[\text{START}, (a_0|a_1|a_2|a_3|a_4|a_5|a_6|a_7|a_8|a_9)+, \text{END}]$. Squish almost recovers the ground truth by returning the singleton-only model; however, it additionally outputs the sequential pattern $[\text{START}, a_1]$. While IMF correctly identifies that all ten activities can happen in arbitrary order, it explicitly does not allow any activity to happen more than once, which contradicts the data generation process. SplitMiner overfits the data and returns a model with 11 nodes and 18 edges.

Next, we examine how well Proseqo can reconstruct a non-trivial model with different types and levels of noise. We generate ground-truth models using the generator proposed by Jouck et al. [65] with the following parameters: min $= 40$, mode $= 50$, max $= 60$, sequence $= 0.5$, choice $= 0.4$ and loop $= 0.1$. We convert the resulting process trees into pattern graphs and sample sequence databases using ran-

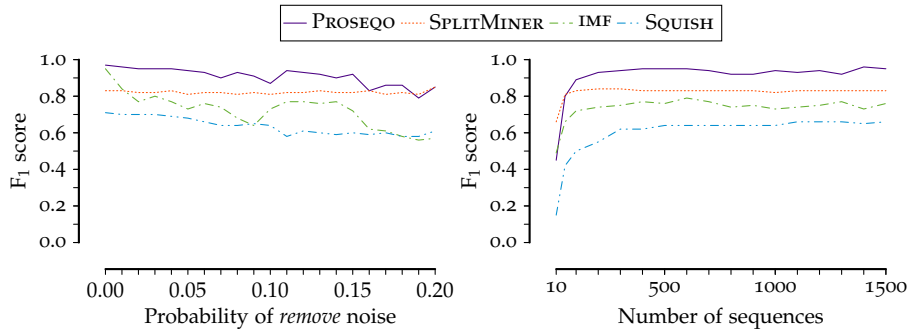


Figure 2.4: **[Proseqo is noise-robust]** F_1 scores for varying amounts of Remove noise (left) and for varying number of sequences with 5% Remove noise (right).

dom walks. To add noise, we apply the following noise models: *Remove* simulates missed recording of events, i. e., for each event in the database we remove it with a given probability. *Addex* simulates recording real events that did not happen, i. e., for every event in the database, with a given probability, we insert a random event $e \in \Omega$. *Addnew* does the same, but inserts a fixed *noise* event $n \notin \Omega$. *Swap* simulates events recorded in wrong order, i. e., for each neighboring pair of events, we swap their order with a given probability.

As a metric of success, we consider the F_1 score measured over correctly identified edges between events. We consider the average result per method over 20 independently generated models, and for each generate a database D of 1 000 sequences each. We plot the results for all four methods, as well as those for the trivial model that Proseqo starts from, in Figure 2.3. We see that Proseqo performs best: it returns near-perfect models when there is no noise, obtains above 0.94 scores for 5% Remove, Addex, or Addnew noise, and still reaches an F1 of above 0.85 when we apply all four noise types simultaneously at 3% each. Our competitors fare less well. IMF only performs well when there is no noise, while Squish and SplitMiner tend to return underfitting models with low recall and high precision.

Next, we evaluate robustness against varying levels of Remove noise, and for varying number of samples $y \in D$ for a fixed amount (5%) of Remove noise. We generate 20 models per setting and report average F_1 scores in Figure 2.4. We see Proseqo performs favorably, its F_1 score only dropping slightly for up to 20% noise, whereas it only needs 100

Data	n	n_u	min $\|y\|$	avg $\|y\|$	max $\|y\|$	$\|\Omega\|$
Permits	1199	1170	4	46	103	291
Loans	31509	5623	9	17	56	26
Purchases	2000	365	3	9	317	28
Rolling Mill	1000	988	13	28	53	191

Table 2.1: **[Statistics of real-world data]** Number of sequences $n = |D|$, number of unique sequences n_u, number of unique events $|\Omega|$, as well as minimal, average and maximal sequence length $|y|$, and size of the event alphabet $|\Omega|$ in four real-world datasets.

sequences to converge. SplitMiner is in second place, whereas IMF and Squish trail by a wide margin. As they perform sub-par, we do not consider IMF and Squish in the remainder of this section.

2.7.2 *Real-World Data*

Next, we evaluate on four real-world event logs. Three stem from the publically available Business Process Intelligence Challenge. *Permits* [34] contains event data for building permit applications of a Dutch municipality. *Loans* [35] corresponds to the recording of a loan application process of a Dutch financial institute. *Purchases* [36] is data on the purchase order handling of an un-named company. As the above experiments showed low sample complexity for all methods, we consider a random sample of 2 000 out of its in total 200 000 sequences. Last, but not least, we consider the *Rolling Mill* production event log of the steel producer Dillinger. We give their base statistics in Table 2.1.

Because we do not know the ground-truth model for real-world data, we cannot compute F_1 scores for these datasets. Instead, we hence report on how complex the models are, and how well they explain (fit) the data. We measure model complexity in terms of number of nodes, number of edges, and *structuredness* $S = \max\{0, \frac{|\Omega|-|V_s|}{|\Omega|-1}\}$, with V_s being the set of nodes after reducing the graph with perfectly matching sequences, choices, optionals and loops. The more such patterns a graph contains the easier it is to understand. A perfectly structured model can be reduced to one single node and has $S = 1$.

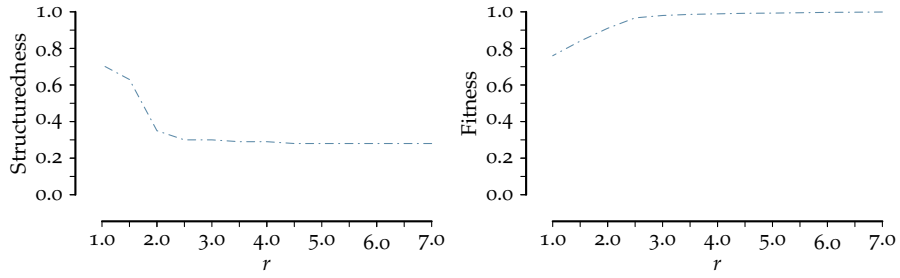


Figure 2.5: **[Prosimple with varying degree ratio]** Structuredness (left) and Fitness (right) for Prosimple with varying degree ratio r on the *Rolling Mill* dataset. For $r = 7.0$, Prosimple is equivalent to Proseqo on this dataset.

	SplitMiner			Proseqo			Prosimple, $r = 1.5$		
Data	$\|V\|$	$\|E\|$	t	$\|V\|$	$\|E\|$	t	$\|V\|$	$\|E\|$	t
Permits	683	1441	33s	181	33071	50h	42	49	51h
Loans	53	71	1s	22	135	2.5h	5	5	2.5h
Purchases	68	135	4s	26	163	4m	6	6	4m
Rolling Mill	427	817	9s	135	934	2.3h	68	101	2.5h

Table 2.2: **[Results on real-world data]** Given are the number of nodes $|V|$ and edges $|E|$ of the discovered models by SplitMiner, Proseqo, and Prosimple with $r = 1.5$ on four real world datasets, and discovery runtime t for all methods.

We measure fitness by converting the models to Petri nets [110] and computing the established fitness score for Petri nets [17]. We consider both Proseqo and Prosimple. For the latter, we focus on simple models and set the degree ratio r to 1.5, which means that on average every node in the model has 1.5 outgoing edges. A degree ratio of $r = 1.5$ gives a good trade-off between gain in structuredness and loss of fitness, as we can see in Figure 2.5. We compare to SplitMiner. We run each of the methods, and give the results in Figure 2.6 and Table 2.2.

We first consider Figure 2.6 and see Proseqo provides models that fit the data best, SplitMiner discovers well-fitting but complicated models, and that at cost of some fit, Prosimple returns by far the simplest models. If we investigate the quantitative results in Table 2.2, we

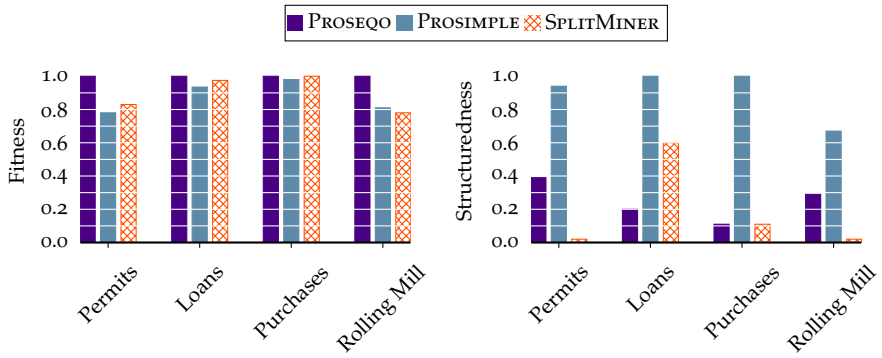
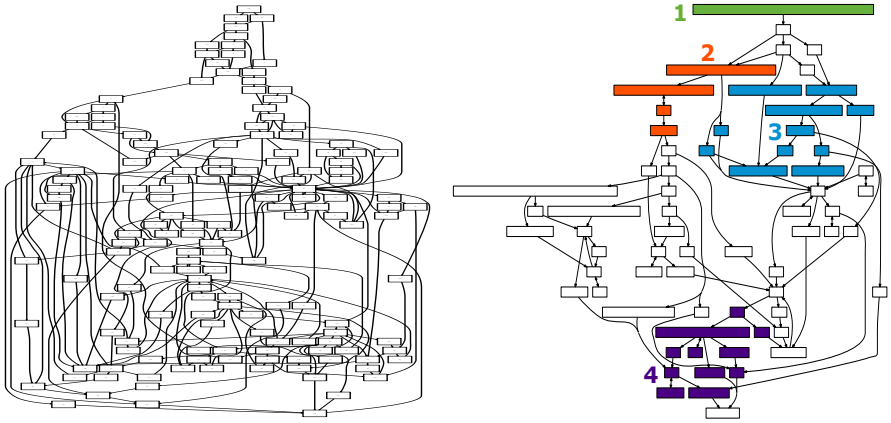


Figure 2.6: [**Prosimple mines well-fitting yet understandable models**] Fitness (left, higher is better) and structuredness (right, higher is better) for PROSEQO, PROSIMPLE with $r = 1.5$ and SPLITMINER on four real-world datasets.

again see that PROSIMPLE creates by far the simplest models in terms of number of nodes and number of edges in the graph. Computing covers between pattern graph candidates and data is an expensive operation. Therefore, runtime is significantly higher for PROSEQO and PROSIMPLE compared to SPLITMINER. Still, PROSEQO and PROSIMPLE discover useful models of complex real-world data in a reasonable amount of time.

The importance of model simplicity becomes even more clear when we visually inspect the models. In Figure 2.7, we show the models discovered by PROSIMPLE and SPLITMINER for the *Rolling Mill* data. While the latter is already much more structured than the trivial directly-follows-graph, it is still a bowl of spaghetti that was not interpretable for our experts; they complained it was too hard to follow the control-flow. The result of PROSIMPLE is much easier to understand: our experts confirm that the model as a whole, as well as the patterns therein are semantically meaningful. That is, the model gives a high-level overview of the production process, and the pattern nodes give detailed insight into what production steps are executed in what order and context. Now, we will have a closer look on the rolling mill process and how PROSIMPLE enables an understanding of it.



(a) SplitMiner (b) Prosimple

Figure 2.7: **[Prosimple discovers detailed yet easily understandable models]** Model discovered by SplitMiner (left) and Prosimple (with $r = 1.5$, right) on the *Rolling Mill* dataset. In the Prosimple model, we highlight four essential parts of the production process (hot zone, scissors, torches, and quality checks).

2.7.3 *Case Study*

Using the Prosimple model in Figure 2.7, we can highlight four parts of the Dillinger rolling mill that are understandable to everyone. The process starts with so-called *slabs*, cast steel cuboids. The green source node (part 1) contains a nested sequence of eight low-level activities that correspond to the *hot zone* of the rolling mill. The slabs are either heated up in so-called pusher-type furnaces or in bogie-hearth furnaces, such that they are soft enough to get rolled to *mother plates* at two rolling stands. Some plates get special treatment and are cooled down with water. After the hot zone, the mother plates are cut into the plates ordered by the customers. This either is done using large scissors for soft and thin enough (part 2) plates, or with cutting torches if the plates are too hard and thick (part 3). The bottom part (part 4) mainly consists of quality checks and corrections.

Not only does the model by Prosimple give a high-level overview of the control-flow of the rolling mill process, it also contains details on low-level behavior allowing for, among others, an anomaly analysis. First, our domain experts questioned some of the modeled behavior

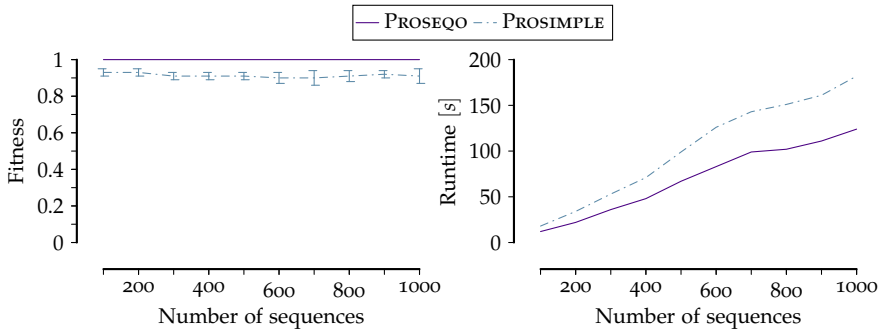


Figure 2.8: [**Proseqo and Prosimple scale favorably**] Scalability of Proseqo and Prosimple on *Loans* in terms of average fitness over ten runs with standard deviation (left) and runtime in seconds (right) for varying subsample size.

as violating their expectation. These all turned out to be due to rare but re-occurring deviations from the normal behavior such as machine failures, as well as alternative routing caused by high workload in parts of the plant. For example, if needs be, thicker plates run through parts of the rolling mill specialized for thinner plates and vice versa.

In a second step, we showed sequences deviating most from the found model to our domain experts. Here, we identified anomalies corresponding to additional and repetitive work necessary to meet certain quality goals. Better monitoring of these cases can support improvement of the process in terms of production efficiency, product quality and reduced production loss.

2.7.4 *Scalability*

Finally, we report in Figure 2.8 on the performance and runtime of Proseqo and Prosimple dependent on the size of a subsample of the *Loans* data. Both methods show low sample complexity by achieving stable fitness on the whole dataset with only 100 out of 31509 sequences, and their runtime scales linearly in the number of sequences.

2.8 DISCUSSION

Although both PROSEQO and PROSIMPLE work well, we see many interesting directions for future work. First of all, MDL is not a magic wand: the encoding we propose includes choices. Different choices may lead to different, and possibly better, models. While we consider a rich set of patterns, it is easy to think of structure such as parallel behavior that we currently cannot succinctly capture. On the topic of discovering better models, it is likely worthwhile to incorporate bottom-up approaches from pattern mining for identifying promising parallel and choice behavior, rather than the pure top-down approach we currently take. Extending pattern graphs such that one event can be part of more than one node could also lead to better models. We pick up the idea of a bottom-up search and using events in multiple nodes in Chapter 4. On the topic of scalability, it is worthwhile to investigate, whether it is possible to formalize accurate and easy-to-compute estimates [45], such that we can be more informed during the search.

We see many applications of pattern graphs. Describing the expected behavior of a process, we see them useful for explainable anomaly detection. Other potential applications include optimizing planning, and especially towards simulation and answering what-if questions. To this end, the formulation of model, problem, and inference would ideally have to be redone in pure counterfactual terms [107].

2.9 CONCLUSION

We studied the problem of discovering accurate yet easily understandable models from complex event sequence data. To this end, we proposed to model data using directed pattern graphs, where the nodes summarize complex behaviors such as sequences, choices, loops, optionals, and combinations thereof in easily interpretable terms. We formulated the problem in terms of the Minimum Description Length (MDL) principle. As the search space is exponentially sized, and determining the quality of a model is already NP-hard, we proposed the greedy PROSEQO algorithm to discover good models in practice. For whenever understandability is of primary, and accuracy of secondary importance, we propose the PROSIMPLE algorithm that further prunes the result of PROSEQO up till an easily interpretable user-specified pa-

rameter is satisfied. Experiments on both synthetic and real-world data validate that our approaches work well in practice, and outperform the state of the art by a margin.

3

DISCOVERING DATA MODIFICATION RULES

Process discovery methods such as PROSEQO and PROSIMPLE from the previous chapter provide an initial understanding of events and potential behaviors within a process. Like most discovery methods, however, these two focus on finding patterns in the activities of event sequences, and thus neglect how event attribute data in the form of categorical and numerical variables changes throughout a process. In this chapter, we propose a novel method to complete the picture and mine accurate yet interpretable if-then rules of data modification rules.

3.1 INTRODUCTION

Given a process event log, process mining [1] provides a better understanding of the underlying process and enables downstream tasks such as monitoring, anomaly detection, simulation, and optimization. Existing work, however, focuses on discovering patterns of event activities, but neglects how event attribute data changes during the process. For instance, a textile company may have an ordering process with the activities *Request*, *Place*, *Delay* and *Receive*. Process discovery algorithms [11, 141] only infer a graph of event activities, where nodes refer to activities and edges visualize the flow of execution. However, how data changes over time is crucial for understanding the process as we show in Figure 3.1. In our textile example, we only know price and delivery date of an order after placing the order. Delaying an order, e. g., due to a shortness of supplies, changes the delivery date.

Surprisingly, data has only been used to predict the occurrences of event activities or the outcome of processes [150, 159]. To the best of our knowledge, none of the existing work mines interpretable rules for data modifications, and related work does not satisfactorily transfer

This chapter is based on [134]: Marco Bjarne Schuster, Boris Wiegand, and Jilles Vreeken. "Data is Moody: Discovering Data Modification Rules from Process Event Logs." Under submission. arXiv: 2312.14571

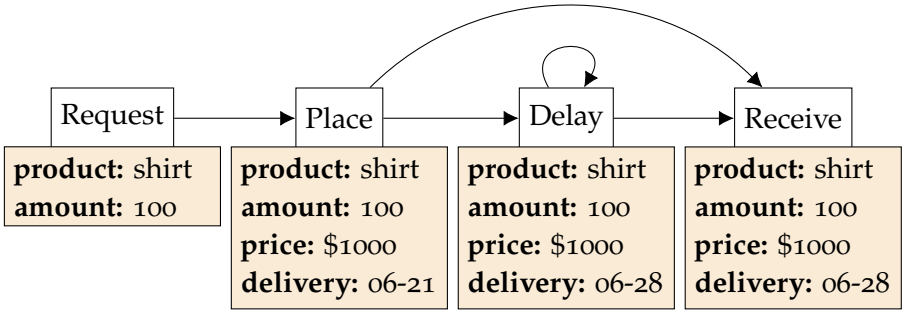


Figure 3.1: **[Processes change data]** Exemplary process for ordering textiles with activities *Request*, *Place*, *Delay* and *Receive*. Arrows indicate the flow of events. Further, we show how the data of an exemplary order changes throughout the process.

to our problem. Subgroup discovery [118] and rule-based prediction methods [163] lack the ability to model the sequential dependencies present in event logs, and thus lead to unsatisfactory results with limited insight into the process behavior.

Given an event log, we are interested in finding accurate yet succinct and interpretable if-then rules how the process modifies data. To this end, we formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we choose the model with the best lossless description of the data. Additionally, we propose our method Moody, which is short for Modification rule Discovery, to efficiently search for rule models in practice. Starting with an empty set, we greedily add the best compressing rule to the model, until we no longer find a rule that improves our MDL score. Through extensive experiments on both synthetic and real-world data, we show Moody indeed finds succinct and interpretable rules, needs little data for accurate discovery, and is robust to noise.

In summary, our main contributions are as follows. We

(a) formulate the problem of finding data modification rules from process event logs,
(b) formalize the problem using the MDL principle,
(c) propose the Moody algorithm to efficiently find accurate yet succinct data modification rules,
(d) run extensive experiments on both synthetic and real-world data,

(e) make code and data publicly available.[1]

The structure of this chapter is as follows: Next, we introduce the notation of our rule model in Section 3.2. In Section 3.3, we formalize the problem of discovering data modification rules in terms of MDL. Afterward, we propose our heuristic discovery algorithm MOODY in Section 3.4. In Section 3.5, we give an overview of related work, before we empirically evaluate MOODY in Section 3.6. We discuss potential future work in Section 3.7 and conclude this chapter in Section 3.8.

3.2 NOTATION FOR DATA MODIFICATION RULES

As input for finding data modification rules, we consider an event log or dataset D collecting traces of a single process. Each trace refers to an instance of the process, such as a specific customer order, and consists of an event sequence. We describe data attributes of events by a set of numerical and categorical variables V.

To model how a process modifies these variables, we use different types of update rules. For a categorical variable $v \in V$, we write $v \in \{\alpha, \beta, \dots\}$, i. e., v takes one of the values in the set. For a numerical variable $v \in V$, we can set v to a specific value, $v = \alpha$, or to a range of values, $v \in [\alpha, \beta]$. We further denote relative changes by $v = v + \alpha$, $v = v + [\alpha, \beta]$, and $v = \alpha \cdot v$.

Updates typically only occur in certain circumstances. For instance, the price of an order may be dependent on the order volume, where a higher volume gives discount. Therefore, we model conditions for update rules. In the simplest case, we check for a specific value $v = \alpha$ or $v \neq \alpha$. Further, we test lower and upper bounds of numerical values by $v \leqslant \alpha$ and $v \geqslant \alpha$. Finally, we can condition on value transitions between the last and the current event with $v : \alpha \to \beta$.

We combine a condition c and an update rule u into a data modification rule IF c THEN u. To join multiple rules into a model M that covers the full complexity of the process, we use an unordered set of rules, since this allows for an independent interpretation of each rule [163]. Furthermore, to avoid contradictory predictions, we only allow acyclic models. For instance, if we condition on v_1 to update v_2, we are not allowed to do the reverse in the same model.

1 https://eda.rg.cispa.io/prj/moody/

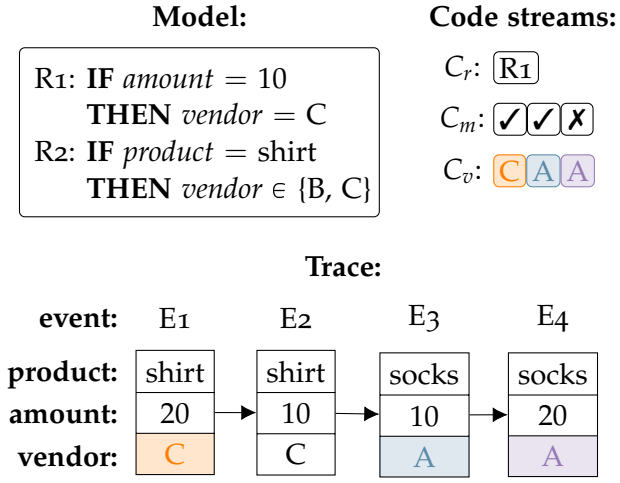


Figure 3.2: **[Data encoding]** Toy example of a rule model (top left) and the corresponding code streams (top right) to encode the categorical variable *vendor* of an exemplary trace (bottom).

3.3 MDL FOR DATA MODIFICATIONS

From an event log D, we aim to find a model M of data modification rules, which accurately describe the data, yet are as succinct as possible, such that domain experts gain insight into the process. Since real-world data is usually noisy, we need a robust model selection criterion. Therefore, we formalize the problem in terms of the MDL principle. To this end, we define length of the data encoding $L(D \mid M)$, length of the model encoding $L(M)$, and finally give a formal problem definition.

3.3.1 *Data Encoding*

To encode a given event log with a model of data modification rules, we encode the variable values at each event. For each event, we check which rules in the model fire, i.e., have satisfied conditions, and use the firing rules to encode the data values. Whenever the model makes ambiguous predictions, we encode the specific value among all possibilities. If no rule fires, we choose and encode a value from the whole domain of the target variable. To ensure a lossless encoding and to handle noisy data, we also encode any errors made by the model.

Conceptually, we split the data encoding into three code streams: In the rule selection stream C_r, we encode which rule among all firing rules we use to encode the current variable value. In the model stream C_m, we encode if the model predicts the correct value. If not, any value of the target domain is possible. Whenever the model predicts multiple values, we choose a value by a code in the value stream C_v.

We give a toy example of a data encoding in Figure 3.2, which we use to describe how to decode the categorical variable *vendor*. First, at event E1, we see that only rule R2 applies. Thus, we do not need to select a rule by reading from C_r. Next, we find a checkmark as the first symbol in C_m, i.e., the model predicts correctly. However, the rule allows two values, B and C, which we disambiguate by reading C from C_v. Next, at event E2, we observe that both rules apply. Therefore, we check C_r to find that we should use rule R1 whose prediction is correct according to the second symbol in C_m. Further, its prediction is not ambiguous and we get the value C in the trace. Afterwards, at event E3, we find that only rule R1 applies but its prediction is incorrect according to the last element in C_m. We obtain the correct value by reading A from C_v. Finally, no rule applies at event E4, so we neither read from C_m nor from C_r. Instead, we read the last value from C_v, A, by which we have successfully decoded the values for *vendor*.

We compute the length of the data encoding by summing the code lengths in C_r, C_m and C_v. Whenever we have to disambiguate multiple firing rules in C_r, we assume all rules in the model are equally important. We compute the encoded length of C_r by

$$L(C_r) = \sum_{i=1}^{|C_r|} \log |R_i| \ ,$$

where $|R_i|$ denotes the set of firing rules at the i-event.

When we compute the encoded length of C_m, we do not know the probabilities of codes for checkmarks and crosses in advance. Therefore, we use a prequential plug-in code (see 2.3) to compute $L(C_m)$: We initialize uniform counts for checkmarks and crosses and update counts after each event, such that we have a valid probability distribu-

tion at each step in the encoding. Asymptotically, this gives an optimal encoded length of C_m. Formally, we have

$$L(C_m) = \sum_{i=1}^{|C_m|} \frac{\mathrm{usg}_i\, C_m[i] + \epsilon}{\mathrm{usg}_i\, ✓ + \mathrm{usg}_i\, ✗ + 2\epsilon} \,,$$

where $\mathrm{usg}_i\, C_m[i]$ denotes how often the i-th code in C_m has been used before, and ϵ with standard choice 0.5 is for additive smoothing.

To compute code lengths in C_v, we use the conditional empirical probability of values. Let u_i be the update rule we selected in C_r to encode the i-th value in C_v. If no rule fires or we encoded an error in C_m, u_i falls back to all possible values in the domain of the target variable. We formally define

$$L(C_v) = -\sum_{i=1}^{|C_v|} \log \frac{\mathrm{fr}(C_v[i])}{\sum_{j \in u_i} \mathrm{fr}(j)} \,,$$

where $\mathrm{fr}(C_v[i])$ denotes how often value $C_v[i]$ occurs in the data, and we normalize this by the frequencies of all values j possible according to the update rule u_i. To encode numerical variables, we assume a histogram-based discretization, by which we can compute all necessary probabilities and code lengths.

Altogether, this gives us a lossless data encoding.

3.3.2 *Model Encoding*

To define the length of the model encoding $L(M)$, we need to encode the number of rules in the model and all conditions c and update rules u in the model. Since the number of rules is unbounded, we use the universal MDL encoding for natural numbers $L_{\mathbb{N}}$. Denoting the number of modification rules in the model as $|M|$, which can be zero, we then obtain the encoded length of the model as

$$L(M) = L_{\mathbb{N}}(|M| + 1) + \sum_{(c,u) \in M} L(c) + L(u) \,.$$

To encode a condition c, we specify its type, which single variable $v \in V$ is tested by c, and all constants used in the condition. Formally, we define

$$L(c) = \log |\{=, \neq, \leqslant, \geqslant, \rightarrow\}| + \log |V| + \sum_{\alpha \in c} L(\alpha) \,.$$

To encode the value $\alpha \in \operatorname{dom} v$ for a categorical variable v, we have

$$L(\alpha) = \log |\operatorname{dom} v| \ ,$$

and for a real-valued α, we have $L_{\mathbb{R}}(\alpha)$.

To encode an update rule u on a variable v, we first specify the type of u, which is one of $v \in \{\alpha, \beta, \ldots\}$, $v = \alpha$, $v \in [\alpha, \beta]$, $v = v + \alpha$, $v = v + [\alpha, \beta]$, or $v = \alpha \cdot v$, for which we need $\log 6$ bits. Then, we encode which variable $v \in V$ is updated by u, and we encode the constants in u the same way we do for conditions. Formally, we have

$$L(u) = \log 6 + \log |V| + \sum_{\alpha \in u} L(\alpha) \ .$$

This gives us the encoded length of the model $L(M)$.

3.3.3 *Formal Problem Definition*

With this, we have all the ingredients to formally define our problem.

Minimal Modification Rules Problem *Given an event log D with variables V, find an acyclic model of data modification rules M that minimizes the total encoding cost $L(D, M) = L(M) + L(D \mid M)$.*

In practice, it is infeasible to solve this problem optimally due to the potentially large number of acyclic models. To approximate this number by a lower bound, we consider the *rule dependency graph* of a model, which is a directed acyclic graph with variables as nodes and their dependencies induced by modification rules as edges. We give a simple example of a rule dependency graph in Figure 3.3. Since each edge requires at least one rule, there are at least as many models as rule dependency graphs. According to Rodionov [127], the number of acyclic graphs with n nodes and up to m edges is

$$A(n, m) = \sum_{i=1}^{n} \left((-1)^{i-1} \binom{n}{i} \sum_{j=0}^{m} \binom{i(n-i)}{m-j} A(n-i, j) \right) \ ,$$

with $A(1, \cdot) := 1$. Because $A(n, m)$ grows exponentially in the number of nodes n and the number of edges m [27, p. 1186], the number of acyclic models grows exponentially in the number of variables $|V|$ and the number of modification rules in the model $|M|$.

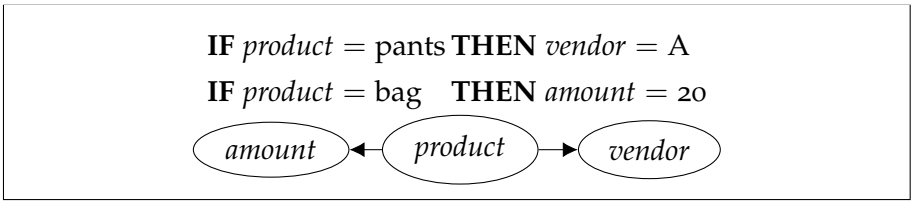


Figure 3.3: **[Rule dependency graph]** We represent the variable dependencies of the model (top) as a graph (bottom), where ovals represent variables, and arrows show whether and in which direction a modification rule induces a dependency between variables.

Furthermore, our search space has no trivial structure such as submodularity or monotonicity, which we could exploit to find an optimal solution in feasible time. Hence, we resort to heuristics.

3.4 THE MOODY ALGORITHM

To efficiently discover good sets of data modification rules in practice, we prune the exponentially sized search space with a quickly computable estimate of our score that avoids repetitively passing the whole event log, and we introduce a greedy search.

3.4.1 *Estimating the MDL score*

Computing the MDL score $L(D, M)$ requires a pass over all events in the event log, because we must check for each event, which of the rules in the model fire. To avoid iterating over the event log each time we evaluate adding a new rule to the model, we prune the large set of potential candidate rules by a quickly computable estimate $\widehat{L}(D, M)$. We optimistically assume that a newly generated rule is the only rule in the model which predicts its target variable, such that we do not need to update C_r or C_m.

By assuming independence between rules, we can independently estimate the contribution of a single rule with condition c and update rule u to $L(C_v)$. We give the pseudocode for estimating $L(C_v \mid c, u)$ as Algorithm 4. First, we compute the support $\text{supp}(c)$, i. e., at how many events c fires (ln. 2). For each value j predicted by u, we compute

Algorithm 4: Estimate $L(C_v)$

input : Rule (c, u)
output: $\widehat{L}(C_v \mid c, u)$

1 $\widehat{L}(C_v \mid c, u) \leftarrow 0$;
2 $b \leftarrow \text{supp}(c)$;
3 **forall** $j \in u$ ordered by increasing $\text{fr}(j)$ **do**
4 $\quad \Delta b \leftarrow \min\{b, \text{fr}(j)\}$;
5 $\quad \widehat{L}(C_v \mid c, u) \leftarrow \widehat{L}(C_v \mid c, u) - \Delta b \cdot \log \frac{\text{fr}(j)}{\sum_{i \in u} \text{fr}(i)}$;
6 $\quad b \leftarrow b - \Delta b$;
7 **return** $\widehat{L}(C_v \mid c, u)$;

how many codes we must add to C_v, which is the minimum of the remaining events to cover, b, and the frequency $\text{fr}(j)$ of j (ln. 4). We compute the length of all these codes and add it to our estimate (ln. 5). At the end of each iteration, we update how many codes we still must add to C_v (ln. 6).

While computing the support requires a pass over the dataset, we only need a single pass when creating the candidate. By assuming independence between rules, we do not need to update the estimated code lengths of all candidate rules, every time we add another rule to the model. Using $\widehat{L}(D, M)$ we can prune out rules with high encoding costs, and avoid computing $L(D, M)$ for those. Next, we use $\widehat{L}(D, M)$ and $L(D, M)$ to find good modification rules from an event log.

3.4.2 *Finding Good Modification Rules*

To reduce the search space, we let the domain experts control how much time they want to invest in model search, and introduce two hyperparameters for a greedy search. Instead of generating conditions with all possible combinations of variables, signs and values, we only generate the N_c most frequently observed combinations of variables and values in the dataset for each sign of the set $\{=, \neq, \leqslant, \geqslant, \rightarrow\}$. For a given condition, instead of generating update rules with all possible combinations of variables and values, we only generate the N_u most frequent combinations of variables and values in the dataset for each type of update rule.

Algorithm 5: MOODY

input : event log D with variables V
output: model of data modification rules M

1 $M \leftarrow \varnothing$;
2 **do**
3 **forall** $v \in V$ **do**
4 $Q \leftarrow$ priority queue of rules r predicting v ordered by $\widehat{L}(D, M \cup \{r\})$;
5 $r^* \leftarrow \varnothing$;
6 **while** $Q \neq \varnothing$ **and** $\widehat{L}(D, M \cup \{\text{top}(Q)\}) < L(D, M \cup \{r^*\})$ **do**
7 $r' \leftarrow$ pop element from Q;
8 $r^* \leftarrow \arg\min_{r \in \{r^*, r'\}} L(D, M \cup \{r\})$;
9 **if** $L(D, M \cup \{r^*\}) < L(D, M)$ **then**
10 $M \leftarrow M \cup \{r^*\}$;
11 **while** M was extended in the last iteration;
12 **return** M;

We provide the pseudocode of our greedy search MOODY as Algorithm 5. Our search starts with an empty model (ln. 1). We iteratively extend this model by modification rules for all target variables (ln. 3). Since computing $L(D, M)$ needs a pass over the whole dataset, we manage the candidate rules in a priority queue sorted by an estimate of our score $\widehat{L}(D, M)$ (ln. 4), such that we check promising rules early. Next, we search the candidates from most promising to least promising and compute their actual encoded length L (ln. 6-8). To not waste computation time for computing L on inferior rules, we perform this search as long as the estimate $\widehat{L}$ is better than the best actual code length L that we have seen so far (ln. 6). After evaluating the candidates, we only add the best candidate to our model if it reduces the total encoded length (ln. 9-10). Finally, we end when no candidate for any target variable could improve our score (ln. 11) and return the resulting model (ln. 12).

In the worst case, all generated candidates improve our MDL score. Since the number of candidates grows linearly with N_c and N_u, the outer loop of MOODY grows linearly with N_c and N_u. In the worst case,

our estimate does not prune any candidate, and we must compute our score for all the $O(N_c \cdot N_u)$ candidates. Since we loop over all variables, and computing our score requires a pass over the whole dataset, the runtime complexity of MOODY is $O\left((N_c \cdot N_u)^2 \cdot |V| \cdot |D|\right)$.

3.5 RELATED WORK

While the problem of finding succinct and interpretable data modification rules from process event logs has been neglected so far, related work on similar problems exists. Krismayer [74] discovers if-then rules for data modification from software execution logs. However, he only creates a large set of potentially redundant candidates, which must be manually filtered by domain experts. Other work [47, 153] infers extended finite state machines from software execution logs. Since business process event logs in contrast to software event logs are usually noisy and contain nondeterministic behavior, these methods are not applicable to our problem.

While process mining [1] focuses on business process event logs, most of the work only models the flow of process activities, and little work deals with additional data. Mannhardt et al. [90] and Mozafari et al. [97] detect at which event data changes, but do not model conditions or update values for these changes. Schönig et al. [132] find rules which cover data modifications. However, since they rely on support and confidence to filter a large set of candidate rules, their method suffers from pattern explosion, i. e., it finds many redundant rules.

Rule-based prediction is closely related to our problem. CLASSY [119] and its successor TURS [163] find classification rules by minimizing an MDL score. Both methods, however, require defining features and predicted variable beforehand, whereas we are interested in finding relationships without any initial knowledge of the data. Similarly, subgroup discovery algorithms such as SSD++ [118] find rules for differently behaving subgroups of a given dataset. None of these methods are able to model sequential relationships present in event logs.

In contrast to the above, MOODY finds compact and interpretable rules for data modifications from process event logs, needs little data for accurate discovery, and is robust to noise.

3.6 EXPERIMENTS

In this section, we empirically evaluate MOODY on both synthetic and real-world data. When we defined our MDL score, we assumed discretization of numerical variables. In our prototype implementation, we discretize numerical values into variable-width histograms. For efficiency, we determine the histogram boundaries by percentiles. We use 50 bins in all our experiments.

We run all our experiments in a Docker-based environment on a Linux server with an Intel® Xeon® Gold 6244 CPU. In all experiments, we observe 16 GB of RAM suffice. As a simple baseline, we consider the empty model $M = \varnothing$. In addition, we learn if-then-else rules using the rule-based classifier TURS [163], and using the subgroup discovery method SSD++ [118]. To ensure reproducibility of all our results, we provide code and data in the supplementary material.

3.6.1 *Synthetic Event Logs*

To control data properties such as noise, we first experiment on synthetic event logs, such that we know the generating ground-truth rules. We randomly generate ten independent ground-truth models in our modeling language, where each model contains five rules, two categorical variables and two numerical variables. Since SSD++ cannot model sequential dependencies $v : \alpha \rightarrow \beta$, we only create models with conditions $v \leqslant \alpha$, $v \geqslant \alpha$ and $v = \alpha$. From these ground-truth models, we generate event logs with $|D| = 2000$ events. To test noise-robustness, we add different amounts of noise, where we randomly swap values of variables. 10% swap noise means that for each variable in the dataset, we randomly swap 10% of its values.

TURS in contrast to SSD++ only discovers rules for categorical target variables, and cannot predict numerical values. Therefore, we separately evaluate on models predicting only categorical variables and on models predicting only numerical variables. However, in both setups, we generate conditions containing categorical and numerical variables.

CHOOSING HYPERPARAMETERS Such that the user can control runtime by reducing the search space, we introduced hyperparameters N_c and N_u when we proposed MOODY. For efficiency, we only search for

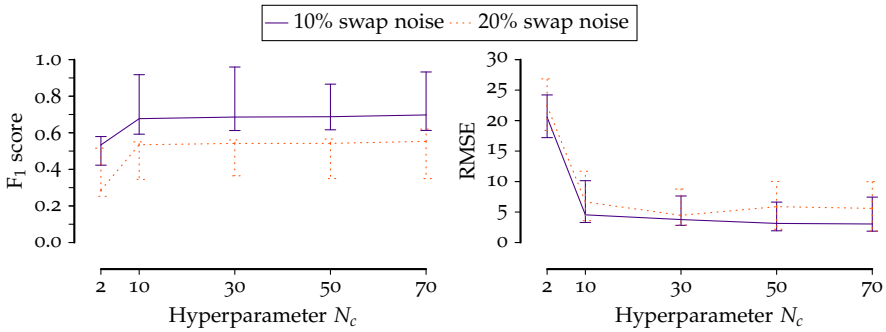


Figure 3.4: [**Choosing** N_c] Median F_1 score on categorical variables (left) and median RMSE on numerical variables (right) for different values of Moody's hyperparameter N_c for 10% and 20% swap noise in the training set. Error bars show interquartile ranges.

the most compressing update rule given a condition and set N_u to 1. To evaluate the accuracy of a set of rules, we compute its median F_1 score on predicting categorical variables, and its median root mean squared error (RMSE) in predicting numerical variables. We report the influence of N_c on prediction accuracy in Figure 3.4. We see that the prediction accuracy drops for only very small N_c, and is relatively constant for larger values. This means, the conditions with the highest support in the dataset tend to give the best compression and accuracy. To have a safety margin on unknown data, we set N_c to 50.

RESULTS ON CATEGORICAL VARIABLES Next, we compare results on synthetic data with different amounts of swap noise for Moody against all baselines in Figure 3.5. We see in the left plot that Moody achieves by a wide margin the highest F_1 score on categorical variables for data with 0% and 10% noise. Up to 20% noise, Moody shows good noise-robustness and still has the highest median F_1 score. For 30% noise, the F_1 score of Moody drops down to the F_1 score of the empty model. We note that 30% noise may sound low, but swapping 30% of the values for each variable in the dataset accumulates to a much higher noise-ratio. Hence, Moody predicts well under reasonable amounts of noise.

RESULTS ON NUMERICAL VARIABLES We see similar results on predicting values of numerical variables in the center plot of Figure 3.5.

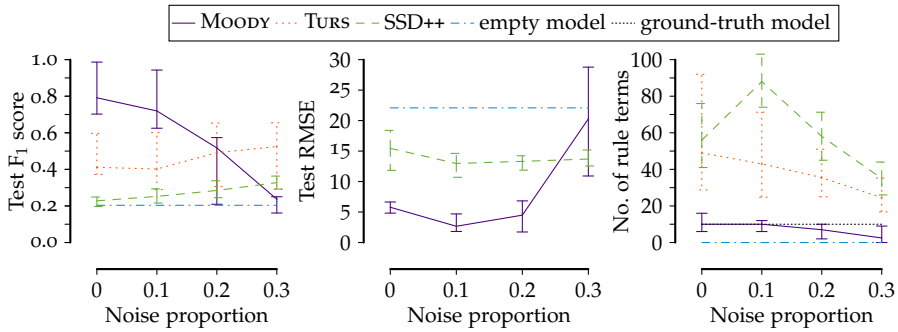


Figure 3.5: **[Moody predicts well under reasonable amounts of noise]** Median F_1 scores on categorical variables (left, higher is better), median root mean squared errors on numerical variables (center, lower is better) and number of rule terms in the discovered models (right, lower is less complex) at different noise levels for Moody, SSD++ and Turs. Error bars indicate interquartile ranges.

Since Turs cannot predict numerical variables, we compare Moody to SSD++ and the empty model. Moody shows by far the smallest test root mean squared error (RMSE) for training data with up to 20% noise. For 30% noise, Moody's RMSE converges to the RMSE of the empty model. We again note swapping 30% of the values for each variable in the dataset accumulates to a much higher noise-ratio than we expect in any real-world event log. Hence, Moody predicts well under reasonable amounts of noise.

MODEL COMPLEXITY Not only does Moody give the best prediction results under reasonable amounts of noise. It also discovers the rule sets with the lowest total number of rule terms as we show in the right plot of Figure 3.5. Both Turs and SSD++ find rule sets with a significantly higher number of rule terms. We see Moody's models have similar complexity compared to the ground-truth models for data with low amounts of noise. If the noise level increases, Moody converges to the empty model. This means, Moody is robust against finding spurious rules on noisy data.

SAMPLE COMPLEXITY To empirically evaluate sample complexity, we compute F_1 scores on the test set dependent on the number of events in the training set. For a realistic setup, we add 10% swap noise

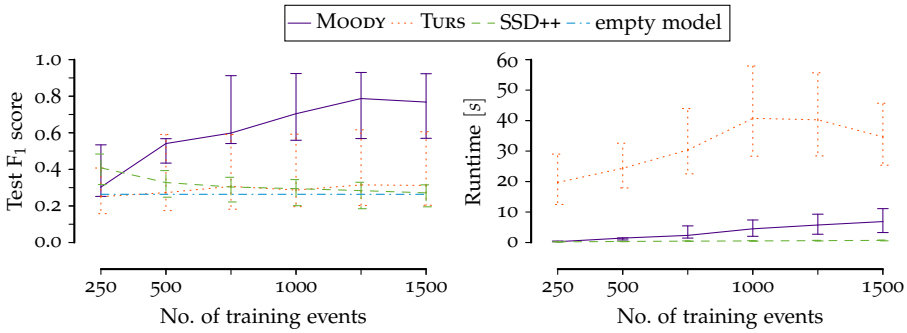


Figure 3.6: [**Moody shows low sample complexity and scales well**] Median test F_1 score on categorical variables (left) and median runtime (right) dependent on the number of training events for Moody, Turs and SSD++. Error bars show interquartile ranges.

to all training sets. We report results for Moody, Turs, SSD++ and the empty model in the left plot of Figure 3.6. We see Moody predicts better the more training data is available, while all baselines do not improve with more training data. Already with 500 training events, Moody shows higher median F_1 score than the baselines.

RUNTIME We report wall-clock runtime for single-threaded execution dependent on the number of training events in the right plot of Figure 3.6. As we expect by our theoretical runtime analysis, we see that Moody scales well and shows a growth of runtime linear to the number of training events. While SSD++ shows a constantly fast, almost zero runtime, Moody still finishes within reasonable time and is significantly faster than Turs.

3.6.2 *Real-World Event Logs*

Next, we evaluate on two publicly available real-world event logs, for which we give the base statistics in Table 3.1. The first one, *Sepsis* [89], contains event traces from treating Sepsis patients in a Dutch hospital. The second one, *Traffic Fines* [84], [90, p. 20] is an event log of handling road-traffic fines by the police of an Italian city. To reduce runtime, we randomly sample 20% of the original 150370 traces in the Traffic Fines event log. Furthermore, we parallelize candidate generation and candidate evaluation of Moody on twelve CPU cores.

Data	$\lvert D\rvert$	$\lvert\lvert D\rvert\rvert$	$\lvert\Omega\rvert$	$\lvert V_{\text{cat}}\rvert$	$\lvert V_{\text{num}}\rvert$
Sepsis	782	15214	18	25	7
Traffic Fines	30074	112245	11	4	5

Table 3.1: **[Real-world event logs statistics]** Number of traces $\lvert D\rvert$, number of events $\lvert\lvert D\rvert\rvert$, number of different activities $\lvert\Omega\rvert$, number of categorical variables $\lvert V_{\text{cat}}\rvert$, and number of categorical variables $\lvert V_{\text{num}}\rvert$ for the Sepsis and Traffic Fines real-world datasets.

IF *group* = C **THEN** *activity* = ER Triage
IF *group* = E **THEN** *activity* = Release A
IF *group* = W **THEN** *activity* = Admission IC
IF *group* = P **THEN** *activity* = Admission IC
IF *group* = F **THEN** *activity* = Admission NC
IF *group* = O **THEN** *activity* = Admission NC

Figure 3.7: **[Responsibility rules for the Sepsis log]** Different groups in the hospital are associated with different activities.

IF *Leukocytes* $\leqslant$ 4.7 **THEN** *Infusion* = True
IF *Leukocytes* $\leqslant$ 2.8 **THEN** *Diagnose* = GB
IF *Leukocytes* = 7.8 **THEN** *Diagnose* = AA

Figure 3.8: **[Leukocytes rules for the Sepsis log]** Lower values for leukocytes are associated with certain diagnoses and infusions.

First, we look at the insight we gain from some exemplary rules we find with Moody on these event logs. Then, we evaluate how well the discovered rules generalize to unseen test data.

SEPSIS RULES On the Sepsis dataset, Moody finds in total 82 rules with an approximate runtime of two hours. 23 of these rules express a correlation between the *group* attribute and the activity of an event, for

IF *amount* $\leqslant$ 68.77 **THEN** *points* $=$ 0.0
IF 143.00 $\leqslant$ *amount* **THEN** *points* $\in$ [0.0, 10.0]
IF 131.00 $\leqslant$ *amount* **THEN** *points* $=$ 0.0
IF *amount* $=$ 80.00 **THEN** *points* $\in$ [0.0, 2.0]

Figure 3.9: [**Traffic Fines rules**] MOODY finds multiple relationships between the fine *amount* and the number of *points* deducted from the offender's driving license.

which we show a subset in Figure 3.7. The rules indicate that certain groups in the hospital are specialized in certain activities.

Furthermore, MOODY finds rules, where leukocytes measurements imply the diagnosis and the treatment of sepsis, as we show in Figure 3.8. These rules enable to ask targeted questions to domain experts and thus are a valuable start to gain insight into this process.

TRAFFIC FINES RULES On the Traffic Fines dataset, MOODY finds in total 117 rules with an approximate runtime of 4.5 hours. While we would expect that a higher fine *amount* correlates with a high number of *points* deducted from the offender's driving license, the rules contradict this intuition, as we show in Figure 3.9. A closer look on the dataset indeed confirms there is no monotonic relationship between these two attributes. Discovering counter-intuitive but data-supported rules like these gives valuable insight into the underlying process.

GENERALIZATION Finally, we evaluate how well the rules discovered by MOODY generalize to unseen data. To this end, we split the Traffic Fines event log into a training set and a test set with a distinct 20% of traces each. Then, we compare the F_1 score respectively RMSE on the training set and the test set for each rule, which MOODY discovers on the training set. We show results in Figure 3.10. As we see, most of the rules have a low prediction error on both sets. The gap between training and test performance is small, which means that MOODY finds well-generalizing rules.

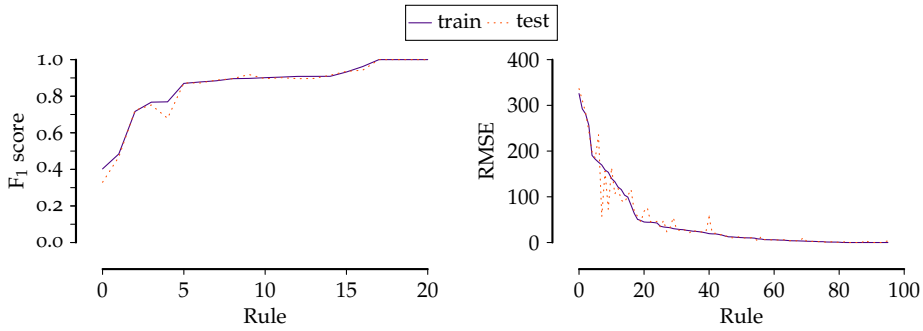


Figure 3.10: **[Moody generalizes well]** Train and test F_1 score on categorical variables (left) and train and test root mean squared error on numerical variables (right) for each rule found by Moody on the Traffic Fines event log.

3.7 DISCUSSION

In our experiments, Moody does not only discover simple and thus interpretable rules to unveil the data modifications within a process. It also finds rules that accurately predicts the data values, and is more robust to sensible amounts of noise than all baselines.

Nonetheless, we think of multiple interesting research directions to improve Moody. First, extending the modeling language of Moody would allow understanding data modifications of very complex processes. For example, this involves rules with complex conditions that consist of many condition terms joined by *(and)* and *(or)*. We see adapting our MDL score to such rules is relatively easy.

However, a more complex modeling language implies an even larger search space, and thus we see the need to improve the runtime efficiency of Moody. As in many MDL-based methods, the bottleneck of Moody is discrete combinatorial search. Hence, we see an approach for mining data modification rules based on differentiable pattern set mining [46] as a promising future direction.

Furthermore, we would like to put a stronger focus on causality of the discovered rules. To this end, we may use well-defined measures for the causal effect of a rule [23]. Alternatively, we would like to examine the link between causality and two-part MDL codes in terms of algorithmic independence [93], and how to use this during search for causal data modification rules.

Last but not least, we see many interesting applications for MOODY. As the most compressing rules found by MOODY define normal behavior, it would be interesting to use them for anomaly detection [98]. Since the behavior of real-world processes usually changes over time, we see MOODY could help to identify and understand concept drift [21, 131]. Finally, predicting data attributes with MOODY may be used in the simulation of process behavior for process optimization [57].

3.8 CONCLUSION

We studied the hitherto largely neglected problem of discovering accurate yet concise and interpretable rules how event attribute data changes throughout a business process. We formalized the problem in terms of the Minimum Description Length (MDL) principle, by which we choose the model with the best lossless description of the data. To efficiently search for rule models in practice, we proposed our greedy algorithm MOODY. Through extensive experiments on both synthetic and real-world data, we showed MOODY indeed discovers succinct and interpretable if-then rules, needs little data for accurate discovery, is robust to sensible amounts of noise, and thus gives valuable insight into data modifications.

Besides applying MOODY on downstream tasks such as anomaly detection, concept drift detection and simulation, future work involves extending the rule language of MOODY to model more complex conditions for data changes, and runtime optimizations to enable search for such complex rules in feasible time.

4

PREDICTING EVENT SEQUENCES

In the previous chapters, PROSEQO and PROSIMPLE provided us with an initial understanding of events and potential behaviors within a process, while MOODY revealed how data changes throughout that process. Now, we transition from comprehension to prediction as we delve into the challenge of predicting an event sequence. Specifically, we are interested in learning easily interpretable models capable of accurately generating a sequence based on a given attribute vector.

4.1 INTRODUCTION

Real-world event sequences are often accompanied by additional metadata. For example, event logs of manufacturing processes contain sequences of production steps with product properties and attributes by the customer's order. Usually, there is a relationship between these attributes and the observed event sequence, like certain product groups require different manufacturing activities. To gain a better understanding of the underlying data generating process, we are often interested in uncovering this mechanism.

In a predictive scenario, for example, production planners want to know the event sequence for a given product in advance, such that they can avoid bottlenecks and optimize the process flow. Rules in production planning systems are often hand-crafted and do not necessarily display the true complexity of the real process. Existing process models tend to show idealized, high-level behavior and thus give a limited picture of the real process [1, p. 30].

We are not the first to study sequence prediction based on metadata. Existing neural network approaches [24, 105, 150] can achieve high accuracy given sufficient training data and hyperparameter tuning; how-

This chapter is based on [159]: Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Discovering Interpretable Data-to-Sequence Generators." In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*. 2022, pp. 4237–4244.

ever, the resulting models are inherently difficult to interpret. As the key applications, such as optimizing planning, require interpretation, these solutions do not suffice in practice. Surprisingly little work results in interpretable models being accurate and robust to noise.

We take a different approach. We propose to model the event sequences as a directed graph with classification rules on the metadata to determine which paths to follow. Such an *event-flow graph* should fit the data well, but at the same time have a low model complexity to increase interpretability by humans. Therefore, we formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we identify the best model as the one giving the shortest lossless description of the data.

Due to NP-hardness of the resulting optimization problem, we propose the greedy method ConSequence, which first discovers a directed graph for a given set of event sequences and then finds classification rules on the metadata for nodes with multiple successors. While in practice any rule-based classifier can be plugged in, we propose the algorithm GERD, which uses a reliable rule effect estimator to find compact and meaningful rules. Through extensive experiments including a case study, we show ConSequence discovers compact, interpretable and accurate models for the generation of event sequences from data. Our method has low sample complexity, works well under noise and deals with different real-world data.

Our main contributions are

(a) formulate the problem of interpretable yet accurate prediction of event sequences from metadata with MDL,
(b) an efficient heuristic to discover event-flow graphs with classification rules for event sequence prediction
(c) an extensive empirical evaluation,
(d) make all code and data publicly available.[1]

Next, we introduce the notation of our event-flow graph model in Section 4.2. In Section 4.3, we formalize the problem of discovering an event-flow graph in terms of MDL. Afterward, we propose our algorithms ConSequence and GERD in Section 4.4. In Section 4.5, we give an overview of related work, before we empirically evaluate ConSequence and GERD in Section 4.6. We discuss potential future work in Section 4.7 and conclude this chapter in Section 4.8.

1 https://eda.rg.cispa.io/prj/consequence/

4.2 NOTATION FOR EVENT-FLOW GRAPHS

Before we formalize the problem, we introduce notation and concepts used in the remainder of this chapter. As input data, we consider datasets of event sequences with meta data. Such a dataset D consists of n instances (x, y), where x is a vector with meta data, and y is a finite event sequence as defined in Chapter 2. We write X to refer to all meta data vectors, and Y to refer to all event sequences in the dataset.

We propose to model the prediction of event sequences from metadata with *event-flow graphs*. An event-flow graph $M = (G, R)$ consists of a directed graph G and a *rule relation* R. As any directed graph, G is defined by a tuple (V, E), where the nodes correspond to events from Ω. Multiple nodes are allowed to refer to the same event. In addition, a valid event-flow graph consists of a source node v_s and a sink node v_e, which do not refer to any event. We use a path from v_s to v_e to represent an event sequence y.

To model the relationship between metadata and event sequence, we assign classification rules to nodes. At a given node v, such a rule predicts the next node to follow. Formally, we denote a rule by $\gamma \to u$, where $\gamma : X \to \{\top, \bot\}$ is the *condition*, that for a given metadata vector x either evaluates to true ($\top$) or false ($\bot$), and $u \in \mathrm{succ}(v)$ is the *consequence*. A condition consists of multiple terms that we stack together using *and* ($\wedge$) and *or* ($\vee$), e. g., $\gamma = (\theta_1 \wedge \theta_2) \vee \theta_3$. A condition term θ always consists of an attribute $a \in A$, an operator from the set $\{>, \leqslant, =\}$ and a value for comparison $q \in \mathrm{dom}\, a$. We combine multiple rules to *decision lists* [125], which are ordered rule sets, where the classification output is determined by the first firing rule, i. e., for which $\gamma(x) = \top$.

Given an event-flow graph M, we describe or *cover* a sequence $y \in \Omega^*$ by traversing M from v_s to v_e. Since real-world processes and data usually contain noise, we allow errors while traversing the graph to enable succinct models. To reconstruct a sequence from a given cover, we read instructional codes from the *code stream* C, which together with the rules in the graph determine the path through M and correct missed or redundant events. Conceptually, we split C into the *model stream* C_m, which encodes how to traverse the model, and the *disambiguation stream* C_d, which encodes ambiguous choices of events.

For better illustration, we give a toy example for the cover of a sequence using a simple event-flow graph in Figure 4.1. Starting at the

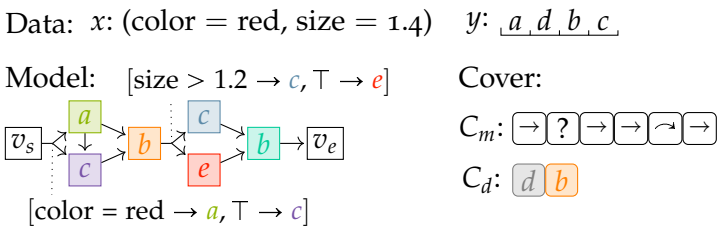


Figure 4.1: **[Cover]** Toy example for a sequence y with metadata x and a cover of the sequence using a simple event-flow graph.

source node v_s, we read the first code from C_m. The [→] tells us to move one step forward in the model and emit the next event we arrive at. Since the color attribute of our exemplary data x has the value *red*, the rule at v_s only allows us to go to a, which we then emit. We read the next code from C_m, [?], which means the next event is not captured by the model. To disambiguate the choice between the events in Ω, we read from C_d and get the code for d. The next code in C_m is [→], so we go forward in the model and emit an event. From a, we can either go to b or to c, and this time, there is no rule telling us, which path to follow. Therefore, we read again from C_d and go to b, which we also emit. We continue by reading [→] from C_m, we evaluate the rule on the size attribute and go to and emit c. The next code in C_m, [↝], tells us to go forward, but not to emit the event, i. e., the event in the model is redundant. From c, we can only go to b. Finally, we read the last [→] and arrive at v_e, which means we have reconstructed y without loss.

We now take this concept of sequence cover to define an MDL score that will formalize how a good event-flow graph for a given dataset should look like.

4.3 MDL FOR EVENT-FLOW GRAPHS

A good event-flow graph should fit the data well and at the same time avoid unnecessary complexity. We use the MDL principle to formalize this requirement, i. e., we are looking for a model with low overall encoding cost. First, we define how to compute the length of the data encoding using the cover concept as introduced in the former section.

4.3.1 *Data Encoding*

Let Y be a given set of sequences, X the corresponding attribute vectors and M an event-flow graph. Then, the encoded length of the data is

$$L(Y \mid M, X) = L(C_m) + L(C_d) \ ,$$

i. e., we have to compute the encoded length of the model stream C_m and the disambiguation stream C_d in the cover.

Since we do not know the distribution of the codes in C_m and C_d beforehand, we use prequential codes (see Section 2.3). We condition code lengths on the current node in the event-flow graph while covering a sequence to make maximal use of available information, and to avoid that local changes in the graph change encoding lengths at all nodes. For the model stream C_m, which contains the codes [→], [↷] and [?], this results in an encoded length of

$$L(C_m) = -\sum_{i=1}^{|C_m|} \log \frac{\mathrm{usg}_i(C_m[i] \mid v_i) + \epsilon}{\sum \mathrm{usg}_i(\ \cdot \mid v_i) + \epsilon} \ ,$$

where $\mathrm{usg}_i(C_m[i] \mid v_i)$ denotes how often the i-th code in C_m has been used before at the current node v_i, and ϵ with standard choice 0.5 is for additive smoothing.

The codes in C_d refer to events in Ω. Which codes are possible at one point of time depends on the last code in C_m. If we go forward in the model after reading a [→] or [↷] code, only events of the directly following nodes in the graph are possible, whereas reading a [?] enables all events in Ω. Therefore, we conceptually split C_d into three individual streams, with C_c being the stream for correctly modeled events after reading [→], C_r being the stream for redundant events after reading [↷] and C_x for missed events after reading [?]. Then, all three streams follow the same computation scheme as C_m. For example,

$$L(C_x) = -\sum_{i=1}^{|C_x|} \log \frac{\mathrm{usg}_i(C_x[i] \mid v_i) + \epsilon}{\sum \mathrm{usg}_i(\ \cdot \mid v_i) + \epsilon} \ .$$

This gives us a lossless encoding of the data using an event-flow graph.

4.3.2 *Model Encoding*

Since we are using prequential codes for the data encoding, the computation for the encoded length of an event-flow graph $L(M)$ is quite simple. Intuitively, a graph with more nodes, more edges and more rules should have a higher encoding length. Formally, we have

$$L(M) = L_{\mathbb{N}}(|V| + 1) + |V| \cdot \log|\Omega| + \log(|V|^2 + 1) + \binom{|V|^2}{|E|} + \sum_{v \in V} L(v) \,,$$

where we first encode the number of nodes in the graph, then the events of the nodes, the number and layout of the edges and finally the rules at each node. We encode the number of edges with o as a lower bound and $|V|^2$ as an upper bound. For the edge layout, we use a data-to-model code [87], which is an index over a canonically ordered set of all directed graphs of $|V|$ nodes and $|E|$ edges.

If a node has less than two successors, no rule to decide which path to follow is necessary, and $L(v) = 0$. Otherwise, we compute the encoded length of the rules at node v with

$$L(v) = L_{\mathbb{N}}(|v|) + \sum_{\gamma, c \in v} L(\gamma) + \log \deg^+(v) \,,$$

where we first encode the number of rules and then the conditions and consequences. The encoded length of a condition is computed depending on its type. For a simple term θ that makes a comparison on attribute a with operator o, we have

$$L(\theta) = \log|\{\text{or}, \text{and}, \text{term}\}| + \log|A| + L(o \mid a) + \log|\operatorname{dom} a| \,,$$

i. e., we encode among the three possible options that we have a simple term, and then specify an attribute, an operator and a value for comparison. The encoded length of the operator depends on the chosen attribute: For categorical attributes, the only possible operator in our rule language is $(=)$, i. e., $L(o \mid a) = 0$, whereas for numerical attributes, we have to distinguish between $(>)$ and $(\leqslant)$, i. e., $L(o \mid a) = \log 2 = 1$. Like in many rule mining or decision tree algorithms, we assume a discretization grid for cut points in conditions [41], such that we can use $\log|\operatorname{dom} a|$ to compute the encoded length for both categorical and

numerical attributes. If a condition consists of multiple subconditions either joined with ($\vee$) or ($\wedge$), we compute the encoded length by

$$L_{\wedge}(\gamma) = L_{\vee}(\gamma) = \log|\{\text{or}, \text{and}, \text{term}\}| + L_{\mathbb{N}}(|\gamma|) + \sum_{i=1}^{|\gamma|} L(\gamma_i) \ ,$$

where we again first choose the type of the term, and then specify the number of subconditions, before we recursively encode each subcondition, which is either a simple term or again consists of subconditions.

Altogether, this gives us a lossless encoding of an event-flow graph.

4.3.3 *Formal Problem Definition*

We now have all ingredients to formally define our sequence prediction from metadata problem.

Minimal Event-Flow Graph Problem *Given a dataset with attributes and event sequences (A, X, Y), find the rule-containing event-flow graph M and cover C, that minimizes the total encoded cost $L(M) + L(Y \mid M, X)$.*

Solving this problem optimally is infeasible in practice. Just finding the optimal cover for a given sequence and event-flow graph is already NP-complete. This is due to the equivalence of event-flow graphs and 1-safe nets, for which computing the optimal alignment with a sequence has been proven as NP-complete, even if the model is acyclic [5, 25].

Furthermore, we do not know the event-flow graph to begin with. The search space for event-flow-graphs is large, because event-flow graphs are directed acyclic graphs and their potential number grows super-exponentially with the number of nodes [126]. Finally, at every branch in the model, we need to mine a rule list that predicts which path to follow. Mining optimal decision trees or rule sets with minimal model complexity, however, is also NP-complete [8, 59].

Since the search space of the problem does not show any trivially exploitable structure, such as monotonicity or submodularity, we resort to heuristics.

4.4 ALGORITHM

To discover good event-flow graphs in practice, we split the Minimal Event-Flow Graph Problem into multiple parts, i. e., computing a cover,

Algorithm 6: COVER an Event Sequence

input : event sequence y, event-flow graph M, beam width w, heuristic h
output: a cover for y using M

```
1 Q ← queue containing the empty cover;
2 while true do
3     C ← pop top element from Q;
4     if C covers both y and M then
5         return C
6     foreach possible extension C′ to C do
7         c ← number of [?] and [↷] in C_m;
8         insert C′ into Q using priority c + h(C′);
9     Q ← the w best candidates in Q;
```


discovering an event-flow graph, and finding classification rules for path prediction. Since each part is already hard to solve by itself, we propose greedy solutions to each of the subproblems separately. We discuss the algorithms for these in turn.

4.4.1 *Computation of a Cover*

We start by finding a good cover for a given model M, i. e., a cover C with low $L(Y \mid M, X)$. To compute a cover with near-optimal encoding length, we use the intuition that the better a model fits the data, the shorter the encoded length of the cover should be. This is equivalent to minimizing the number of [?] and [↷] codes in C_m. This formulation of the problem is equivalent to finding an optimal alignment between a Petri net and a sequence, where the standard approach to solve this problem optimally is to apply an A* search strategy [5].

We follow this approach, for which we give the pseudocode as Algorithm 6. We start with an empty cover, which we iteratively extend, until we find a complete cover, i. e., after decoding we have reconstructed the whole sequence and have arrived at v_e. The candidates are ranked by their cost, i. e., the number of errors the model makes in terms of [?] and [↷] codes in the cover, plus a heuristic h, which is an estimate of the cost for making the candidate a complete cover. If the heuristic

function h is *admissible*, i. e., a lower bound of the true cost, and *consistent*, i. e., non-increasing for following states, A* finds the optimal solution [55]. One admissible and consistent heuristic for our problem is the number of uncovered events in the sequence for which there is no corresponding node reachable [5, p. 66].

Unfortunately, always having the guarantee to find the optimum comes quite with a cost of runtime. Therefore, to cover a sequence in feasible runtime, we modify the A* search in a beam search fashion, where we only keep the w best candidates in each iteration.

The runtime of computing a cover depends on the branching factor b, which is the average number of expansions for nodes during search, and on the depth d of the search tree. Without limiting the capacity of the candidate queue, A* has worst-case runtime complexity $O(b^d)$. With the beam width parameter w, the number of expanded nodes in the worst-case scenario is $\sum_{i=0}^{w-1} b^i + wb(d+1-w)$, i. e., the runtime grows linearly with depth d and exponentially with beam width w. The branching factor b strongly depends on the average degree in the event-flow graph. When expanding an existing partial cover, we can go to any successor of the current node in the event-flow graph using a [→] or a [⇝], or we can have a [?] without moving in the model. This leads to the upper bound $b \leqslant 2\frac{|E|}{|V|} + 1$.

We find a complete cover by the latest after as many [?] codes as events in the sequence, and as many [⇝] codes as the length of the shortest path from v_s to v_e. In other words, we can cover sequence and model independently of each other, which gives an upper bound on d. Since, in any sensible event-flow graph, the shortest path from v_s to v_e is not longer than the longest sequence in the dataset, the upper bound is $d \leqslant 2 \max_{y \in Y} |y|$.

4.4.2 *Discovering an Event-Flow Graph Without Rules*

With the cover algorithm, we can now compute our optimization target $L(M, Y, X) = L(M) + L(Y \mid M, X)$. To reduce the search space, we first try to find a compact event-flow graph without rules. One can interpret this as an event-flow graph with optimal rules emulated by the cover algorithm. After having an event-flow graph, we can learn rules that try to reproduce the routing choices of the cover.

Algorithm 7: DISCOVER EVENT-FLOW GRAPH

input : event sequences Y
output: event-flow graph M

```
M ← empty graph;
foreach unique sequence y ∈ Y in descending frequency do
    M' ← M ∪ y;
    if L(M', Y) < L(M, Y) then
        M ← M'
return M
```

To find a good event-flow graph without rules for a given dataset, we propose a greedy bottom-up search, for which we provide the pseudocode as Algorithm 7. We start with an empty graph, which just consists of source v_s and sink v_e. Iteratively, we add paths to the model corresponding to the most frequent sequences in the dataset. We only keep paths, that improve the objective score.

In the worst-case, every sequence in Y is unique, which means that we need $n = |Y|$ iterations. In each iteration, we compute the cover to find extension points in the model for an uncovered sequence y, and to compute the total encoded length to decide whether we keep or reject the extension. This makes the cover algorithm the main bottleneck in the discovery algorithm.

4.4.3 *Finding Classification Rules for Path Prediction*

After having found an event-flow graph and a cover, we now discover rules to reproduce the routing decisions by the cover. At each node with more than one successor, we learn a rule-based classifier, that predicts the next node for given metadata. The learning algorithm should produce rules, that fit the data well, which leads to small $L(Y \mid M, X)$, because the rules reduce entropy and thus code lengths in the code stream of the cover. At the same time the rules should not get too complex, which would result in high $L(M)$.

One should notice that each node contains its own classification dataset. If this node corresponds to infrequent yet relevant behavior, the number of instances in the dataset will be low. To deal with datasets

containing many attributes, we need a statistically robust learner that needs little data to infer meaningful, well-generalizing rules.

To this end, we try to find rules with a high *effect e* on predicting the next event node $u \in V$, $e(\gamma, u) = \mathrm{P}(u \mid \gamma) - \mathrm{P}(u \mid \bar{\gamma})$. A positive effect means that setting attributes x such that $\gamma = \top$ increases chances to observe u as the next node. The effect is robustly estimated by

$$\hat{e}(\gamma, u) = \frac{n_{\gamma,u} + 1}{n_\gamma + 2} - \frac{n_{\bar{\gamma},u} + 1}{n_{\bar{\gamma}} + 2} - \frac{\beta}{2\sqrt{n_\gamma + 2}} - \frac{\beta}{2\sqrt{n_{\bar{\gamma}} + 2}},$$

where $n_{\gamma,u}$ and $n_{\bar{\gamma},u}$ are counts how often node u is observed as next node, if condition γ evaluates to $\top$ or $\bot$, n_γ and $n_{\bar{\gamma}}$ are counts how often condition γ in total evaluates to $\top$ or $\bot$, and β is a confidence parameter. Higher values for β require more evidence to compute a positive effect and increase robustness to outliers. [23]

Maximizing the effect e minimizes our MDL defined objective score: Applying the logarithm for an information theoretic interpretation of probability distributions, one can transform $P(c \mid \gamma) - P(c \mid \bar{\gamma})$ into $-\log\left(P(c, \gamma) - P(c)P(\gamma)\right) - \log P(\gamma)$. This means, rules that have high predictive power on the next node in the event-flow graph, decrease $L(Y \mid M, X)$. The term $-\log P(\gamma)$ can be seen as a regularizer for infrequent rules, which has a positive effect on minimizing $L(M)$.

Since mining rules with minimal model complexity is NP-complete [8, 59], we propose a greedy approach for finding rules with maximal effect. We call our method *greedy effective rule discovery* (GERD) and give its pseudocode as Algorithm 8. Stating with an empty rule list, we iteratively add rules until we have covered all instances in the dataset. To greedily find a rule with high effect, we first look at rules with one term (ln. 3) and only keep the rule with the highest effect (ln. 4). If this rule does not have a positive effect, we replace it with the default rule (ln. 6), which is a rule that always fires and predicts the node with the highest support. Otherwise, we try to extend the rule with one additional term using ($\wedge$) and ($\vee$), such that the effect of the rule increases (ln. 8-11). If this is successful, we again try to extend the rule, else we append the rule to the list of found rules. We repeat this process until the list of rules covers all instances in the dataset.

Computing the effect of a rule needs a pass over all n instances. Since at least one instance is covered in every iteration, the outer loop is called at most n times. Finding a rule with one term respectively

Algorithm 8: Rule Discovery (GERD)

input : attribute vectors X, label vector z, confidence β
output: a list of rules R

$R \leftarrow [\,]$;
while $|X| > 0$ **do**
  $R' \leftarrow$ possible rules with one term;
  $r^* \leftarrow \arg\max_{r \in R'} \hat{e}(r, X, z, \beta)$;
  if $\hat{e}(r^*, X, z, \beta) \leqslant 0$ **then**
    $r^* \leftarrow$ default rule;
  else
    repeat
      $R' \leftarrow$ possible one term extensions of r^*;
      $r^* \leftarrow \arg\max_{r \in R' \cup \{r^*\}} \hat{e}(r, X, z, \beta)$;
    until r^* remains unchanged;
  append r^* to R;
  remove instances from X, z covered by r^*;
return R

extending a rule with one term scales linearly in the number of attributes $|A|$. Since we require an improvement over $\hat{e}$ in every iteration of the inner loop, the number of newly covered instances by a rule is at least as high as the number of inner iterations. In other words, each additional inner iteration reduces the number of outer iterations. This leads to a worst-case runtime complexity of $O(n^2 \cdot |A|)$.

4.5 RELATED WORK

Event sequence prediction is a broadly studied topic. Much work deals with predicting the next event in a sequence based on past events, without considering additional metadata. This includes association rule mining [128], Markov models [14] and pattern mining [162].

Recent work using metadata to predict sequences is mostly based on neural networks. The approaches mainly differ in their chosen feature encoding and their concrete network architecture. Proposed methods include Long short-term memory (LSTM) networks [58] with one-hot

encoding [149], LSTMs with embedding techniques [24], convolutional neural networks [105] and adversarial LSTMs [150].

While some research applies approaches from explainable artificial intelligence [95] to neural networks for business process prediction, only a limited amount of work focuses on more accessible models. One recent exception is the data-aware transition system DATS [113]: The observed prefixes of event sequences are used to create a state machine. Which prefixes are mapped to which state is determined by a *state abstraction function*. Examples are the *list* function, where each unique prefix is mapped to its own state, and the *set* function where prefixes with the same set of events are mapped to the same state. For predicting future events given an attribute vector, a Naïve Bayes classifier estimates the transition probabilities between states and the path with the highest probability is predicted.

Inferring state machines from event data is also studied in software engineering with the goal of anomaly detection and test case generation but not sequence prediction [88, 154]. Model complexity and interpretability play a minor role in these approaches and software traces are usually much less noisy than business process traces.

Data-to-text generation deals with the creation of text, which can be seen as a kind of event sequence, from data. While much of today's work is based on neural networks, traditional approaches generate text by handcrafted rule-based templates for sentences [50]. First work tries to reduce the manual effort in creating those templates, but human supervision is still required [80].

Outstanding from the above, ConSequence enables sequence generation and prediction from data with an accessible white-box model, while requiring no handcrafted templates or rules and with minimal need for hyperparameter tuning.

4.6 EXPERIMENTS

In this section, we evaluate ConSequence on both synthetic and real-world datasets. We measure accuracy of sequence prediction by Levenshtein similarity [85] between actual and predicted sequence. Let $\rho(y_1, y_2)$ be the minimal number of deletions, insertions and replacements needed to transform sequence y_1 into sequence y_2. *Normalized*

Levenshtein similarity is defined by $\text{NLS} = 1 - \frac{\rho(y_1,y_2)}{max(|y_1|,|y_2|)}$, i. e., $\text{NLS} \in [0,1]$ and $y_1 = y_2 \rightarrow \text{NLS} = 1$.

We run all experiments on a server with two Intel(R) Xeon(R) Silver 4110 CPUs, 128 GB of RAM and two NVIDIA Tesla P100 GPUs. Except for LSTM, which uses the GPU, we report wall-clock runtimes for single-thread execution.

4.6.1 *Baseline Methods*

We compare ConSequence to various baselines that show different strengths and weaknesses on our task. First, we compare to a data-to-sequence LSTM. To prevent overfitting, we use dropout and early-stopping. We tune the size of the network, the batch size and the amount of dropout by conducting five runs with a random search on the hyperparameter values. We report results for the model with the highest accuracy on a hold-out validation set.

To show that the order of events matters, i. e., it is not sufficient to just predict occurrence and frequency of events, we also compare to the rule-based multilabel classificator Boomer [120]. To get a simple baseline for ordering the predicted multiset of events, we put events to positions in the sequence, where we have seen them most frequently in the training set. Formally, we solve the linear program

$$
\begin{aligned}
\text{maximize} \sum_{i}^{n}\sum_{j}^{n} c_{ij}x_{ij} & \\
\text{subject to} \sum_{i}^{n} x_{ij} = 1, & \qquad j = 1,\ldots,n \\
\sum_{j}^{n} x_{ij} = 1, & \qquad i = 1,\ldots,n \\
x_{ij} \in \{0,1\}\,, &
\end{aligned}
$$

where n is the number of events in the multiset, c_{ij} is the number of times event i occurs at position j, $x_{ij} = 1$ means event i is set to position j, and the constraints ensure, that all positions in the sequence will be filled and no event will be set to more than one position.

Since Boomer uses ensemble learning to infer the set of rules, its focus is on prediction accuracy and less on model complexity. Boomer

has a hyperparameter to prune individual rules, which we activate, such that single rules do not have unnecessarily many condition terms. We additionally post-process the rule set by removing the rules with the lowest weight in the ensemble until prediction accuracy on a held-out validation set decreases.

As additional baselines, we use KNN that predicts the event sequence with a k-nearest neighbor classifier ($k = 5$), and DATS as described in the related work section. To build the transition system for DATS, we use *list state abstraction*, since it shows best NLS in our experiment setting, which matches the results in the original paper [113]. When preparing the experiments, we observed that DATS struggles with complex event logs, which results in large and overfitting transition systems. Therefore, we preprocess the data before building the transition system: Let y^* be the most frequent sequence in the dataset and $\operatorname{supp} y$ denote the support or absolute frequency of y. Then, for a given threshold $\alpha \in [0, 1]$, we only keep sequences in the dataset with $\frac{\operatorname{supp} y}{\operatorname{supp} y^*} \geqslant \alpha$. In our experiments, we try $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ and report results for the run with highest NLS.

As our last baseline, PosCl, we use GERD to predict events for each possible position in the sequence. This means, if the longest sequence in the training set has length 100, then we learn 100 independent classifiers. The length of the predicted sequence is determined by the lowest positioned classifier that predicts *end-of-sequence*.

4.6.2 *Synthetic Data*

First, we evaluate on synthetic data, such that we know and can control the ground-truth model of the data.

Generation of Synthetic Data and Models

To generate one synthetic dataset and ground-truth model, we first generate categorical and numerical attributes in the dataset. For each metadata instance in the dataset, we sample a value for each of its categorical attributes from a discrete uniform distribution over a given number of possible categories. For numerical attributes, we sample from a uniform distribution over the interval $[0, 1]$.

In the second step, we generate an event-flow graph as ground-truth model. To this end, we generate a list of nested if-then-rules, e. g.,

```
IF num-feature-3 > 0.043 THEN
   Append 7, 19
ELSE
   Append 2, 0, 6
IF cat-feature-1 = 3 THEN
   Append 1, 11, 17
ELSE
   IF cat-feature-2 = 2 THEN
      Append 11
Append 1 ,
```

which we then convert to an event-flow graph with branching rules. Finally, we use the generated ground-truth model to predict sequences for the generated metadata instances.

Our implementation of this synthetic data generator allows to adapt many hyperparameters to control complexity of both the dataset and ground-truth model. Examples include the number of instances, the number of categorical and numerical attributes, the number of categories per categorical attribute, and the length and depth of the if-then-rules. For further details, we refer to our code and documentation.

Results on Synthetic Data

In our first experiment, we test noise-robustness. We independently generate ten synthetic datasets and divide each of them into a training set with 8000 instances and a test set with 2000 instances. Then, we inject noise into the training data. We consider destructive noise, where we remove each event in the dataset with some probability p, and additive noise, where at each position in the dataset, we add a random event from Ω with probability p. In Figure 4.2, we show the average NLS on the test set for various sequence predictors dependent on the amount of noise injected into the training set. We see ConSequence performs well under realistic amounts of noise, especially being more robust to noise than its competitors.

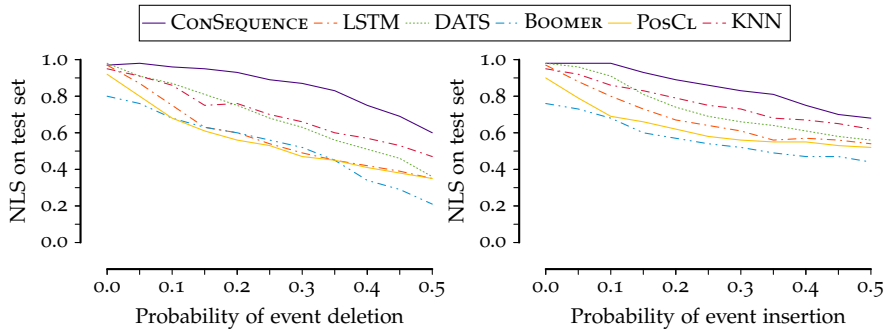


Figure 4.2: [**ConSequence is noise-robust**] NLS (higher is better) on test set dependent on the amount of destructive (left) and additive noise in the training set (right).

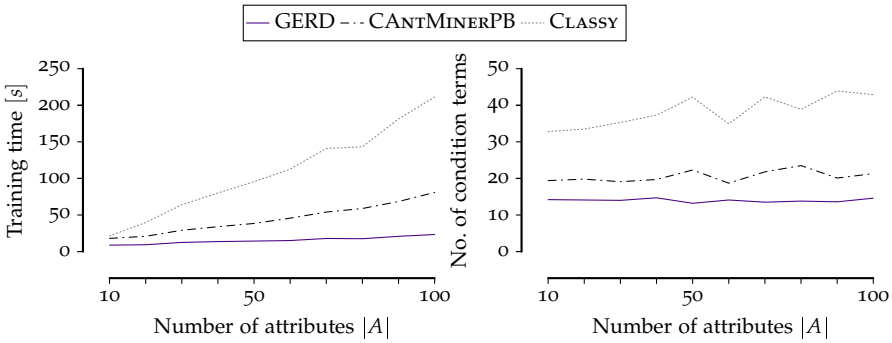


Figure 4.3: [**GERD is fast and produces small models**] Runtime in seconds (left) and model complexity in number of condition terms (right) for GERD, CAntMinerPB and Classy on synthetic data with a varying number of attributes.

While in theory, we allow any classifier for the prediction of successors in the event-flow graph, we propose using GERD for good reason. In Figure 4.3, we compare the use of ConSequence with three different rule-based classifiers. Besides GERD, we examine CAntMinerPB [103], which uses an ant colony optimization to mine a decision list with low prediction error, and Classy [119], which uses MDL to select a set of classification rules. GERD produces models with a lower number of decision terms, which are thus easier to be understood by humans. Furthermore, due to its scaling behavior, it deals with a larger number of attributes in a lower amount of time than its competitors.

Data	n	n_u	avg $\lvert y \rvert$	$\lvert \Omega \rvert$	$\lvert A \rvert$
Production	255	209	13	57	2
Sepsis	782	733	19	18	23
Rolling Mill	49233	1639	39	270	175
Software	500	200	151	24	12

Table 4.1: **[Real-world dataset statistics]** Number of sequences n, number of unique sequences n_u, average sequence length $\lvert y \rvert$, event alphabet size $\lvert \Omega \rvert$, and number of attributes $\lvert A \rvert$ for four real-world datasets.

Data	ConSequence	LSTM	DATS	Boomer	PosCl
Production	24s	82s	1s	3s	4s
Sepsis	2.5m	6m	5s	13s	16s
Rolling Mill	5h	5h	13m	2.5h	14d
Software	16m	6m	2s	5s	37s

Table 4.2: **[Runtimes]** Mean runtime on the training set of four real-world datasets for ConSequence, LSTM, DATS, Boomer and PosCl.

4.6.3 *Real-World Data*

To show ConSequence performs well in practice, we now evaluate on four real-world datasets with different properties as summarized in Table 4.1. Production [86] is a collection of event sequences from a production process, which contains only 255 relatively short sequences, from which 209 are unique. Sepsis [89] contains trajectories of Sepsis patients in a Dutch hospital. We filter out incomplete sequences and only consider attributes which are available at the beginning of a sequence. Rolling Mill is a manufacturing event log of a German steel producer. It stands out with its high number of instances, unique events and attributes. Software, the last dataset, is a profiling log of the Java program *density-converter* [40] that takes image files as input and converts them to different formats and densities, such that they can be used on different target platforms like Android or iOS. The events in this dataset refer to classes called during program execution, and the attributes refer to command line arguments.

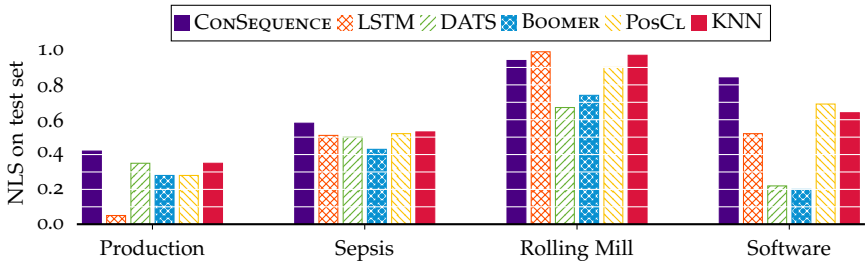


Figure 4.4: [**ConSequence predicts well**] Mean normalized Levenshtein similarity (higher is better) on the test set of four real-world datasets for ten independent runs on a random 80% train-test-split.

We run Boomer, PosCl, DATS, LSTM and ConSequence ten times on these datasets with a random train-test-split of 80%. We give an overview of the runtimes of all competitors for discovery on the training set in Table 4.2, except for KNN, which has no training time. While ConSequence is not the fastest, it still runs within a reasonable amount of time. In particular, its single-thread runtime is comparable to training an LSTM. Since we learn a list of classification rules at each branch in the event-flow graph, we can trivially parallelize ConSequence to reduce the overall runtime.

To evaluate accuracy of the predicted sequences, we report NLS on the testset of all methods in Figure 4.4. ConSequence achieves the highest NLS on the testset for datasets with few instances, and especially outperforms other methods on the Software dataset with a large average sequence length. As expected, the black-box LSTM performs well with a large training set, which is only available for the Rolling Mill data, while ConSequence still beats the other white-box approaches by a large margin.

Next, we evaluate the model complexity of all white-box methods. Since both ConSequence and DATS produce graph-based models of the event sequences, we count the number of nodes to measure model complexity, and show results in the left plot of Figure 4.5. ConSequence produces graphs with fewer nodes and thus better understandable models on the Production, Sepsis and Rolling Mill dataset. On the Software dataset, ConSequence outputs a significantly larger graph. This complexity, however, is justified by the significantly higher NLS of ConSequence on this dataset compared to DATS.

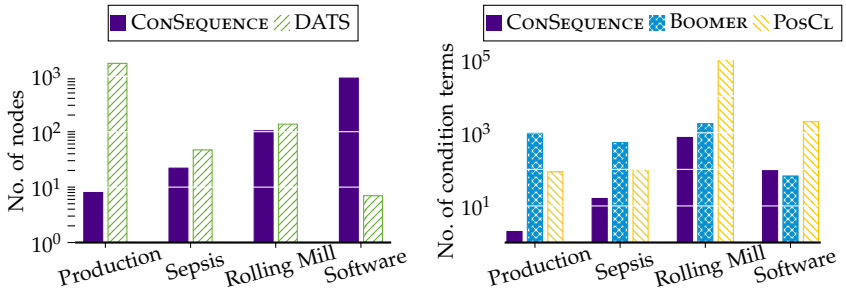


Figure 4.5: **[ConSequence discovers simple models]** Number of nodes in the event-flow graph of ConSequence versus number of nodes in the transition system of DATS discovered on four real-world datasets (left, lower is less complex) and number of conditition terms in the rules discovered on four real-world datasets for ConSequence, Boomer and PosCl (right, lower is less complex).

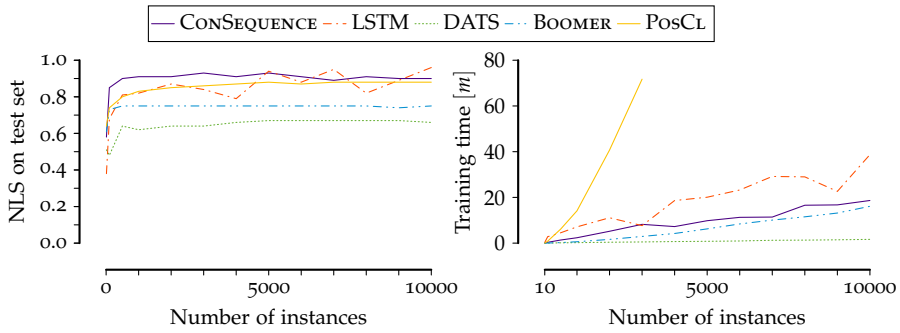


Figure 4.6: **[ConSequence has low sample complexity and scales linearly]** Normalized Levenshtein similarity on the test set (left, higher is better) and training runtime in minutes (right) depending on the number of training instances in the Rolling Mill dataset.

Since ConSequence, Boomer and PosCl produce rule-based models of the event sequences, we count the number of condition terms in the rules to measure model complexity, and show results in the right plot of Figure 4.5. We see the models by ConSequence generally have less condition terms than found by Boomer and PosCl.

To empirically evaluate sample complexity, we report how the size of the training set impacts the training time and the NLS on the test set in Figure 4.6. ConSequence already achieves its best performance with 1000 training instances in the Rolling Mill dataset. Although there are

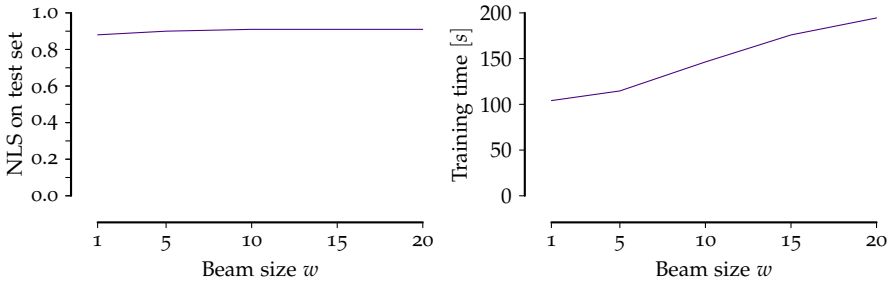


Figure 4.7: [**The cover algorithm has low sensitivity to the beam width parameter**] Normalized Levenshtein similarity on the testset (left) and runtime (right) for the cover algorithm on 1000 instances of the Rolling Mill dataset with varying beam width parameter w.

certainly faster methods such as DATS, ConSequence shows a linear scaling behavior regarding the number of instances. Together with the low sample complexity, this enables applicability on a wide range of real-world datasets.

4.6.4 *Hyperparameter Sensitivity*

In this section, we evaluate the hyperparameter sensitivity of ConSequence. Since we modeled the problem using MDL, we are almost free of hyperparameters. The cover algorithm we give as Algorithm 6 contains a beam width parameter, that provides a trade-off between runtime and quality of the search. In GERD, we have a confidence parameter β to control robustness of the discovered classification rules.

First, we examine the influence of the beam width parameter w in the cover algorithm, and report the results in Figure 4.7. We see that the choice of the beam width has only marginal influence on the prediction accuracy in terms of NLS on the testset. For the Rolling Mill dataset, we observe a slight increase of the NLS from 0.88 for $w = 1$ to 0.91 for $w = 20$; however, $w = 10$ is already large enough to reach a NLS of 0.91. Obviously, the choice of w largely impacts the training time of ConSequence, because a greater beam width increases the number of necessary computations in the cover algorithm. Since the relationship between training time and beam size looks exponentially, we conclude that the used heuristic in the cover algorithm does its job, i. e., steering the search into the right direction.

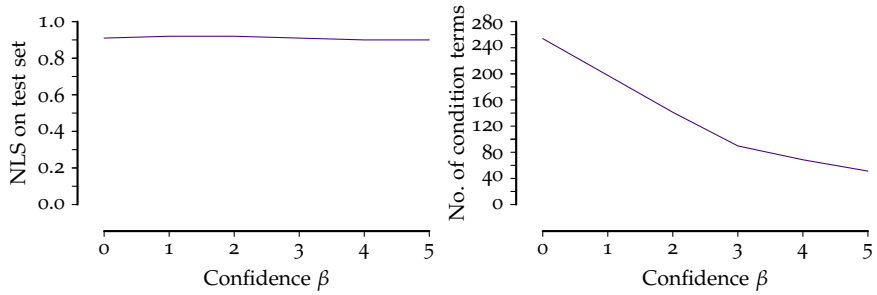


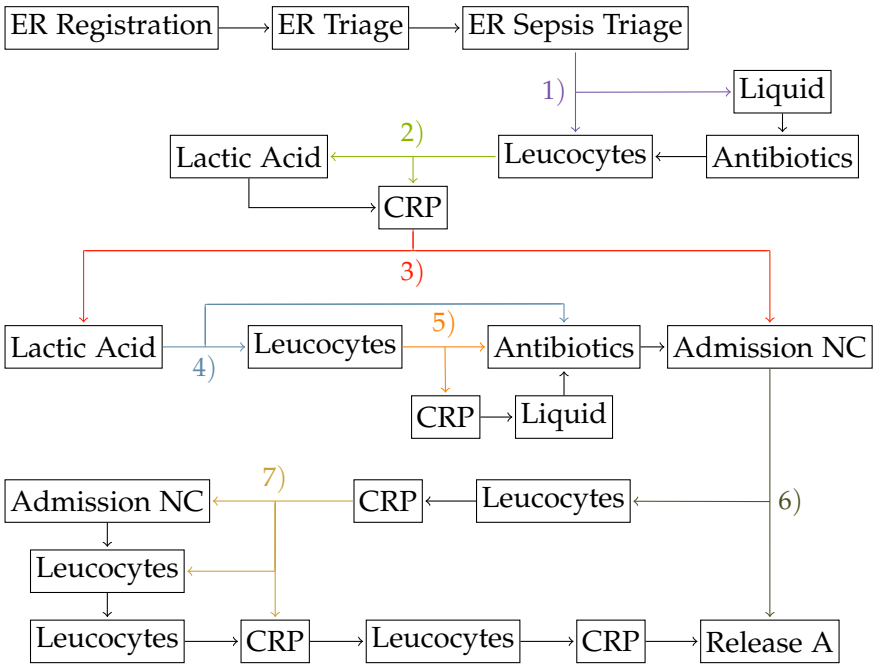
Figure 4.8: **[Hyperparameter sensitivity of GERD]** Normalized Levenshtein similarity on the testset (left) and model size in terms of condition terms (right) for ConSequence using GERD on 1000 instances of the Rolling Mill dataset with varying confidence β.

Next, we evaluate the influence of GERD's confidence parameter β and report results in Figure 4.8. We see the NLS on the test set is relatively independent of β. However, β has a huge impact on the number of discovered condition terms: A lower β leads to significantly bigger models, because adding rules into the model requires more evidence.

To produce the results in our experiments, we set $w = 10$ and $\beta = 2.0$. We especially choose $\beta = 2.0$, because the same value is used by the authors who proposed the rule effect estimator $\hat{e}$, where $\beta = 2.0$ corresponds to a 95.45% confidence level [23].

4.6.5 *Case Study on Sepsis Dataset*

Next, we show that ConSequence finds an understandable and meaningful model of the Sepsis dataset, and show its discovered event-flow graph in Figure 4.9. One can clearly recognize the typical flow of a Sepsis patient in the hospital. The process starts with the arrival in the emergency room (ER). If the patient has an Oligurie (malfunction of kidneys) or for other reasons needs an infusion, he or she is provided with liquid and antibiotics. In any case, leukocytes are counted for further diagnosis. After admission to normal care (NC), patients without certain symptoms, which are potentially younger, are released soon, while other patients need further treatment.



1) Oligurie ∨ Infusion → Liquid
⊤ → Leucocytes

2) Oligurie ∨ Hypotensie → Lactic Acid
⊤ → CRP

3) Infusion → Lactic Acid
⊤ → Admission NC

4) ¬Infusion → Antibiotics
⊤ → Leucocytes

5) DiagnosticArtAstrup → CRP
⊤ → Antibiotics

6) ¬SIRS2OrMore ∨ Age ⩽ 34 → Release A
⊤ → Leucocytes

7) Age ⩽ 27 → CRP
DisfuncOrg → Leucocytes
Hypotensie ∧ Oligurie → Admission NC
⊤ → Leucocytes

Figure 4.9: **[Model for Sepsis Dataset]** Event-Flow Graph as found by ConSequence on the Sepsis dataset.

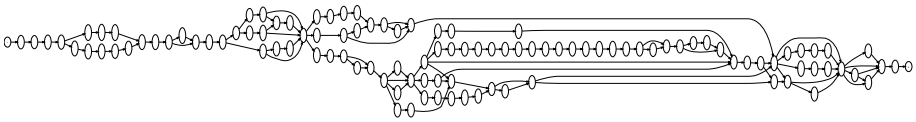


Figure 4.10: **[Event-flow graph of the Rolling Mill]** Event-flow graph without rules found by ConSequence on the Rolling Mill dataset.

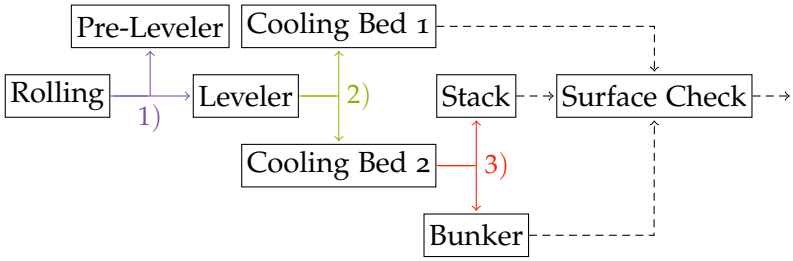
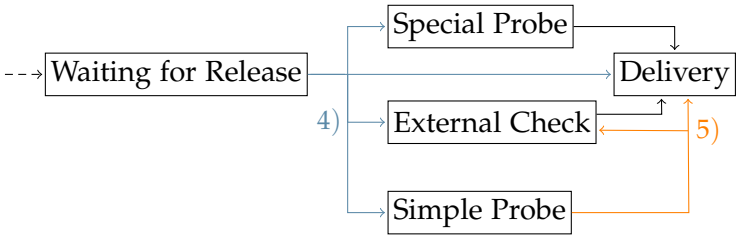


Figure 4.11: **[Model for the Hot Zone of the Rolling Mill]** Excerpt showing the source part of the model found by ConSequence on the Rolling Mill dataset. Dashed arrows indicate skipped nodes.

4.6.6 *Case Study on Rolling Mill Dataset*

Finally, we show details of the model found by ConSequence on the Rolling Mill dataset. First, we depict the complete event-flow graph without rules in Figure 4.10. We clearly see that the model is well-structured and easy-to-follow. However, the graph is too large to discuss all rules in full detail. Therefore, we provide a closer look on the beginning and the end of the graph.

We show the source part of the Rolling Mill model in Figure 4.11, and explain the beginning of the underlying rolling process. First, the plates are rolled at rolling-stands to meet their customer defined thickness. This happens at high temperatures and forces, otherwise, thickness reduction would not be possible. Therefore, this part of the rolling mill is called *hot zone*. The large rolling stands cannot completely level the plate surface, which is compensated by a special leveler. In addition, plates with a special accelerated cooling (ACC) treatment or with a special requirement on their use, need a pre-leveler activity (rule 1).



4) Simple Probe Requested → Simple Probe
Special Probe Requested → Special Probe
Inspection ∈ {WWF, WFF, FFF} → External Check
⊤ → Delivery
5) Inspector = Dillinger → Delivery, ⊤ → External Check

Figure 4.12: **[Model for the End Part of the Rolling Mill]** Excerpt showing the end part of the model found by ConSequence on the Rolling Mill dataset. Dashed arrows indicate skipped nodes.

After leveling, plates need to cool down before further processing. At this rolling mill, production splits up into a part for thicker plates and a part for thinner plates, which both have their own cooling beds (rule 2). After the cooling bed, production of thicker plates temporarily splits into interim storage of plates into stacks and bunkers (rule 3). While thin and thick plates flow through different parts of production, both type of plates go through a surface check.

Eventually, plates wait at the end of the rolling mill for release as we show in Figure 4.12. Before a plate can be delivered to the customer, different probes must confirm that the plate meets the product quality requirements (rule 4). For some plates, an external inspector conducts additional checks (rule 5). Delivery is the last activity in the process.

4.7 DISCUSSION

In our experiments, we showed ConSequence finds succinct and understandable models that yet accurately predict event sequences from meta data. Nonetheless, we see many interesting directions to extend and improve ConSequence. First, we choose GERD to predict the next node in the event-flow graph, because it gives a good trade-off between speed, noise-robustness and sample complexity. Other classifiers may

lead to better results depending on the amount of training data, the number of attributes, and the degree of noise present in the data.

To explain and predict more complex behavior, it is promising to extend the event-flow graph model by using event patterns such as loops and concurrency. Instead of predicting one single event sequence, we could discover rules modeling the stochastic behavior, where multiple routing choices are possible. While ConSequence focuses on predicting whole event sequences from sequence meta data, predicting running cases given an event sequence prefix and event meta data may be an interesting extension.

Besides explaining and predicting event sequences, we see many applications for event-flow graphs. Using the discovered event-flow graph with rules as a definition of normal behavior builds the foundation of explainable anomaly detection. Many factories use business rule engines for production planning, which includes planning event sequences of production steps. Maintaining a large set of rules, where rules permanently change to meet new requirements in the process, is a challenging task. Mining event-flow graphs from actual data and from planning data, together with our MDL score, could help to identify redundant and unnecessarily complex rules.

Last but not least, we see how event-flow graphs integrate into process simulation to predict and prevent bottlenecks during production planning. Having an executable process model that is able to predict waiting and service times of different stations in the process, ConSequence can predict the sequence of stations for a given set of process cases. We elaborate this idea in Chapter 7.

4.8 CONCLUSION

We studied the problem of accurate yet interpretable sequence prediction from data. For this, we modeled event sequences with directed graphs and discovered classification rules to explain the relationship between attributes in the dataset and paths in the graph. We formalized the problem in terms of the MDL principle, i. e., the best model is the one that compresses the data best. As the resulting optimization problem is NP-hard, we proposed the efficient ConSequence algorithm to discover good models in practice.

Through an extensive set of experiments including a case study, we showed that our approach indeed produces compact, interpretable and accurate models, is robust against noise and has low sample complexity, which enables applicability on a wide range of real-world datasets.

Future work might extend ConSequence to more applications like prediction of running cases given an event sequence prefix, where metadata belongs to events instead of the whole sequence. A richer modeling language for event-flow graphs, using patterns instead of single event nodes, could result in even more succinct models, that better fit complex behavior like concurrent events.

5 PREDICTING SOJOURN AND WAITING TIMES

Knowing the flow of events in a process and being able to predict event trajectories, we now focus on the time dimension of processes. More specifically, we aim to discover interpretable queueing models that explain and predict sojourn and waiting times.

5.1 INTRODUCTION

We have all stood in a waiting line, wondering why is it taking so long and how much longer we have to wait. Explaining and predicting waiting times is a highly relevant topic in service-oriented and manufacturing processes. Process time prediction methods [113, 150] usually assume independence between jobs and neglect varying waiting times due to queueing. On the contrary, waiting time is the core concept of queueing models [129], in which servers process incoming jobs. If all servers are busy, arriving jobs must wait until a server becomes available. Although queueing models have been used in many domains such as customer service, traffic control, manufacturing and healthcare [53], modeling processes typically involves intensive handcrafting by domain experts, which often results in idealized models that do not fit the actual process behavior well [1].

Existing approaches to discover queueing models from data [138, 139] are restricted to first-come first-serve order, which results in poor fitness on processes with different behavior. They generally discover only one specific part of a queueing model, such as number of servers [66] or batch sizes [69], and require expert knowledge for the remaining parts. Neural networks can predict service times with high accuracy [101]; however, they require large training datasets and extensive

This chapter is based on [160]: Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Why Are We Waiting? Discovering Interpretable Models for Predicting Sojourn and Waiting Times." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Minneapolis, MN*. 2023, pp. 352–360.

hyperparameter tuning. Their black box nature impedes what-if analysis like what happens if we increase the number of available servers.

In practice, we frequently face datasets with arrival and departure times of jobs, but without any knowledge about the underlying waiting and service times [138]. We propose a novel approach to discover interpretable queueing models with rich modeling language from such data. To this end, we formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we identify the best model as the one giving the shortest lossless description of the data. Since the resulting optimization problem is computationally hard, we propose our greedy algorithm CUEMIN (a pun of the spice cumin and cue miner) to find good queueing models in practice. CUEMIN discovers the key parts of a queueing model, i. e., service order, number of servers, batch sizes and service time. Furthermore, CUEMIN considers additional features in the data, such as the type of product in manufacturing, to explain service order and to predict service time.

Through extensive experiments on both synthetic and real-world data including a case study on call center data, we show that CUEMIN in contrast to the state of the art discovers inherently interpretable models, which explain and predict behavior of waiting line processes. Our main contributions are as follows. We

(a) formulate the problem of discovering queueing models in terms of the MDL principle,
(b) propose an efficient heuristic to find interpretable yet accurate models to predict waiting and sojourn time from data,
(c) perform an extensive empirical evaluation,
(d) make code and data publically available.[1]

The structure of this chapter is as follows: Next, we introduce the necessary notation of queueing models in Section 5.2. In Section 5.3, we formalize the problem of discovering a queueing model from data in terms of MDL. Afterward, we propose our algorithm CUEMIN in Section 5.4. In Section 5.5, we give an overview of related work, before we empirically evaluate CUEMIN in Section 5.6. We discuss potential future work in Section 5.7 and conclude this chapter in Section 5.8.

1 https://eda.rg.cispa.io/prj/cuemin

5.2 NOTATION FOR QUEUEING MODELS

In queueing theory [53, 129], a queueing model M consists of c servers that process incoming jobs. We denote the arrival time of the i-th job as $a_i \in \mathbb{N}$. If all servers are busy, jobs must wait until a free server is available. We use $w_i \in \mathbb{N}$ to refer to the waiting time of job i. Servers can process jobs in batches. We refer to B as the univariate discrete batch size distribution. If the current batch size is $k \sim B$, a server waits until k jobs are available to get served.

We model with different univariate discrete distributions [64]. Favoring a concise notation, we write, whenever clear from context, $\mathrm{P}(x)$ instead of $\mathrm{P}(X = x)$ for the probability mass function (pmf). Below, we consider four distributions which are particularly commonly used in queueing theory. We note, however, that our theory accepts any distribution with a pmf of a finite set of parameters. The simplest distribution we consider is a degenerate, or *fixed* distribution $\mathrm{F}(k)$. We use it to model constant values, because it has only support for a single value $k \in \mathbb{N}$, i. e., $\mathrm{P}(k) = 1 \wedge \forall x \neq k : \mathrm{P}(x) = 0$. As a more flexible distribution, we denote the Poisson distribution with expected value k as $\mathrm{Pois}(k)$ with pmf $\mathrm{P}(x) = \frac{e^{-k}k^x}{x!}$. For a geometric distribution with success probability p and pmf $\mathrm{P}(x) = (1-p)^{x-1}p$, we write $\mathrm{Geo}(p)$. The negative binomial distribution $\mathrm{NB}(k, p)$ with number of successes k and success probability p has pmf $\mathrm{P}(x) = \binom{x+k-1}{k-1}p^k(1-p)^x$.

Servers process jobs in service order R. We consider first-come, first-served (FCFS), last-come, first-served (LCFS) and priority queueing (PQ), where jobs own priority classes and jobs with same priority are either served FCFS (PQ + FCFS) or LCFS (PQ + LCFS). The service time of a job is drawn from the service time model S. In its simplest form, S is a univariate discrete distribution. If service time depends on additional features like heavier products in manufacturing need more time, S can be a regression function $f_S : \mathbb{R}^m \to \mathbb{R}$, plus an additive error distribution E_S. Load on the system may lead to different service times. Therefore, S can consist of k submodels $S'_1, \cdots, S'_k$, where $k-1$ load thresholds $\tau_1, \cdots, \tau_{k-1} \in \mathbb{N}$ define when to use which submodel. If the number of jobs in the queue is between τ_{j-1} and τ_j, service time is predicted by S'_j. We denote the service time of the i-th job as s_i.

When a job has been served, it leaves the system. We denote the departure time of the i-th job as $d_i = a_i + w_i + s_i$. The sojourn time

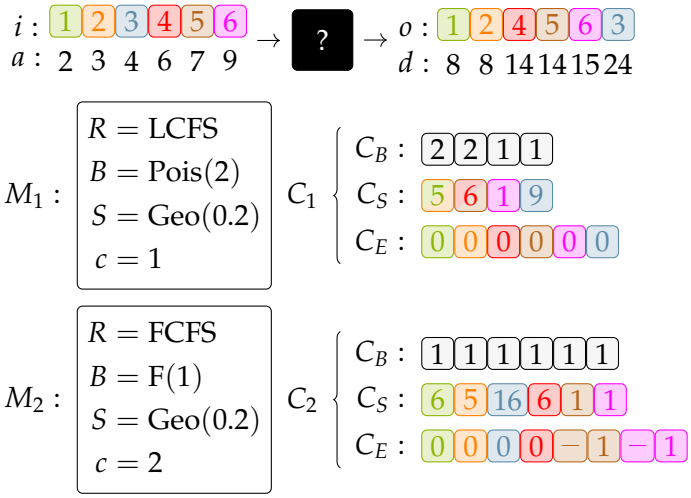


Figure 5.1: **[Data Encoding]** Toy example of jobs (top) with arrival times a, departure times d, and two possible processing models M_1 and M_2 and their corresponding data encodings C_1 and C_2.

v is the time span between arrival and departure, i. e., $v_i = d_i - a_i = w_i + s_i$. This gives us an interpretable yet powerful model to explain and predict waiting, service and sojourn times.

5.3 MDL FOR QUEUEING MODELS

We favor queueing models that are simple and interpretable yet at the same time are sufficiently rich to fit real-world process behaviors. Therefore, we formalize the problem of discovering a queueing model in terms of the MDL principle. We encode the data given a model with codes in a code stream or *cover* C, where we specify how the model serves arriving jobs. Conceptually, we split C into three streams: C_B encodes the batch sizes in which jobs are served, C_S encodes service times, and C_E encodes errors to ensure a lossless encoding.

We show a toy example of data, model and cover in Figure 5.1. Model M_1 consists of a single server that processes jobs in LCFS order, batch sizes are Poisson and service times are geometrically distributed. Now, we decode the departure times d from the arrival times a using cover C_1. We start by reading the size of the first batch from C_B, which tells us the next two jobs are served in a batch. Then, we read the ser-

vice time of this batch from C_S. Now, we know the server waits until the first two jobs arrive and needs five time steps to serve this batch, which results in a departure time of eight for both jobs. For each of the jobs, we correct the departure time if necessary by reading a code from C_E. In this example, we read 0, i. e., the observed departure time equals the departure time given by the model.

We continue by reading the next batch size, two, from C_B. When the server becomes free at time step eight, job 3, 4 and 5 are waiting. Due to LCFS order, job 4 and 5 are served next. We read the code for service time six from C_S, which results in departure time 14. The next two codes in C_E tell us that 14 is correct. We decode the remaining two jobs as before and are done.

In model M_2, we have two servers processing jobs in FCFS order with batch size one. Now, we use cover C_2 to decode departure times of the arriving jobs. Job 1 and job 2 are served in batches of size one using separate servers. After the first two jobs leave, job 3 blocks one server until the end of our example, and when job 4 leaves at time step 14, job 5 requires zero service time. However, the geometric service time distribution of M_2 does not allow zeros. Therefore, C_S contains a code for service time one, which is corrected by two codes in C_E, giving us sign and magnitude of the error. We decode job 6 analogously by which we decoded all jobs.

5.3.1 *Data Encoding*

We define length of the data encoding as the sum of the code lengths in the code stream C. Formally, we have

$$L(D \mid M) = -\sum_{b \in C_B} \log \mathrm{P}_B(b) - \sum_{s \in C_S} \log \mathrm{P}_S(s) + \sum_{e \in C_E} L(e),$$

where we first encode the batch sizes with optimal prefix-free codes using the model's batch size distribution, then we encode the service times also with optimal prefix-free codes using the service time model, and we encode the errors of the modeled departure times, which ensures a lossless encoding as required by MDL.

We encode an error $e \in C_E$ by first encoding its sign $\operatorname{sgn} e \in \{-1, 0, 1\}$ and then its magnitude. If we knew the distribution of the signs beforehand, we could compute lengths of optimal prefix-free codes with

Shannon entropy. To avoid any arbitrary choices in the model encoding, we use prequential codes (see Section 2.3). Formally, we define the encoded length of the error by

$$L(e) = -\log \frac{\operatorname{usg}(\operatorname{sgn} e) + \epsilon}{\sum \operatorname{usg}(\cdot) + \epsilon} + \begin{cases} 0, & \text{if } e = 0 \\ L_{\mathbb{N}}(|e|), & \text{otherwise,} \end{cases}$$

where $\operatorname{usg}(\operatorname{sgn} e)$ denotes how often the code for $\operatorname{sgn} e$ has been used before, ϵ with standard choice 0.5 is for smoothing. This gives us a lossless encoding of the data.

5.3.2 *Model Encoding*

We encode all model parts separately. For the service order R, we use $\log 3$ bits to encode whether we have FCFS, LCFS or PQ. In case of PQ, we additionally encode, which of the categorical features in our dataset contains the priority classes, and encode the order of the categories by an index over all possible orders. Since multiple waiting jobs can have the same priority class, we use one bit to encode whether the default order is FCFS or LCFS. This results in

$$L(R) = \log 3 + \begin{cases} \log m_{cat} + \log k! + 1, & \text{if } R = \text{PQ} \\ 0, & \text{otherwise,} \end{cases}$$

where m_{cat} denotes the number of categorical features and k the number of categories of the chosen feature.

For the batch size distribution B, we specify the type of the distribution and encode its parameters. Distinguishing between four types of univariate discrete distributions costs two bits. In general, the distribution parameters have arbitrary values. Therefore, we encode integer parameters with $L_{\mathbb{N}}$ and real parameters with $L_{\mathbb{R}}$.

We defined three types of service time models S, i. e., encoding the type costs $\log 3$ bits. If S is a distribution, we encode it like the batch size distribution. In case of a regression function, we encode the parameters using $L_{\mathbb{R}}$ and encode the error distribution as before. If service time is load-dependent, we encode the number and values of the load thresholds using $L_{\mathbb{N}}$, and encode the submodels accordingly. We define the encoded length of the model by

$$\mathrm{L(M)} = \mathrm{L}(R) + \mathrm{L}(B) + \mathrm{L}(S) + L_{\mathbb{N}}(c),$$

which gives us a lossless encoding of the model.

5.3.3 *Formal Problem Definition*

We now have all necessary parts to formally state the problem.

Minimal Queueing Model Problem *Given a dataset D of arrival and departure times, find the queueing model M and cover C, such that the total encoded cost $L(D, M) = L(D \mid M) + L(M)$ is minimal.*

Finding the optimal cover for a given model is computationally hard: There is no product-form solution to compute the future state of a queueing model [67], i. e., whenever we choose batch size and service time at one point of the cover, we must compute the impact on all later time steps. Due to many valid choices of batch size and service time at each step, we face an intractable, exponentially growing search space.

Finding the optimal model is not easier. Without a product-form solution for queueing states, every change in the model requires recomputation of the cover. We cannot search for different parts of a model independently: Different service orders lead to completely different service times, and a change of batch sizes requires adapting service times or the number of servers. Hence, we resort to heuristics.

5.4 THE CUEMIN ALGORITHM

Since solving the minimal queueing problem is difficult, we divide it into two and propose greedy solutions for finding a cover and discovering a model separately.

5.4.1 *Finding a Cover*

To find a good cover, we first need to know which jobs are served as one batch, such that we can estimate the corresponding service times. Similar to the existing BatchMiner [69], we discover batches by jobs with the same departure time; however, we restrict batch sizes to values that we can explain by the batch size distribution B of the model.

We give the pseudocode of FindBatches as Algorithm 9. Initially, all servers j have empty batches at any time t (ln. 1). To consider all changes in the model state, we iterate over all arrival and departure

Algorithm 9: FINDBATCHES

input : dataset D, queueing model M
output: list of detected batches Z

1 $b_j^t \leftarrow \varnothing \; \forall j \in \{1, \ldots, c\}, t \in \{a_1, \ldots, d_n\}$;
2 **foreach** $t \in \{a_1, \ldots, d_n\}$ **do**
3 **foreach** job i waiting for M at time t **do**
4 **if** $\exists j : b_j^t \neq \varnothing$ **and** i should be in b_j^t **then**
5 $b_j^k \leftarrow b_j^k \cup \{i\} \; \forall k = t, \ldots, d_i$;
6 **else if** $\exists j : b_j^t = 0$ **then**
7 $b_j^k \leftarrow \{i\} \; \forall k = t, \ldots, d_i$;
8 **foreach** $j \in \{1, \ldots, c\}$ **do**
9 **if** $b_j^{t+1} = \varnothing$ **then**
10 add b_j^t to Z;
11 **return** Z;

times (ln. 2). In each iteration, we try to find a suitable batch for each job in the waiting line. We first check if we should add the job to an already existing batch (ln. 4): We add the job to a batch if the model demands a higher batch size ($P_B(|b_j^t|) = 0$), or if the job has the same departure time as the jobs in the batch ($b_j^{d_i} \neq \varnothing$) and the model supports a larger batch ($P_B(|b_j^t| + 1) > 0$). In this case, we mark the job to be part of batch b_j blocking server j until departure time d_i (ln. 5). If we could not add the job to an existing batch, we create a new batch if there is a free server (ln. 6-7). Whenever a batch has been processed, we add it to our list of detected batches (ln. 10).

Using FINDBATCHES, computing the cover is fairly easy. We give the pseudocode of FINDCOVER as Algorithm 10. Starting with an empty cover, we iterate over all batches found by FINDBATCHES. We add the size of the batch to C_B (ln. 3) and compute the required service time s to meet the departure time of the batch (ln. 4), where t denotes the current time. If the model cannot produce the required service time s, we replace s with the most likely service time of the model (ln. 5-6). Next, we add the code of s to C_S (ln. 7). Then, for each job in the batch, we compute the error on the departure time and add its code to C_E

Algorithm 10: FINDCOVER

input : dataset D, queueing model M
output: cover (C_B, C_S, C_E)

1 $C_B \leftarrow \emptyset, C_S \leftarrow \emptyset, C_E \leftarrow \emptyset, t \leftarrow 0$;
2 **foreach** $b \in \text{FINDBATCHES}(D, M)$ **do**
3 append $\boxed{|b|}$ to C_B;
4 $s \leftarrow d_{b_1} - t$;
5 **if** $\mathrm{P}_S(s) = 0$ **then**
6 $s \leftarrow \arg\max \mathrm{P}_S(\cdot)$;
7 append $\boxed{s}$ to C_S;
8 **foreach** $i \in b$ **do**
9 $e_i \leftarrow s + t - d_i$;
10 append $\boxed{\operatorname{sgn} e_i}$ to C_E;
11 **if** $e_i \neq 0$ **then**
12 append $\boxed{|e_i|}$ to C_E;
13 $t \leftarrow$ service start time of next batch;
14 **return** (C_B, C_S, C_E);

(ln. 8-12). At the end of each iteration, we set the current time t to the service start of the next batch (ln. 13). This gives us a cover for a model.

FINDBATCHES has runtime $O(nc)$, i. e., it scales linearly with the number of observed jobs n and the number of servers c. Hence, it is fast enough to compute sufficiently many covers during model search. The runtime of FINDCOVER is mainly driven by FINDBATCHES, i. e., $O(nc)$.

5.4.2 *Discovering a Good Queueing Model*

With FINDCOVER, we are able to compute our MDL score, which we now use for model selection. The overall idea is to discover a model for each possible service order and take the one with the lowest total encoding cost. While FCFS and LCFS are non-parametric, we select the most promising categorical attribute for priority queueing PQ. To this end, we choose the attribute and the permutation of its categories π that interpreted as priority classes minimize the conditional entropy on the departure order of jobs for the observed arrival order. Formally,

Algorithm 11: BRUTEFORCE

```
input : dataset D, service order R, upper bound on the number
        of servers c_max
output: queueing model M
1 M ← ∅;
2 foreach c ∈ {1, ..., c_max} do
3     M̂ ← (R, B, S, c);
4     foreach S, B ∈ FITSERVICE(D,R,c) do
5         if L(D, M̂) < L(D, M) then
6             M ← M̂;
7 return M
```

if Y is a random variable of the category of the next leaving job and X is a random variable of the predicted category by the order of π, we minimize $H(Y \mid X) = -\sum_{x \in X, y \in Y} \mathrm{P}(x, y) \log \frac{\mathrm{P}(x,y)}{\mathrm{P}(x)}$.

We restrict the search space by a simple yet effective upper bound on the number of servers, which is the smallest number of servers such that all jobs can be served without waiting time. Any greater number leads to unused servers and cannot be inferred from data.

BRUTEFORCE We propose two different search strategies to discover a model with a given service order. The first one is a naive brute-force search, for which we give the pseudocode as Algorithm 11. We generate service time and batch size distribution candidates for each possible number of servers, and select the model with the best score. We always generate the batch size distribution candidate $\mathrm{F}(1)$, i. e., batch size is constantly one. In addition, we use maximum likelihood estimation to fit a candidate for each type of distribution we introduced in Section 5.2 on the number of jobs with the same departure time. We then use the batch size distribution candidates to compute required service times for each job as we did in line 4 of FINDCOVER in Algorithm 10.

Next, we generate service time candidate models. We fit distributions by maximum likelihood estimation. For regression models with an error distribution, we first fit the regression model and then fit a distribution on the residuals. We generate load dependent service

Algorithm 12: CUEMIN

input : dataset D, service order R, upper bound on the number of servers $c_{\max}$
output: queueing model M

```
1  M ← ∅;
2  c ← 1, δ ← 1;
3  repeat
4      foreach S, B ∈ FITSERVICE(D,R,c) do
5          M̂ ← (R, B, S, c);
6          if L(D, M̂) < L(D, M) then
7              M ← M̂;
8      if L(D, M) improved then
9          δ ← δ · 2;
10     else if δ = 1 then
11         δ ← −1;
12     else
13         δ ← ⌈δ/2⌉;
14     c ← max {1, min {c_max, c + δ}};
15 until δ = 0;
16 return M
```

time models by finding $\tau_1, \ldots, \tau_{k-1}$ for multiple values of k, and then fit k submodels, i. e., distributions or regression models. For given k, we choose $\tau_1, \ldots, \tau_{k-1}$ such that we minimize $\sum_{i=1}^{k} \sum_{s \in K_i} (s - \bar{K}_i)^2$, with K_i being the i-th cluster of service times implied by $\tau_1, \ldots, \tau_{k-1}$ and $\bar{K}_i$ the average service time of the cluster.

By this, we discover a good queueing model M for a given dataset D. However, if $c_{\max}$ is large, BRUTEFORCE wastes a lot of runtime by searching through models with few servers.

CUEMIN We propose CUEMIN as an efficient alternative to BRUTEFORCE. Although our score is not strictly convex, it does exhibit convex-like behavior that we can exploit towards discovering good models. In particular, whenever a model contains *too few* servers, it will incur a heavy penalty because it has trouble serving jobs on time; adding more servers will reduce this penalty. In contrast, when a model has

too many servers, it also incurs a heavy penalty because it serves jobs too early; reducing servers reduces this penalty.

We give the pseudocode of CueMin as Algorithm 12. We start by finding the best model for one server. Whenever the current number of servers leads to an improvement of the MDL score, we increase the step size δ (ln. 9), which determines the next candidate number of servers (ln. 14). This way, if the number of optimal servers is large, we quickly jump over values, which are much too low. At some point, large steps do not lead to better models. We then start to decrease the step size (ln. 13). If increasing the number of servers does not have an effect anymore, we repeat the search in the opposite direction in case we jumped over the optimum (ln. 11).

As we show in Section 5.6, CueMin works well. It finds models as good as those by BruteForce, while being significantly faster. Domain experts can, if wanted, easily include knowledge into the search: They can restrict the number of servers to speed up the search or adapt any part of the model to their expectation. For instance, they can create a domain-specific service order or service time distribution.

RUNTIME COMPLEXITY The runtime of BruteForce is strongly dependent on the upper bound on the number of servers $c_{\max}$. In each of the $c_{\max}$ iterations, we compute our score with runtime complexity $O(nc)$, which results in a total runtime complexity of $O(nc_{\max}^2)$. Depending on the dataset and the underlying data generating process, $c_{\max}$ is so large that this runtime becomes unacceptable. In practice, we can either restrict the number of servers with domain knowledge or use heuristics to speed up the search.

Instead of exhaustively searching over all possible number of servers, CueMin skips unpromising candidates through its adaptive step size δ. Since our MDL score is not convex, in the worst case, every second search candidate improves the score. This means, δ alternates between one and two, and we test all $c_{\max}$ possible values of c. Hence, in the worst case, CueMin has runtime complexity $O(nc_{\max}^2)$, i. e., it falls back to BruteForce. In practice, however, CueMin is significantly faster than BruteForce as we show in Section 5.6.

5.5 RELATED WORK

Before we show our evaluation, we first give an overview of related work. Predicting event duration in business processes is closely related to our problem. Polato et al. [113] predict future events of running process instances based on a Naïve Bayes classifier and use support vector regression to estimate event duration. Deep learning leads to more accurate predictions; however, existing approaches [24, 150] equally neglect dependencies between jobs and thus cannot consider increased waiting times due to queueing effects in the process.

Surprisingly little work deals with the discovery of queueing models from data. Senderovich et al. [139] coin the term *Queue Mining* and pioneer in synthesizing data mining and queueing theory to predict delays in service processes. In addition to arrival times and departure times, they assume availability of observations on the waiting times.

In case waiting times are not available, Senderovich proposes K-PHF for waiting time prediction from arrival and departure times only [138]. Assuming FCFS order, K-PHF clusters sojourn times by K-MEANS and infers a phase-type distribution for each cluster. The clusters correspond to different load states such as low, moderate and high.

Unfortunately, K-PHF does not provide any information about batch service or the number of servers. Keith et al. [66] propose COrder to estimate the number of servers in a FCFS queue. They also propose a LCFS version of COrder; however, one needs to know the service order to choose the right version.

Klijn and Fahland [69] detect service batch sizes through jobs with close departure time, but do not consider other queue modeling aspects. Ojeda et al. propose RAS [101] to combine queueing theory and adversarial deep learning with Wasserstein loss [9] to predict sojourn times and their distribution from an embedding of arrival times [38] and covariates of jobs in a queue.

In contrast to the above, CueMin finds models for the prediction of process behavior, where all parts of the model are based on interpretable building blocks from queueing theory. To the best of our knowledge CueMin is the first method that jointly discovers batch service, service order, number of servers and service time.

5.6 EXPERIMENTS

Now, we evaluate CUEMIN on both synthetic and real-world datasets. We conduct all our experiments on a PC with an Intel i7-6700 CPU and 32 GB of memory, running Windows 10. We report wall-clock running times for single-threaded execution, except for RAS, which uses our GeForce RTX 2080 Ti during training.

5.6.1 *Synthetic Data*

We start with experiments on synthetic data. We sample 1000 ground-truth models with $R \in \{\text{FCFS}, \text{LCFS}\}$, $c \in [1, 30]$, $B = \text{F}(1)$ and a large set of different service time distributions. We generate data by sampling job arrivals from several interarrival distributions from which we sample departure times with the ground-truth models.

First, we evaluate CUEMIN's ability to find the ground-truth number of servers c compared to BRUTEFORCE, CORDER [66] and a naive baseline DELTAMAX [66]. Since intuitively more servers are needed to explain a greater difference between arrival and departure order, DELTAMAX enumerates arriving jobs from 1 to n, and estimates c by the maximal index difference of consecutively leaving jobs. For a fair comparison, we feed our knowledge of the ground-truth service order into CORDER, whereas CUEMIN and BRUTEFORCE additionally have to discover the service order.

We report the mean absolute error (MAE) on estimating the number of servers on the left of Figure 5.2. Computing the MAE dependent only on the number of observed jobs to evaluate sample complexity would overpenalize complex ground-truth models with more servers, which need more jobs to utilize all servers. We therefore normalize the number of jobs by the number of ground-truth servers. As expected, all methods improve with more data. Although we give CORDER the advantage of knowing the ground-truth service order, we see CUEMIN and BRUTEFORCE have a competitive MAE and significantly beat the naive DELTAMAX baseline. CUEMIN produces almost equivalent results to the exhaustive search by BRUTEFORCE.

Next, we show the MAE dependent on the type of ground-truth queueing model on the right of Figure 5.2. The left group of bars (M_1) refers to the left line plot, i. e., the assumptions of CORDER still hold.

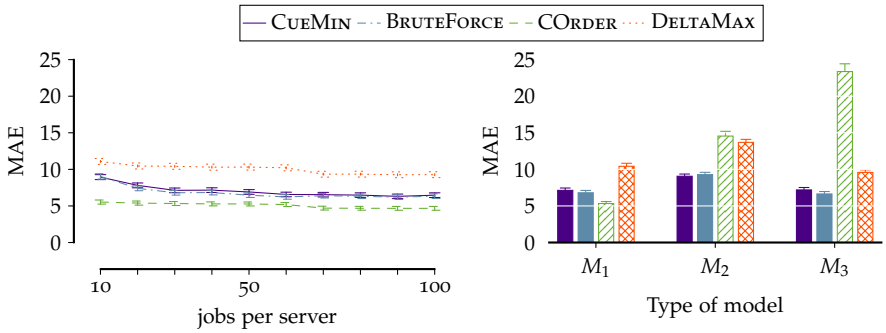


Figure 5.2: [**Number of servers estimation**] Mean absolute error (MAE) on estimating the number of servers $c \in [1, 30]$ of a $R \in \{\text{FCFS}, \text{LCFS}\}$ queue with $B = \text{F}(1)$ dependent on the number of observed jobs per server $\frac{n}{c}$ (left) and MAE dependent on the type of model (right). M_1 refers to the model type of the left plot, M_2 extends M_1 by Poisson batch sizes, and M_3 serves jobs with $R = \text{PQ}$. We show standard error in both plots.

If we add Poisson batch size distributions to the ground-truth models, both COrder and DeltaMax significantly lose accuracy as we show in the middle bar group (M_2), whereas CueMin successfully detects batch service. Service order based on priority classes, i. e., $R = \text{PQ}$, heavily violates the assumptions of COrder. We add three uniformly distributed categorical features with three categories each to the jobs and randomly select one of the features as the priority class. In this scenario (M_3), we see that COrder has an even higher increase of MAE. CueMin considers the service order in its search for the number of servers, and thus shows stable performance under varying service order and batch size.

Finally, we report accuracy and runtime on discovering the ground-truth service order for CueMin and BruteForce in Figure 5.3. We see that CueMin achieves accuracy equivalent to BruteForce, while being magnitudes faster. As expected, more jobs in the training data lead to higher service order detection accuracy. We see that the difference in runtime between CueMin and BruteForce becomes larger the more jobs per server are in the training data.

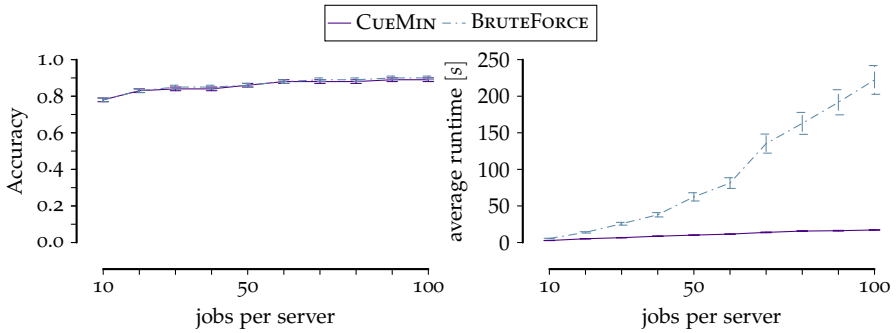


Figure 5.3: **[CueMin vs. BruteForce]** Accuracy on discovering service order R in synthetic data dependent on the number of observed jobs per server $\frac{n}{c}$ (left) and average runtime in seconds (right) for CueMin and BruteForce. We show standard error in both plots.

Data	n	m	m_{num}	$\bar{v}_{train}$	$\bar{v}_{test}$
Callcenter	21703	3	2	246s	191s
Laser	196	2	1	51h	28h
Lapping	224	2	1	79h	129h
Steel A	6969	4	3	82167	44739
Steel B	6961	4	3	1966	2232
Steel C	16458	4	3	440	635

Table 5.1: **[Real-world datasets]** Number of jobs n, number of all features m, number of numerical features m_{num} and mean sojourn time of train $\bar{v}_{train}$ and test timespan $\bar{v}_{test}$ for six different real-world datasets.

5.6.2 *Real-World Data*

Next, we show practical applicability of CueMin by evaluating on six real-world datasets of different domains, for which we give base statistics in Table 5.1. The *Callcenter* dataset [22] contains service calls along with customer priority, weekday and daytime of an Israeli bank. *Laser* and *Lapping* consist of arrival and departures times together with product category and work order quantity at two stations of a production process [86]. With relatively few jobs, they test the ability to learn from little data. Finally, *Steel A*, *Steel B* and *Steel C* correspond to three sta-

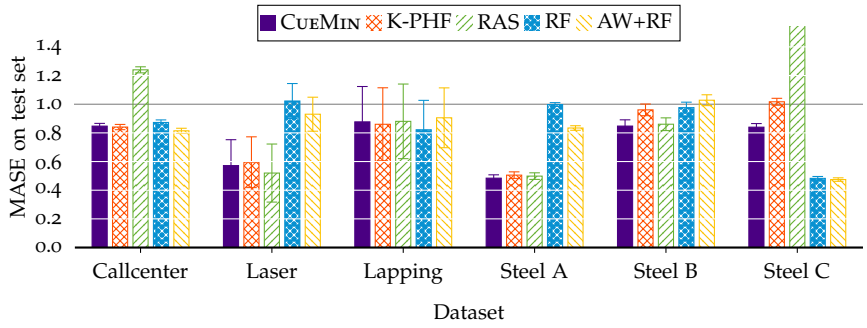


Figure 5.4: [**Sojourn time predictions**] Mean absolute scaled error (MASE, lower is better) with standard error on predicted sojourn times of six real-world datasets for CueMin, RAS, K-PHF, RF and AW+RF.

tions in the rolling mill of the German steel producer *Dillinger*. We split all datasets into a train timespan followed by a test timespan with roughly 20% of all jobs. The difference between the mean sojourn time $\bar{v}_{\text{train}}$ of the training data and the mean sojourn time $\bar{v}_{\text{test}}$ of test data indicates, that any method learning from the training data must generalize well to predict behavior of the test data.

We use CueMin to discover a queueing model on the training data, and run 1000 simulations on all arrivals to predict a distribution of sojourn times for each job. We do the same for a re-implementation of the ideas from K-PHF [138] and set hyperparameters as suggested by the authors. Furthermore, we compare to RAS [101] for which we conducted an extensive hyperparameter search and selected the best performing. As an additional baseline, we train a random forest (RF) to predict sojourn times just by the features of jobs and thus ignoring any dependence between jobs and system load. We also train a random forest (AW+RF) on the interarrival times and features in a window of k jobs to consider system load. We select hyperparameters of the random forests by grid search and cross-validation.

To compare across datasets with different timescale, we evaluate predicting sojourn time of individual jobs by the mean absolute scaled error (MASE). MASE is the mean absolute error (MAE) of the individual predictor divided by the MAE of the naive predictor that predicts the mean of the training data. We show the MASE on the test set for all methods in Figure 5.4. We see CueMin always beats the naive baseline, i. e., MASE < 1. On all datasets, it performs on par or better than

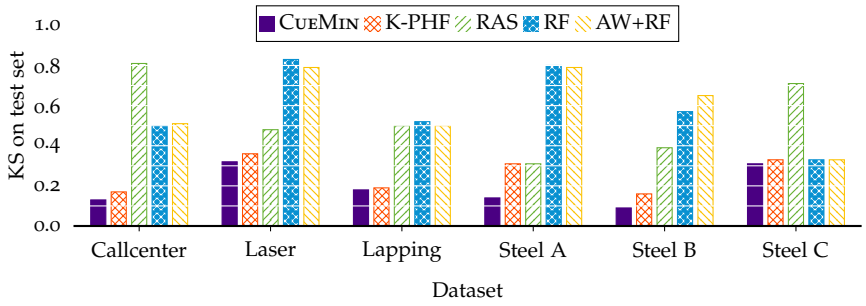


Figure 5.5: **[Distribution fitting]** KS distance (lower is better) between predicted and actual sojourn time distribution of six real-world datasets for CueMin, RAS, K-PHF, RF and AW+RF.

K-PHF, and with exception to the Steel C dataset, performs on par or better than RF and AW+RF.

During process performance analysis, domain experts are especially interested in the distribution of sojourn times and not in individual jobs. We report the Kolmogorov-Smirnov (KS) statistic between predicted and actual sojourn time distribution in Figure 5.5. We clearly see that the random forests suffer from regression to the mean. K-PHF and CueMin provide a much better approximation of the distribution than RF and AW+RF. Between the two, CueMin outperforms K-PHF especially on the Steel A and Steel B dataset.

5.6.3 *Case Study: Call Center*

We finish evaluation with a case study on the Callcenter dataset, in which we highlight the insight we can gain from the model discovered by CueMin. CueMin finds a FCFS queue with nine servers, batch size one and a load-dependent service time. Although the dataset contains an attribute for customer priority, we see CueMin favors FCFS over PQ. According to the dataset description, customers are served by their waiting time. Prioritized customers are assigned 1.5 minutes of initial waiting time. CueMin discovers that this is a neglible advantage and FCFS captures the actual behavior of the process.

We see batch size one is the correct description for calls being served one after another. Nine servers almost perfectly match the eight agents in the call center. The service time distribution is $\mathrm{NB}(1.4, 0.007)$ if six or

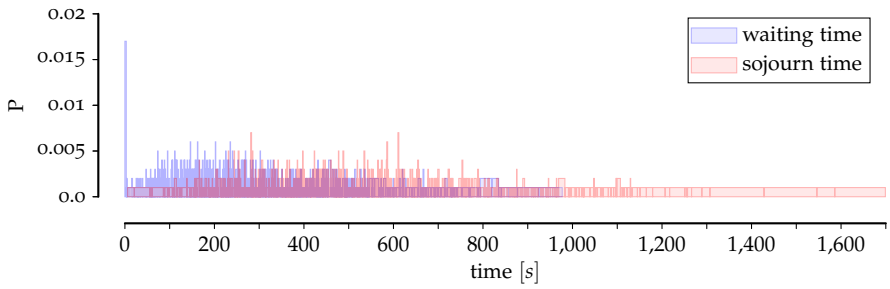


Figure 5.6: **[CueMin predicts distributions]** Distribution of predicted waiting time and sojourn time of a single job in the Callcenter dataset for 1000 simulation runs of the model discovered by CueMin.

fewer customers are in the line, and changes to NB(2.2, 0.007) if there are more calls. A former study on this dataset [22] confirms increased service time due to system load.

We predict the total call duration, i. e., sojourn time, and the waiting time of the call center's customers by conducting 1000 simulation runs of the model discovered by CueMin. We show the predicted waiting and sojourn time distribution of a single, exemplary customer in Figure 5.6. CueMin reveals the whole bandwidth of stochastic behavior of a service call. The customer has a high chance to have no waiting time at all, but if the preceding calls take more time, the waiting time for this customer increases. We see a small probability for a call duration of a few seconds due to technical issues, and we see the chance of a very long service call.

Since CueMin discovers queueing models that are inherently interpretable, domain experts can modify the model to simulate different scenarios and to find potential process optimizations. As an example, we vary the number of servers in the queueing model found by CueMin and report the predicted maximal waiting time of customers on the left of Figure 5.7. As we expected, reducing the number of servers results in an exponential growth of waiting time. If we assign costs to the usage of servers and weight them against the risk of losing customers due to high waiting times, we can run such an experiment to find the optimal number of servers.

Next, we show the impact of different service orders on the waiting time of regular and prioritized customers in the dataset on the right of Figure 5.7. We see that the model with FCFS service order results

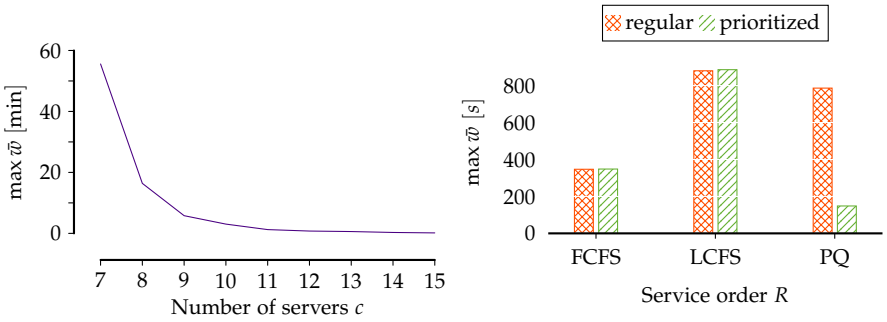


Figure 5.7: **[How to influence waiting time]** Predicted maximal waiting time $\max \bar{w}$ with varying number of servers c of the queueing model discovered by CueMin (left) and $\max \bar{w}$ with varying service order R for different classes of customers (right).

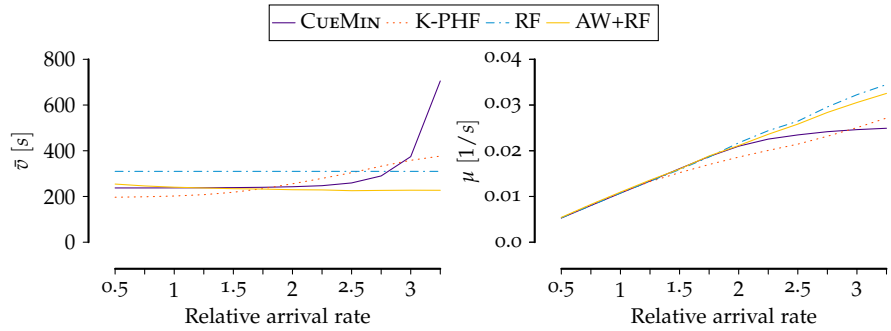


Figure 5.8: **[CueMin ably extrapolates]** Mean predicted sojourn time $\bar{v}$ (left) and predicted departure rate μ (right) dependent on down- or up-sampled arrival rate in the Callcenter dataset. We expect $\bar{v}$ to grow exponentially with system load, whereas μ should flatten when the maximal capacity of the call center is reached.

in equal and relatively low waiting times for both types of customers. LCFS order increases the maximal waiting time and does not make sense in a call center. If the call center always served prioritized before regular customers, regular customers would face a significant increase of waiting time. This explains, why in the actual process prioritized customers gain a rather small advantage in waiting time. Therefore, FCFS is a reasonable model.

Finally, we evaluate how well the models discovered by CueMin, K-PHF, RF and AW+RF extrapolate. We down- and up-sample the number of arriving jobs in the test set to simulate decreased and in-

creased system load. We show the mean predicted sojourn time $\bar{v}$ and the departure rate μ, i. e., the number of leaving jobs per second, in Figure 5.8. RF and AW+RF do not capture the expected increase of waiting time for a higher arrival rate: They predict a constant sojourn time and a linearly increasing departure rate. We see that K-PHF does slightly better: Its predicted sojourn time increases with growing arrival rate. However, since it does not model the number of servers, it misses that exceeding a certain load threshold leads to exponentially increasing waiting and thus sojourn times [54]. In contrast to all other methods, CueMin models the dependency between system load and performance that we by human intuition would expect. It predicts the expected explosion of waiting times and a limit on the departure rate, if servers are constantly overloaded. This makes CueMin a valuable tool for analyzing different scenarios in waiting line processes.

5.7 DISCUSSION

While CueMin discovered inherently interpretable models using general building blocks from queueing theory with reasonable prediction accuracy, we see interesting research directions for further improvements. First, we expect domain-specific extensions to our modeling language lead to an even better performance. For instance, we could extend our modelling language to other service orders, such as earliest-deadline-first [37], or by priority auctions [68], where customers bid for priority. Discovery of impatience [32], i. e., customers leave before served, would increase insights on service-oriented processes. Another enhancement would consider varying availability of servers over time due to vacation or machine breakdown [155].

Last but not least, we see the opportunity to apply our approach in a network of process stations [137], where each station is modeled by a queue. Regression-based remaining time prediction [113, 150] neglects dependencies between jobs and also between different process stations, and thus is not able to consider congestion in predicting sojourn times. In contrast to that, we expect our queueing model to identify congestion in these networks. We elaborate this idea in Chapter 7.

5.8 CONCLUSION

We studied discovering queueing models for interpretable waiting and sojourn time prediction from data. We formalized the problem in terms of the MDL principle, by which the best model gives the best lossless compression of the data. Due to hardness of the resulting optimization problem, we proposed the greedy CueMin algorithm to find good models in practice. Through an extensive set of experiments and a case study on call center data, we showed it ably discovers inherently interpretable models of queueing processes.

Future work includes extensions to our modeling language, such as additional service orders, customer impatience or varying availability of servers over time, as well as extending CueMin to discover a network of queueing models.

6

DISCOVERING CONSTRAINTS FOR PLANNING AND OPTIMIZATION

The final goal of any event log analysis is process optimization. While constraint programming and AI planning are powerful tools to solve assignment, optimization and scheduling problems, both methodologies require the rarely available combination of domain knowledge and mathematical modeling expertise. In this chapter, we study how learning constraints from data closes this gap.

6.1 INTRODUCTION

Constraint programming, the holy grail of programming [13], separates the concerns of modeling a problem and finding a solution. Since modeling the problem requires the rarely available combination of both domain knowledge and mathematical modeling expertise, learning constraints from data enables broader application of constraint programming [99]. Handcrafted solutions are often recorded in real-world assignment problems like scheduling and employee shift rostering, and thus provide a promising knowledge base to mine constraints. Existing approaches, however, do not satisfactorily solve this task. Active learning [15, 20, 151] requires thousands of queries even for simple problems, which is intractable if a human expert must label these queries. Passive learning approaches [76, 106, 116] need invalid examples, i. e., non-solutions, in their training set, which are usually not collected and must be generated by domain experts at great expense.

State-of-the-art methods to learn constraints purely from valid solutions either suffer from a limited constraint language, resulting in a long list of hard-to-read constraints, and need a lot of data [117], or cannot learn from real-world data because they are not noise-robust [75,

This chapter is based on [161]: Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "What are the Rules? Discovering Constraints from Data." In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI), Vancouver, Canada*. 2024.

77]. Furthermore, while learning conditions for actions in AI planning is closely related to learning constraints for constraint programming, none of the existing approaches is directly applicable to AI planning problems. Interestingly, most of AI planning specific work [7, 10, 136] considers constraint learning as only one of many problems to solve. Not focusing on constraint learning prevents these methods from being effective on this task. Despite the connection, also none of these methods is directly applicable to constraint programming problems.

To overcome all these limitations, we formalize the problem of learning constraints from exemplary solutions in terms of the Minimum Description Length (MDL) principle, by which we select the model with the best lossless compression of the data. As solving the problem exactly involves #P-hard model counting, we propose the greedy UrPiLs algorithm for Unveiling Rules from PosItive LabelS. Through extensive experiments on both constraint programming and AI planning benchmark data, we empirically show UrPiLs finds more accurate and more succinct, i. e., interpretable constraints, is more robust to noise, and has lower sample complexity than the state of the art. In summary, our main contributions are as follows. We

(a) formalize the problem of learning constraints from exemplary solutions in terms of the MDL principle,
(b) propose an efficient heuristic to discover constraints for both constraint programming and AI planning,
(c) provide an extensive empirical evaluation,
(d) make code and data publicly available.[1]

The structure of this chapter is as follows: Next, we introduce the notation of our constraint model in Section 6.2. In Section 6.3, we formalize the problem of discovering constraints in terms of MDL. Afterward, we propose our heuristic discovery algorithm UrPiLs in Section 6.4. In Section 6.5, we give an overview of related work, before we empirically evaluate UrPiLs in Section 6.6. We discuss potential future work in Section 6.7 and conclude this chapter in Section 6.8.

6.2 NOTATION FOR BOOLEAN CONSTRAINT PROGRAMMING

Before we formalize the problem, we introduce notation for boolean constraint programming we use in this chapter. Assume we are given

1 https://eda.rg.cispa.io/prj/urpils

a list of *object sets* $O_1, \ldots, O_k$ and their Cartesian product $X = \prod_{i=1}^{k} O_i$. As an example, consider the 8-Queens problem, where we want to place eight queens on a 8×8 chess board, such that no two queens attack each other. We define an object set $O_1 = \{Q_1, \ldots, Q_8\}$ for queens, and an object set $O_2 = \{S_1, \ldots, S_{64}\}$ for squares on the board. An *assignment* is a boolean function $f_a : X \to \{0,1\}$, e.g., $f_a(Q_1, S_{42}) = 1$ means queen Q_1 is on square S_{42}. We call f_a a *valid* assignment, if it satisfies a set of constraints, i. e., a *model* $M = \{C_1, \ldots, C_m\}$, with $C_i : X \to \{0,1\}$ and f_a is valid iff $\forall x \in X \; \forall C_i \in M : C_i(x) = 1$. For a given model M, we denote the set of valid assignments by $\mathcal{F}_M$.

We define constraints by a boolean algebra over the assignment f_a, a set of boolean relations between objects $\mathcal{F}_{\mathbb{B}} = \{f_1, \ldots, f_{|\mathcal{F}_{\mathbb{B}}|}\}$ with $f_i : \prod_{j \in \{1,\ldots,k\}^+} O_j \to \{0,1\}$, and arithmetic expressions over a set of numeric relations $\mathcal{F}_{\mathbb{R}} = \{f_1, \ldots, f_{|\mathcal{F}_{\mathbb{R}}|}\}$ with $f_i : \prod_{j \in \{1,\ldots,k\}^+} O_j \to \mathbb{R}$. In the 8-Queens example, we assign rows and columns to squares. Formally, we define $\mathcal{F}_{\mathbb{R}} = \{f_x, f_y\}$ with $f_x : O_2 \to \{1, \ldots, 8\}$ and $f_y : O_2 \to \{1, \ldots, 8\}$. The constraint that no more than one queen is allowed to be placed in a row can then be written as $\forall (q_1, q_2, s_1, s_2) \in O_1 \times O_1 \times O_2 \times O_2 : (s_1 \neq s_2 \wedge f_x(s_1) = f_x(s_2)) \to (f_a(q, s_1) \to \neg f_a(q, s_2))$. Our goal is to find constraints like these from a dataset of exemplary valid assignments $D = \{f_a^1, \ldots, f_a^n\}$.

6.3 MDL FOR CONSTRAINT LEARNING

From a set of exemplary assignments, we aim to discover a succinct set of constraints fitting and explaining the observed data and generalizing well to unseen data. To account for potential noise in real-world data, we need a noise-robust discovery approach. Thus, we formalize the problem of constraint discovery from exemplary solutions in terms of the MDL principle. To this end, we define length of the data encoding $L(D \mid M)$, length of the model encoding $L(M)$, and finally give a formal problem definition.

6.3.1 *Data Encoding for Constraint Programming*

To encode a dataset D, we encode all its assignments, i. e.,

$$L(D \mid M) = \sum_{f_a \in D} L(f_a \mid M) \; .$$

An empty model without constraints has $|\mathcal{F}_M| = 2^{|X|}$ valid assignments, and we need $|X|$ bits to choose one. The more constraints the model contains, the smaller the set of valid assignments, and the cheaper it is to identify the actual one. As real-world data is often noisy, there may not exist a valid assignment matching the exemplary data exactly. To ensure a lossless encoding, we have to encode the errors of the best fitting assignment. We denote the number of errors by

$$error(M \mid f_a) = \min_{f'_a \in \mathcal{F}_M} \sum_{x \in X} \mathbb{1}_{f'_a(x) \neq f_a(x)}(x) \ .$$

To encode the errors, we first specify their number by the MDL-optimal encoding for integers $L_{\mathbb{N}}$. Then, we encode the incorrect assignment values by a data-to-model code [87], i. e., an index to choose $error(M \mid f_a)$ out of $|X|$ values. In summary, we have

$$\begin{aligned} L(f_a \mid M) = \ & \log|\mathcal{F}_M| + L_{\mathbb{N}}\left(1 + error(M \mid f_a)\right) \\ & + \log \binom{|X|}{error(M \mid f_a)} \ . \end{aligned}$$

This gives us a lossless encoding of the data.

6.3.2 *Model Encoding*

Next, we compute the length of the model encoding by

$$L(M) = L_{\mathbb{N}}(|M| + 1) + \sum_{C \in M} L(C) \ ,$$

i. e., we encode the number of constraints, which can be zero, and encode each constraint. Since complex real-world problems require a rich constraint language, we first define a grammar for our constraint language, which we then use to define the encoded length of a constraint. We formally define the grammar by

$$\begin{aligned} & C \to \langle C_V \rangle \text{ “|” } \langle C_F \rangle \ : \ \langle C_T \rangle \\ & C_V \to \epsilon \mid \forall x \in X \mid \forall x, y \in X \\ & C_F \to \epsilon \mid \langle v \rangle \ = \ \langle v \rangle \mid \langle v \rangle \ \neq \ \langle v \rangle \mid \langle f_{\mathbb{B}} \rangle(\langle v \rangle) \mid \langle \textsc{nf} \rangle \mid \\ & \qquad \neg\langle C_F \rangle \mid \langle C_F \rangle \wedge \langle C_F \rangle \mid \langle C_F \rangle \vee \langle C_F \rangle \\ & v \to x_{\langle i \rangle} \mid y_{\langle i \rangle} \qquad i \to 1 \mid \ldots \mid k \qquad f_{\mathbb{B}} \to \text{one of } \mathcal{F}_{\mathbb{B}} \end{aligned}$$

$$
\begin{aligned}
&\textsc{nf} \to \langle \textsc{ne} \rangle < \langle \textsc{ne} \rangle \mid \langle \textsc{ne} \rangle \leqslant \langle \textsc{ne} \rangle \mid \langle \textsc{ne} \rangle = \langle \textsc{ne} \rangle \\
&\textsc{ne} \to \langle z \in \mathbb{R} \rangle \mid \langle f_{\mathbb{R}} \rangle (\langle v \rangle) \mid \langle \textsc{ne} \rangle \langle \odot \rangle \langle \textsc{ne} \rangle \mid \\
&\qquad \text{“}|\text{”} \langle \textsc{ne} \rangle \text{“}|\text{”} \mid \lfloor \langle \textsc{ne} \rangle \rfloor \mid \lceil \langle \textsc{ne} \rangle \rceil \\
&f_{\mathbb{R}} \to \text{one of } \mathcal{F}_{\mathbb{R}} \qquad\qquad \odot \to + \mid - \mid \cdot \mid / \\
&C_T \to f_a(x) \mid f_a(y) \mid f_a(X_{\langle j \rangle}) \mid \neg \langle C_T \rangle \mid \langle C_T \rangle \wedge \langle C_T \rangle \mid \\
&\qquad \langle C_T \rangle \vee \langle C_T \rangle \mid \langle \textsc{count} \rangle \\
&j \to 1 \mid \ldots \mid |X| \\
&\textsc{count} \to \langle \textsc{ne} \rangle \leqslant \sum_{\langle v \rangle} f_a(x) \leqslant \langle \textsc{ne} \rangle \,,
\end{aligned}
$$

where we conceptually split a constraint C into three parts, i. e., $C = (C_V, C_F, C_T)$. In C_V, we can define *variables* of object tuples in X. In C_F, we *filter* the possible values of these variables: we can test for equality and inequality of variables, we can query values of boolean and numerical relations, and we can compose complex filters with boolean operators. A numeric filter NF compares the values of two numeric expressions NE, which are any real number, any numeric relation, or a composite of arithmetic operations.

The *target* of any constraint is to define the set of valid assignments. In C_T, we restrict the valid values of an assignment f_a by a boolean expression over f_a. In its simplest form, C_T requires f_a to be true for a variable defined by C_V and C_F. We can also require f_a to be true for one specific object tuple X_j with $j \in \{1, \ldots, |X|\}$. We compose more complex constraints using boolean operators. In many real-world problems, we can distribute some kind of budget. For instance, if we assign shifts to employees during rostering, employees require a minimal and maximal workload. We model such COUNT constraints by a lower and upper bound on a sum over the assignment values of f_a.

When computing the encoded length of a constraint, we want to avoid any undue bias and therefore assume that whenever we have multiple modeling choices, all options are equally likely. Formally, we use our defined constraint grammar to recursively compute $L(C)$ by

$$
L(A) = \log |A| + \sum_{\langle \alpha \rangle \in A} L(\alpha) \,,
$$

where A is a nonterminal in the grammar, and we first encode which of the $|A|$ branches we produce, before we encode all remaining nonterminals. In the special case of $\langle z \in \mathbb{R} \rangle$, we compute the encoded length

by the MDL encoding for real numbers $L_{\mathbb{R}}(z)$. Altogether, this gives us a lossless encoding of the model.

6.3.3 *Formal Problem Definition*

Using our MDL score, we now formally state our problem.

Minimal Constraint Model Problem *Given a dataset D of assignments $f_a^1, \ldots, f_a^n$, find the constraint model M minimizing the total encoded cost $L(D, M) = L(D \mid M) + L(M)$.*

Solving this problem optimally is intractable in practice. Potentially, we have up to $2^{|X|}$ valid assignments, i. e., we face an exponentially growing search space for constraints. Moreover, our MDL score does not exhibit properties such as monotonicity or submodularity that we can exploit to efficiently find an optimal solution. We prove this statement in Section 6.3.4 by counterexamples.

Additionally, even computing $L(D \mid M)$ is hard by itself. Finding a valid assignment f_a' for M that is nearest to a given assignment f_a corresponds to finding a valid assignment having maximal Manhattan distance to f_a with negated values, which in general is NP-hard [30].

Computing the number of valid assignments $|\mathcal{F}_M|$ is equivalent to counting the solutions of a boolean formula, which is #P-complete, i. e., at least as hard as NP-complete [152]. Researchers have proposed algorithms like Ganak [140], SharpSat-TD [72] or ApproxMC [142] to tackle the problem. Dependent on the complexity of the formula, these approaches take several seconds, minutes or even hours [43], which is too slow for evaluating many constraint candidates during search. Hence, we resort to heuristics.

6.3.4 *$L(D, M)$ is neither monotone nor submodular*

Here, we show by counterexamples that our MDL score does not exhibit monotonicity or submodularity, which we could exploit to efficiently solve the minimal constraint model problem. To this end,

we define a dataset with $O_1 = \{o_1^1, \ldots, o_{10}^1\}$, $O_2 = \{o_1^2, \ldots, o_{10}^2\}$ and $D = \{f_a \mid \sum_{x \in X} f_a(x) = 1\}$. We further define the constraints

$$C_1 = 1 \leqslant \sum f_a(\cdot) \leqslant 1$$
$$C_2 = 0 \leqslant \sum f_a(\cdot) \leqslant 1$$
$$C_3 = 1 \leqslant \sum f_a(\cdot) \leqslant 2 \,.$$

The encoding of these constraints differs in the lower and upper bound only. We compute the encoded length of each constraint as defined in Section 6.3.2. In our example, we have

$$L(C_1) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(2) + 2 + L_{\mathbb{N}}(2)$$
$$L(C_2) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(1) + 2 + L_{\mathbb{N}}(2)$$
$$L(C_3) = \log 3 + \log 8 + \log 7 + L_{\mathbb{N}}(2) + 2 + L_{\mathbb{N}}(3) \,.$$

Next, we show that $L(D, M)$ is not a monotone function. A function f is called monotone if

$$\forall T \subseteq S : f(T) \leqslant f(S) \,.$$

We compute

$$L(D, \varnothing) = L_{\mathbb{N}}(1) + 100 \cdot (100 + L_{\mathbb{N}}(1)) \approx 10153$$
$$L(D, \{C_1\}) = L_{\mathbb{N}}(2) + L(C_1) + 100 \cdot (\log 100 + L_{\mathbb{N}}(1)) \approx 833$$
$$L(D, \{C_1, C_2\}) = L_{\mathbb{N}}(3) + L(C_1) + L(C_2) + 100 \cdot (\log 100 + L_{\mathbb{N}}(1)) \approx 847 \,,$$

where we first encode the number of constraints, then each constraint, and for each of the 100 examples in the dataset, we compute the number of bits to select a valid assignment, and $L_{\mathbb{N}}(1)$ encodes the number of errors is zero. By this, we see $L(D, M)$ is not monotone.

Next, we show that $L(D, M)$ is not submodular. A function f is called submodular if

$$\forall X \subseteq \Omega : \forall x_1, x_2 \in \Omega \backslash X : \\ f(x \cup \{x_1\}) + f(x \cup \{x_2\}) \geqslant f(X \cup \{x_1, x_2\}) + f(X) \,.$$

We compute

$$L(D, \{C_1, C_3\}) + L(D, \{C_2, C_3\}) \approx 1698$$
$$L(D, \{C_1, C_2, C_3\}) + L(D, \{C_3\}) \approx 2265 \,,$$

and

$$L(D, \{C_1, C_2\}) + L(D, \{C_1, C_3\}) \approx 1697.1$$
$$L(D, \{C_1, C_2, C_3\}) + L(D, \{C_1\}) \approx 1696.6\,,$$

by which we see that $L(D, M)$ is not submodular.

6.4 THE URPILS ALGORITHM

Since solving the minimal constraint model problem optimally is intractable, we resort to greedy solutions.

6.4.1 *Estimating the Number of Valid Assignments*

To compute the encoded length of an assignment, $L(f_a \mid M)$, we must count the number of valid assignments $|\mathcal{F}_M|$ for a given model M. We use an approximation, which is fast to compute and still enables useful comparison of constraint candidates. We estimate the number of valid assignments based on a standard algorithm for exact counting [166]. First, we transform our constraint model M into a boolean function of conjunctive normal form (CNF), where each possible parameter combination of f_a corresponds to a boolean variable. Next, we compute the *constraint graph* G of the formula, which is an undirected graph with variables as nodes, and two variables are connected if they occur together in a clause. We count the number of valid assignments separately for disconnected, i.e., independent, components and get the total count by multiplying the result of each component.

If the graph is small enough and contains less than five variables, it is feasible to count the number of valid assignments exactly by enumeration. Otherwise, we use a polynomial-time approximation. If clauses in the CNF contain at most two variables, the number of valid assignments corresponds to the number of independent sets in G [31], where an independent set is any set of non-adjacent nodes. We compute a lower bound for $|\mathcal{F}_M|$ by [130]

$$|\mathcal{F}_M| \geqslant \prod_{v \in V} (\deg v + 2)^{1/(\deg v + 1)},$$

with V being the set of nodes in G, and $\deg v$ denotes the degree of node v. The number of variables per clause only depends on the target

part C_T of a constraint. If C_T has the form $f_a(\cdot)$ or $\neg f_a(\cdot)$, we have one variable per clause, whereas implications like $f_a(x) \rightarrow f_a(y)$ result in two variables per clause. In these cases, our lower bound leads to valid results. As we will show later in the experiments, these unary and binary relationships between variables are sufficient to describe most of the constraints in many problems.

COUNT constraints, however, in general lead to clauses with more than two variables. For instance, let $x_1, \ldots, x_6$ be boolean variables and consider the constraint $\sum_{i=1}^{6} x_i = 3$. Then, the CNF of this constraint is $\prod_{i=1}^{4} \prod_{j=i}^{5} \prod_{k=j}^{6} (x_i + x_j + x_k)$, i.e., we have three variables per clause. Therefore, we need to compute $|\mathcal{F}_M|$ differently for COUNT constraints. For a single equality constraint $\sum_{i=1}^{n} x_i = a$, we have $\binom{n}{a}$ satisfying assignments. We generalize this to inequality constraints $a \leqslant \sum_{i=1}^{n} x_i \leqslant b$ which have $\sum_{i=a}^{b} \binom{n}{i}$ satisfying assignments.

Since we do not know how the intersection of the valid assignments for multiple COUNT constraints looks like, because enumerating them is intractable, we can only make assumptions. We assume all COUNT constraints equally contribute to the final count, and thus divide the mean of the individual counts by the number of constraints, i.e.,

$$|\mathcal{F}_M| = \left\lceil \frac{\sum_{C \in M} |\mathcal{F}_{\{C\}}|}{|M|^2} \right\rceil .$$

By this, we can quickly estimate the number of valid assignments.

6.4.2 *Estimating the Best Fitting Valid Assignment*

To compute $L(f_a \mid M)$, we also need to compute $error(M \mid f_a)$, i.e., the minimal number of values we need to change in a valid assignment of M to get f_a. Since we must repeat this computation many times during our search for constraints, we want this to be as fast as possible. As in counting the number of valid assignments, enumerating all assignments to find $error(M \mid f_a)$ is intractable.

In contrast to $error(M \mid f_a)$, the number of unsatisfied clauses in the CNF formula of the model is cheap to compute. The more clauses are unsatisfied, the more variables we expect must be flipped to satisfy the formula, and hence the higher is $error(M \mid f_a)$. We estimate the number of variables we must flip to satisfy the formula by using the coupon collector's problem [114][42, p. 225]: If we assume that for each of the

Algorithm 13: UrPiLs

input : dataset D
output: set of constraints M

$M \leftarrow \emptyset$;
$M \leftarrow \text{Filter}(M, D, \text{GenerateSimpleCandidates}(D))$;
$M \leftarrow \text{Filter}(M, D, \text{GenerateComplexCandidates}(M, D))$;
$M \leftarrow \text{Filter}(M, D, \text{GenerateCountCandidates}(D))$;
return M;

m unsatisfied clauses, we draw one of $|V|$ variables with replacement to flip, the expected number of flipped variables is

$$error(M \mid f_a) = \left\lceil |V| - |V| \cdot (1 - \frac{1}{|V|})^m \right\rceil .$$

The value of $error(M \mid f_a)$ is 0 if no clause is unsatisfied, increases with m and does not exceed the number of variables $|V|$. By this, we can compute $L(D, M)$ for model selection.

6.4.3 *Discovering a Good Constraint Model*

We now want to minimize $L(D, M)$ for a given dataset D, i.e., we want to discover a good constraint model in feasible time. Since computing $L(D \mid M)$ is harder for models with COUNT constraints, we search for these at the end. Many satisfiability and optimization problems contain a set of relatively simple constraints, even if they also contain a set of very complex constraints. Simple constraints typically involve none or only one feature relation in the filtering part C_F. Therefore, we propose our method UrPiLs, in which we split the search for constraints into three stages.

We give the pseudocode of UrPiLs as Algorithm 13. Starting with an empty model, we first search for the low-hanging fruit and generate simple constraint candidates. In constraint programming, we are often interested in modeling the pairwise relationship between variables. For example, we require in Sudoku that two cells in the same row do not have the same value. Hence, we generate a set of simple candidates with all constraints of the form $\forall x, y \in X \mid C_F : f_a$. In C_F, we compare the values of at most one boolean and numerical relation, e.g. $f(x_1) <$

Algorithm 14: GENERATESIMPLECANDIDATES

input : dataset D with boolean $\mathcal{F}_{\mathbb{B}}$ and numerical functions $\mathcal{F}_{\mathbb{R}}$
output: set of candidates Q

1 $\mathcal{C}_F^1 \leftarrow \{x \neq y\} \cup \bigcup_{i=1}^{k} \{x_i = y_i \wedge \forall_{j \neq i} x_j \neq y_j\}$;
2 $\mathcal{C}_F^2 \leftarrow \{\epsilon\}$;
3 **foreach** $f \in \mathcal{F}_{\mathbb{B}}$ **do**
4 **foreach** $i \in \{1, \ldots, k\}$ **do**
5 **if** $\operatorname{dom} f = O_i$ **then**
6 $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i), \neg f(x_i)\}$;
7 **foreach** $j \in \{1, \ldots, k\}$ **do**
8 **if** $\operatorname{dom} f = O_j$ **then**
9 $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i) \wedge f(y_i)\}$;
10 **foreach** $f \in \mathcal{F}_{\mathbb{R}}$ **do**
11 **foreach** $i, j \in \{1, \ldots, k\}$ **do**
12 **if** $\operatorname{dom} f = O_i = O_j$ **then**
13 **foreach** $\odot \in \{<, \leqslant, =, >, \geqslant\}$ **do**
14 $\mathcal{C}_F^2 \leftarrow \mathcal{C}_F^2 \cup \{f(x_i) \odot f(y_i)\}$;
15 $\mathcal{C}_T \leftarrow \{f_a(x) \rightarrow f_a(y), f_a(x) \rightarrow \neg f_a(y)\}$;
16 $Q \leftarrow \varnothing$;
17 **foreach** $C_F^1, C_F^2, C_T \in \mathcal{C}_F^1 \times \mathcal{C}_F^2 \times \mathcal{C}_T$ **do**
18 add $\left(\forall x, y \in X \mid C_F^1 \wedge C_F^2 : C_T\right)$ to Q;
19 **return** Q;

$f(y_1)$ with $f \in \mathcal{F}_{\mathbb{R}}$. To restrict the pairwise assignment values of x and y, we generate implications of the type $f_a(x) \rightarrow f_a(y)$ and $f_a(x) \rightarrow \neg f_a(y)$ for C_T.

For further reference, we give the pseudocode of GENERATESIMPLECANDIDATES as Algorithm 14. The main part of the algorithm deals with generating candidates for C_F. Every constraint requires that the variables x and y in $C_V = \forall x, y \in X$ differ at all indices or at precisely one index (ln. 1). The candidates for the second part of C_F include an empty constraint (ln. 2). We further compare values of boolean relations with domains compatible to the input variables (ln. 3–9), and we do the same for numerical relations (ln. 10–14). For the target part, we

Algorithm 15: FILTER

input : current model M, dataset D, set of candidates Q
output: extended model M

```
1  sort Q by L(D, ·);
2  foreach C′ ∈ Q do
3      foreach C ∈ M do
4          if C′_V = C_V ∧ C′_T = C_T then
5              C′_F ← C′_F ∨ C_F;
6              M′ ← M\{C};
7              break;
8      M′ ← M′ ∪ {C′};
9      if L(D, M′) < L(D, M) then
10         M ← M′;
11 return M;
```

create implications of assignment values (ln. 15). We generate simple constraint candidates from the cross product of the candidates for the individual constraint parts (ln. 17–18).

We filter the generated candidates using our FILTER subroutine, for which we give the pseudocode as Algorithm 15. We test the most promising candidates first through sorting candidates by their individual gain. To minimize model complexity, we try to merge each candidate C' with an existing constraint $C \in M$. We can merge constraints if they share the same variable and target part. For example, we merge

$$\forall x, y \in X \mid g(x_1) < g(y_1) : f_a(x) \to \neg f_a(y)$$

$$\text{and } \forall x, y \in X \mid h(x_2) = h(y_2) : f_a(x) \to \neg f_a(y)$$

$$\text{into } \forall x, y \in X \mid g(x_1) < g(y_1) \vee h(x_2) = h(y_2) : f_a(x) \to \neg f_a(y)\,.$$

If a candidate improves our score, we add it to the model.

A model with simple constraints gives us a good baseline from which we search for constraints with a more complex filtering part. We give the pseudocode for GENERATECOMPLEXCANDIDATES as Algorithm 16. We first initialize the filtering part, C_F^1, and the target part, C_T, as in GENERATESIMPLECANDIDATES (ln. 1–2). For the second filtering part, we then create a set of basic filter elements B by producing

Algorithm 16: GENERATECOMPLEXCANDIDATES

input : model with simple constraints M, dataset D, grammar production depth d
output: set of candidates Q

1 $\mathcal{C}_F^1 \leftarrow \{x \neq y\} \cup \bigcup_{i=1}^{k} \{x_i = y_i \wedge \forall_{j \neq i} x_j \neq y_j\}$;
2 $\mathcal{C}_T \leftarrow \{f_a(x) \rightarrow f_a(y), f_a(x) \rightarrow \neg f_a(y)\}$;
3 $B \leftarrow \left\{ u \mid u \cap \{\wedge, \vee\} = \varnothing \wedge \langle C_F \rangle \overset{d}{\Rightarrow} u \right\}$;
4 $Q \leftarrow \varnothing$;
5 **foreach** $C_F^1, C_T \in \mathcal{C}_F^1 \times \mathcal{C}_T$ **do**
6 $\quad C_F^2 \leftarrow \arg\min_{C_F^2 \subset B} L(C_F^2 \mid C_F^1, C_T, D, M)$;
7 $\quad$ add $\left(\forall x, y \in X \mid C_F^1 \wedge C_F^2 : C_T\right)$ to Q;
8 **return** Q;

all possible expressions from our grammar for C_F up to a user-defined depth d (ln. 3). In our experiments, we set $d = 3$. Later in the search, we combine basic filter elements to more complex expressions, and hence only create basic elements without $(\wedge)$ or $(\vee)$. Instead of an intractable search over all possible filtering expressions, we map the problem to a simpler binary classification problem. We test for each pair $x, y \in X$ whether $C_T(x, y)$ improves the fit on the data, i.e., it leads to a lower $L(D \mid M)$. Finally, we look for a conjunction of elements from B best explaining the division into positive and negative pairs (x, y), which gives us a good candidate for C_F.

To find this conjunction C_F^2, we define a two-part MDL score to evaluate candidates. Formally, we find the C_F^2, which minimizes

$$
\begin{aligned}
L(C_F^2 \mid C_F^1, C_T, D, M) = {} & L_{\mathbb{N}}(\left|C_F^2\right|) + \log \binom{|B|}{\left|C_F^2\right|} \\
& + L_{\mathbb{N}}(1 + error(C_F^2 \mid C_F^1, C_T, D, M)) \\
& + \log \binom{\left|\{x, y \mid C_F^1(x, y)\}\right|}{error(C_F^2 \mid C_F^1, C_T, D, M)},
\end{aligned}
$$

where we compute the length of the model encoding by the number of elements in the conjunction and an index to select the elements of

C_F^2 from B. Then, we encode the number and identity of pairs $x, y \in X$, for which $C_F^2(x, y)$ makes a classification error. We formally define

$$error(C_F^2 \mid C_F^1, C_T, D, M) = \sum_{x,y \in X} C_F^1(x, y) \left| \mathbb{1}_{\Delta L(D \mid M \cup \{C_T\}) < 0}(x, y) - C_F^2(x, y) \right|,$$

with $\Delta L\,(D \mid M \cup \{C_T\})$ being the gain in the encoded length of the data, after adding C_T to the model. This gives us candidates with complex filtering expressions.

In the last stage of UrPiLs, we search for count constraints. To this end, we create candidates for different input partitions of f_a similar to the existing CountOR algorithm [77]. Formally, we create constraints of the form $\forall x \in X \mid C_F : a \leqslant \sum f_a(x) \leqslant b$, where we generate candidates for $a, b \in \mathbb{N}$ by observations in D. We generate an empty C_F, and we generate all possible $C_F = f(x_i)$ with $f \in \mathcal{F}_{\mathbb{B}}$ and $i \in \{1, \ldots, k\}$. Again, we use Filter to select which candidates we add to our final model. This gives us a set of constraints from exemplary assignments.

6.4.4 UrPiLs *for AI Planning*

Next, we adapt UrPiLs for AI planning problems, where actions modify the state of an environment until we reach a predefined goal state. We reuse notation and define a state by boolean and numerical relations between objects from different object sets. We write f_i^j to denote relation f_i at state j. W.l.o.g we consider a single action a. We denote the assignment at state j by f_a^j, and $f_a^j(x) = 1$ if a is executed with objects x at state j and $f_a^j(x) = 0$ else. As before, we aim to find constraints M for valid assignments and thus preconditions to execute a.

As all valid assignments satisfy $\sum_{x \in X} f_a(x) = 1$, the empty model has $|X|$ instead of $2^{|X|}$ valid assignments. To encode errors efficiently, we specify for each assignment in the data if it is valid for M or not. Since we do not know the number of valid and invalid assignments beforehand, we use prequential codes (see Section 2.3). If an assign-

Algorithm 17: GENERATEPLANNINGCANDIDATES

input : dataset D
output: set of candidates Q

$Q \leftarrow \varnothing$;
foreach $u \in \bigcup_{i=1}^{k} \{1, \ldots, k\}^i$ **do**
  foreach $f \in \mathcal{F}_{\mathbb{B}}$ **do**
    if $\operatorname{dom} f = \prod_{i \in u} O_i$ **then**
      add $(\forall x \in X \mid f(x[u]) : \neg f_a(x))$ to Q;
      add $(\forall x \in X \mid \neg f(x[u]) : \neg f_a(x))$ to Q;
foreach $i, j \in \{1, \ldots, k\}$ **do**
  foreach $f \in \mathcal{F}_{\mathbb{R}}$ **do**
    if $i \neq j \wedge O_i = O_j = \operatorname{dom} f$ **then**
      foreach $\odot \in \{<, \leqslant, =, >, \geqslant\}$ **do**
        $C_F \leftarrow f(x_i) \odot f(x_j)$;
        add $(\forall x \in X \mid C_F : \neg f_a(x))$ to Q;
return Q;

ment is valid, we encode it via an index over all valid assignments, otherwise via an index over all other assignments. Formally, we have

$$L_{\mathrm{AI}}(D \mid M) = \sum_{i=1}^{|D|} -\log \binom{\operatorname{usg}_i f_a^i \in \mathcal{F}_M + \epsilon}{\operatorname{usg}_i 0 + \operatorname{usg}_i 1 + 2\epsilon} + \begin{cases} \log |\mathcal{F}_M|, & \text{if } f_a^i \in \mathcal{F}_M \\ \log(|X| - |\mathcal{F}_M|), & \text{otherwise,} \end{cases}$$

where $\operatorname{usg}_i x$ is how often code x has been used up to the i-th assignment, and ϵ with standard choice 0.5 is for smoothing. This gives us an efficient encoding for AI planning data.

Since $\sum_{x \in X} f_a(x) = 1$ for all valid assignments, we neither need constraints on pairwise relationships of f_a nor count constraints. Instead, we search for constraints telling us when we are not allowed to execute an action. This means we create candidates $\forall x \in X \mid C_F : \neg f_a(x)$, where C_F compares boolean and numerical relations. We give the pseudocode of GENERATEPLANNINGCANDIDATES as Algorithm 17. We start

by generating candidates with boolean relations (ln. 2-6). To generate syntactically valid constraints, we ensure that the parameters of the relation are a subset of the parameters of the assignment function f_a. We generate comparisons of numerical relation using different comparison operators and also ensure that the candidates are syntactically valid constraints (ln. 7-12).

By this we can apply UrPiLs to AI planning data.

6.5 RELATED WORK

Learning constraints for constraint programming is a widely studied research problem. Active learning approaches [15, 20, 151] derive constraints by asking queries in the form of partial or complete solutions and non-solutions. Even for simple problems, these approaches, however, may require thousands of queries, which limits their applicability if a human expert must label these queries. Therefore, researchers proposed to learn constraints from a static set of both solutions and non-solutions [76, 106, 116]. While handcrafted solutions are usually recorded in real-world applications like scheduling and staff rostering, non-solutions representing forbidden behavior often are not collected. Hence, data and label acquisition as a bottleneck still can prevent application of such methods.

Recent work finds constraints from solutions only. This often results in methods being specialized in narrow contexts, such as integer linear programming [96], scheduling sequences [112] or tabular spreadsheets [71]. CountOR [77] infers count constraints. It, however, cannot handle noise, can only create simple expressions, and does not consider redundancy between constraints. CountCP [75] extends CountOR by a richer modeling language and reduced redundancy, but still does not handle noise. MineAcq [117] selects constraints by permutation testing. In contrast to CountOR and CountCP, MineAcq does not find quantified constraints, which can lead to a large result set. CABSC [28] also selects constraints by counting valid assignments, but needs user-provided knowledge of constraints and does not handle noise.

In AI planning, many approaches try to infer domain models including preconditions (i.e. constraints) for executing actions from exemplary execution plans [10]. Strictly assuming no noise, FAMA [7] formalizes the problem as a planning problem itself. PlanMiner [136]

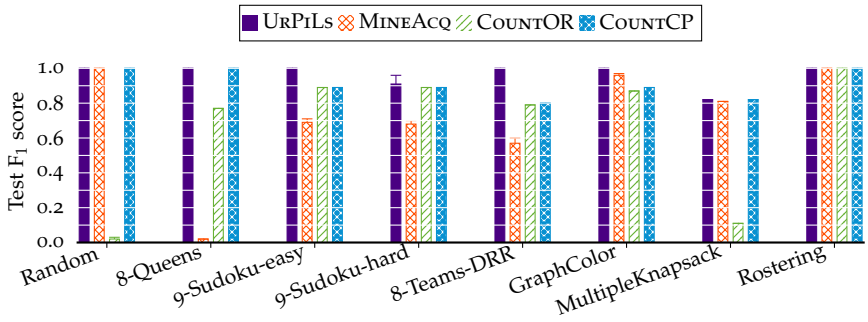


Figure 6.1: **[UrPiLs finds high-quality constraints]** Average F_1 score (higher is better) on the test set for ten independent runs on 1000 randomly drawn training examples, for constraints discovered by UrPiLs, MineAcq, CountOR and CountCP. Error bars show standard deviation.

translates the problem to rule-based classification, and PlanMiner-N [135] improves PlanMiner's noise-handling. AI planning domain acquisition methods tackle multiple tasks, treating the learning of action constraints as an unfocused subproblem. In contrast to all other methods above, UrPiLs discovers a succinct set of constraints with low sample complexity, is robust to noise, and can be applied to a broad domain of optimization, satisfiability and planning problems.

6.6 EXPERIMENTS

Now, we evaluate UrPiLs on constraint programming and AI planning datasets. Since both domains have specialized state-of-the-art methods that are not applicable to both problems, we split the experiments into two. We conduct all our experiments on a PC with Windows 10, an Intel i7-6700 CPU and 32 GB of memory. To ensure reproducibility, we make code and data publicly available in the extra materials.

6.6.1 *Experiments on Constraint Programming Datasets*

We start by comparing UrPiLs with the state of the art from related work. While CountOR and CountCP have no hyperparameters, we must generate candidate constraints for MineAcq and set parameters τ and ρ to control the acceptance threshold of its permutation test for

candidate selection. To ensure MINEACQ can find all necessary constraints to model the datasets without providing too much knowledge about the ground-truth constraints, we generate all pairwise implications $f_a(x) \rightarrow f_a(y)$ and $f_a(x) \rightarrow \neg f_a(y)$. By a manual hyperparameter search, we find $\tau = 10$ and $\rho = 0.001$ lead to the best results.

DATASETS We experiment on datasets with different characteristics. To test if the constraint learners find spurious results, we create a synthetic dataset *Random*, where we uniformly and randomly sample values for f_a. We also uniformly and randomly sample values for boolean and numerical relations in the dataset, i. e., there is no dependency to f_a, and the ground truth is an empty model without any constraints. Besides, we evaluate on datasets with non-empty ground-truth constraints. *8-Queens* contains examples for positioning eight queens on a chessboard such that no two queens attack each other. Since modelers may include knowledge about the problem into the modeled relations, we create two versions of a 9×9 Sudoku dataset. In *9-Sudoku-easy*, we specify for each cell its row, column and block number. In *9-Sudoku-hard*, we only specify row and column.

For *8-Teams-DRR*, we generate data of eight teams in a double round-robin competition, i. e., $f_a(x, y, z) = 1$ if on match day x team y plays against team z, each team plays twice against each other on 14 match days, and we require symmetry between first and second half of the matches. In *GraphColor*, we generate a random undirected graph with ten nodes and twenty edges, and valid assignments are node colorings where neighbors have different colors. For *MultipleKnapsack*, we assign twenty items of different weight and value to three knapsacks of limited size. Our last dataset, *Rostering*, is an instance of a nurse rostering problem[2], where boolean relations refer to shift types and numerical relations model the start times, end times and durations of shifts.

QUALITY OF DISCOVERED CONSTRAINTS To see how well the discovered constraints match the ground-truth, we generate valid assignments for all datasets and split them into training and test set. For the test set, we additionally generate examples violating the ground-truth constraints. First, we run all methods on the training data. Then, we classify test examples *positive* if they satisfy all found constraints and

2 http://www.schedulingbenchmarks.org/nrp/

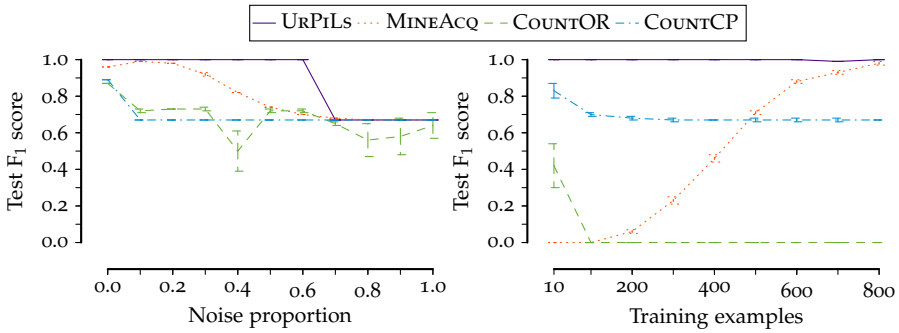


Figure 6.2: [**UrPiLs is noise-robust with low sample complexity**] Average F_1 score on the GraphColor test set for ten independent runs, for constraints discovered by UrPiLs, MineAcq and CountOR, CountCP dependent on the proportion of noisy, i. e., invalid, assignments in the training set (left, higher is better). Average F_1 score on the 5-Queens test set with 10% noise on a varying number of training examples (right, higher is better). Error bars show standard error.

negative otherwise. We report the F_1 score with 1000 training examples in Figure 6.1. We see that UrPiLs in contrast to its competitors achieves almost perfect F_1 score on all datasets. On 9-Sudoku-hard, UrPiLs does not find the block constraint in all runs, but on average still performs best. On MultipleKnapsack, none of the methods recovers the ground-truth constraints.

NOISE ROBUSTNESS To evaluate noise-robustness, we inject noise into the training data by adding invalid assignments. We report test F_1 score on GraphColor dependent on the noise proportion in Figure 6.2 (left). We see UrPiLs recovers the ground-truth for up to 60% noise and is on par for higher noise levels. The F_1 score of CountOR and CountCP drops for significantly less noise to $\frac{2}{3}$, i.e., a model that accepts all test examples with recall 1 and precision 0.5.

We also test noise-robustness on the queens problem. MineAcq has much higher F_1 score with same training set size for lower dimensional problems. Furthermore, runtime of all methods is lower for smaller problems. To enable many runs, we evaluate on 5-Queens, reducing the problem to five queens on a 5×5 chessboard. We report F_1 score on the test set with 10% noise on the training set dependent on the num-

	UrPiLs		MineAcq		CountOR		CountCP	
Dataset	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]	$\|\|M\|\|$	t [s]
Random	0	87	0	1	136	1	51	473
8-Queens	52	1543	10^5	8	46	2	90	12350
4-Sudoku-easy	40	8	1280	1	87	1	58	192
4-Sudoku-hard	45	9	1280	1	66	1	48	195
9-Sudoku-easy	40	8107	50674	13	87	3	58	24931
9-Sudoku-hard	40	3735	50774	12	66	2	48	25217
8-Teams-DRR	90	950	10^6	19	72	1	44	38425
GraphColor	33	14	30162	2	18	1	28	471
Rostering	78	5930	10^6	46	81	14	83	87459

Table 6.1: **[Model size and runtime on constraint programming datasets]** Number of constraint terms $||M||$ and average discovery runtime t in seconds over ten independent runs for UrPiLs, MineAcq, CountOR and CountCP on different constraint programming datasets.

ber of training examples in Figure 6.2 (right). We see CountCP and especially CountOR pick up noise and discover bad generalizing constraints. MineAcq performs better, but needs 800 examples to achieve 100% F_1 score. UrPiLs is not only robust to noise. It also achieves 100% F_1 score with ten training examples.

MODEL SIZE To evaluate model size, we count the total number of constraint literals $||M||$ as defined by our constraint grammar in the models discovered by UrPiLs, MineAcq, CountOR and CountCP. We report results in Table 6.1. Across all datasets UrPiLs finds a compact set of constraints with a total of only 33 to 90 literals. CountOR and CountCP show similar results, but tend to need more constraint terms for equal F_1 score. Since MineAcq does not look for quantified expressions, it produces constraint sets with 1280 literals for 4x4-Sudoku and 10^6 literals on the 8-Teams-DRR dataset.

RUNTIME We report wall-clock running times for single-threaded execution of UrPiLs, MineAcq, CountOR, and CountCP on all datasets in Table 6.1. We see that UrPiLs is not the fastest but still shows reasonable runtime on all datasets. In particular, it is significantly faster

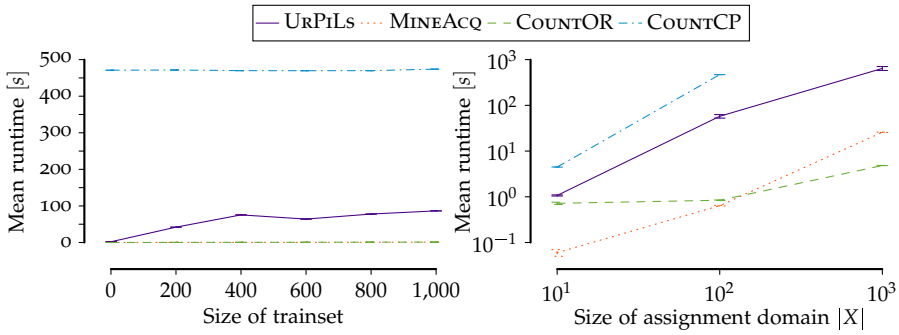


Figure 6.3: **[Runtime on Random dataset]** Mean runtime in seconds for UrPiLs, MineAcq, CountOR and CountCP on the Random dataset for $|X| = 100$ and varying number of training examples (left), and mean runtime in seconds for all methods on the Random dataset for 1000 examples and varying $|X|$ (right). CountCP did not finish within 12 hours for $|X| = 10^3$.

than CountCP. CountOR and MineAcq are magnitudes faster than UrPiLs, but CountOR cannot deal with noise in the data and MineAcq produces models with significantly more constraint terms and needs more examples to discover constraints.

To evaluate runtime scaling behavior of all methods, we use the Random dataset and vary the number of training examples and the size of the assignment function domain $|X|$ as a proxy for problem size in Figure 6.3. We see that UrPiLs shows a linear runtime behavior in the number of training examples, while the other methods work on a compressed representation of a dataset and thus remain relatively constant in their runtime. Furthermore, we see that the size of the assignment function domain has a high impact on the runtime of all methods. Since $|X| = \prod_{i=1}^{k} |O_i|$, adding a few objects to the problem can lead to a significant growth in runtime. Furthermore, the size of the space of possible assignments increases significantly, which makes it harder to discover the right constraints, and we need more examples for constraint discovery. We observe all these effects when comparing 4×4 and 9×9 Sudoku in Table 6.1. Therefore, future work should examine how to reduce the size of a given problem.

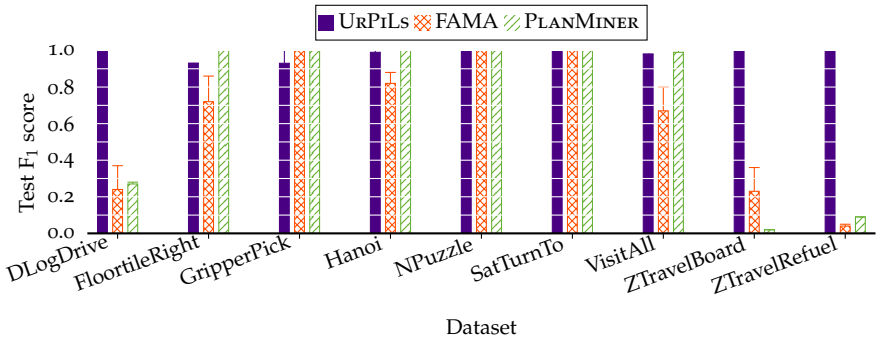


Figure 6.4: **[AI Planning Results]** Average F_1 score with standard error on the test sets of AI planning datasets for ten independent runs for UrPiLs, FAMA, and PlanMiner.

6.6.2 *Experiments on AI Planning Datasets*

Finally, we evaluate UrPiLs on AI planning benchmark datasets [7] and compare to the state-of-the-art methods FAMA and PlanMiner from related work. Unfortunately, the authors of PlanMiner-N have not published code for their method and did not respond to our emails. As before, we generate a test set for each dataset with valid and invalid executions of an action in the corresponding planning domain. We report the classification F_1 score for each method in Figure 6.4. We see that UrPiLs beats the state of the art by a wide margin.

In our last experiment, we evaluate noise-robustness on the Hanoi dataset. We report F_1 score and the number of relations in the discovered constraints for varying noise proportion in Figure 6.5. If the data contains noise, FAMA does not find any constraints. PlanMiner seems to pick up noise and finds constraints with a poor F_1 score on the test set. In contrast to that, UrPiLs is very robust to sensible amounts of noise. If noise level increases, UrPiLs finds fewer constraints, i. e., it does not find spurious constraints.

6.7 DISCUSSION

In our experiments, we empirically show UrPiLs not only finds more accurate constraints, but also finds more succinct constraints, is more robust to noise, and has lower sample complexity than the start of the

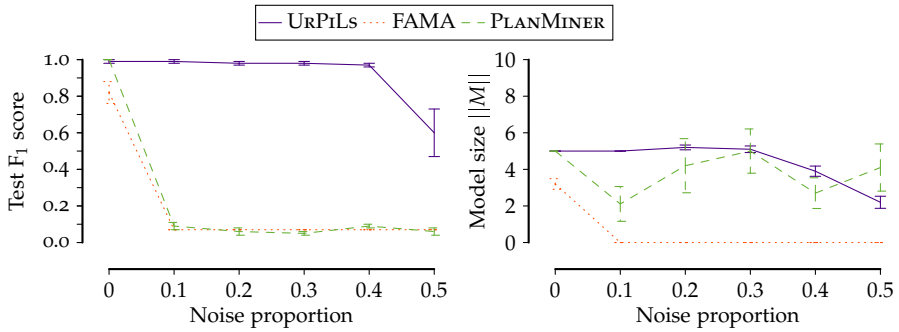


Figure 6.5: **[UrPiLs is noise-robust on AI planning data]** Average F_1 score on the Hanoi test set over ten independent runs for constraints discovered by UrPiLs, FAMA and PlanMiner under varying noise proportion on the training data (left, higher is better), and discovered model size $||M||$ under increasing noise (right). Error bars show standard error.

art. Nonetheless, UrPiLs has its limitations, and we see interesting research directions to overcome them. First, although we use a rich modeling language, we cannot model everything. As we see in Figure 6.1, UrPiLs does not achieve a 100% F_1 score on MultipleKnapsack, because, with our current constraint language, we cannot model that the sum of the item weights in a knapsack must not exceed its capacity. In general, we need a new type of constraint to model lower and upper bounds on the sum of numerical relation values. We see computing the number of valid assignments for such models is even harder than for count constraints, and thus is a challenging problem. Ideally, we would extend our constraint language to the global constraint catalog by Beldiceanu, Carlsson, and Rampon [16], which lists a large set of reusable constraints for constraint programming problems.

Second, the size of a satisfiability problem massively impacts the runtime and sample complexity of UrPiLs. While UrPiLs finds all constraints in the majority of runs from 40 examples of 4-Sudoku-hard, it only finds all constraints one out of ten times from 1000 examples of 9-Sudoku-hard. However, the rules of 4×4 and 9×9 Sudoku are basically the same, and many problems have constraints that are independent of the problem size. We, therefore, think it is promising to study how to reduce the size of a given problem as a preprocessing step. Other ways to improve performance on high dimensional problems may include

expert knowledge to restrict the large search space of constraints, e.g. by symmetries in the assignments or active learning.

Finally, we see an interesting connection between ConSequence in Chapter 4 and UrPiLs. While ConSequence discovers *positive* rules that predict events for a given attribute vector, UrPiLs can find *negative* constraints for executing events.

6.8 CONCLUSION

To close the gap between domain experts and mathematical modeling experts in constraint programming and AI planning, we studied the problem of discovering constraints from exemplary solutions. We formalized the problem in terms of the Minimum Description Length (MDL) principle, by which we select the model with the best lossless compression of the data. Since solving the problem involves #P-hard model counting, we proposed the greedy UrPiLs algorithm to find high-quality constraints in practice. Through extensive experiments on both constraint programming and AI planning benchmark datasets, we empirically showed UrPiLs not only discovers more accurate constraints, but also finds more succinct and hence interpretable constraints, is more robust to noise, and has lower sample complexity than the state of the art. To apply UrPiLs on more complex problems, potential future work involves extending its modeling language and improving its efficiency on high dimensional problems.

7

FROM PROSEQO TO URPILS: PUTTING IT ALL TOGETHER

In this chapter, we demonstrate how the different solutions proposed in the previous chapters can be used together to gain insight into a business process. Furthermore, we discuss remaining challenges and open research questions for future work.

7.1 INTRODUCTION

Owners of both service-oriented and manufacturing processes design, analyze and improve their processes using discrete-event simulation [57]. Creating process models not only involves a lot of manual effort, it also requires the rarely available combination of mathematical modeling and process domain knowledge, and often results in over-simplified models not fitting the actual process behavior well [1]. Therefore, analyzing event logs promises to alleviate the effort in building simulation models and to improve their accuracy [2].

In this chapter, we create and evaluate a simulation model of an exemplary real-world process through applying the methods we previously introduced in this dissertation. As our exemplary event log, we use the Rolling Mill event log, which we already used in Chapter 2, Chapter 4 and Chapter 5. The events in this dataset refer to production steps of a steel producing company, and each event sequence belongs to one produced plate. The attributes in this dataset refer to product categories and product dimensions.

More specifically, we create a discrete-event simulation, where we model the rolling mill by a network of queueing models. First we need to identify the process layout, i. e., the nodes in the queueing network. To this end, we use ProSimple from Chapter 2 to get an easily understandable visual representation of the process layout. As we need to predict paths of individual plates through the queueing network, we discover routing rules of plates with ConSequence from Chapter 4. Next, we learn queueing models for the different nodes in the net-

work using CueMin from Chapter 5, by which we have an executable process model for predicting throughput and bottlenecks. Finally, we discuss potential optimization problems for which UrPiLs from Chapter 6 could be used to learn constraints, as well as remaining challenges and open research questions for future work.

7.2 CREATING THE SIMULATION MODEL

In this section, we give an overview of our steps to create a simulation model for the underlying process of the Rolling Mill event log.

7.2.1 *Modeling the Process Layout*

Before we can learn the behavior of different production stations in the rolling mill, we first need to model the layout of the rolling mill. Normally, we would need a lot of domain knowledge to identify the most important parts of the process and how they connect to each other. Purely relying on interviews of domain experts would not only cost a lot of time, it also involves the risk of an over-simplified model focusing on the expected instead of the actual process behavior.

Using Prosimple, we discover the process layout from the given Rolling Mill event log, and obtain a graph of event patterns capturing the actual process behavior yet simple enough to be easily understood. We show the pattern graph in Figure 7.1. To facilitate understanding the result, we highlight different parts of the process using different colors. Furthermore, we provide a descriptive label for each part in Figure 7.2. At the beginning of the process, the plates are rolled at rolling-stands to meet their customer defined thickness. This happens at high temperatures and forces, otherwise, thickness reduction would not be possible. Therefore, we call this part of the rolling mill *hot zone*. Next, production in this rolling mill splits into two separate lines for thicker and thinner plates.

Large scissors cut thin plates into their final size. Depending on the customer's order, multiple special production steps ensure that the plates meet the desired product properties. As in most manufacturing processes, quality control releases delivery, before the plates can be loaded to transport. This gives us the layout for our simulation model.

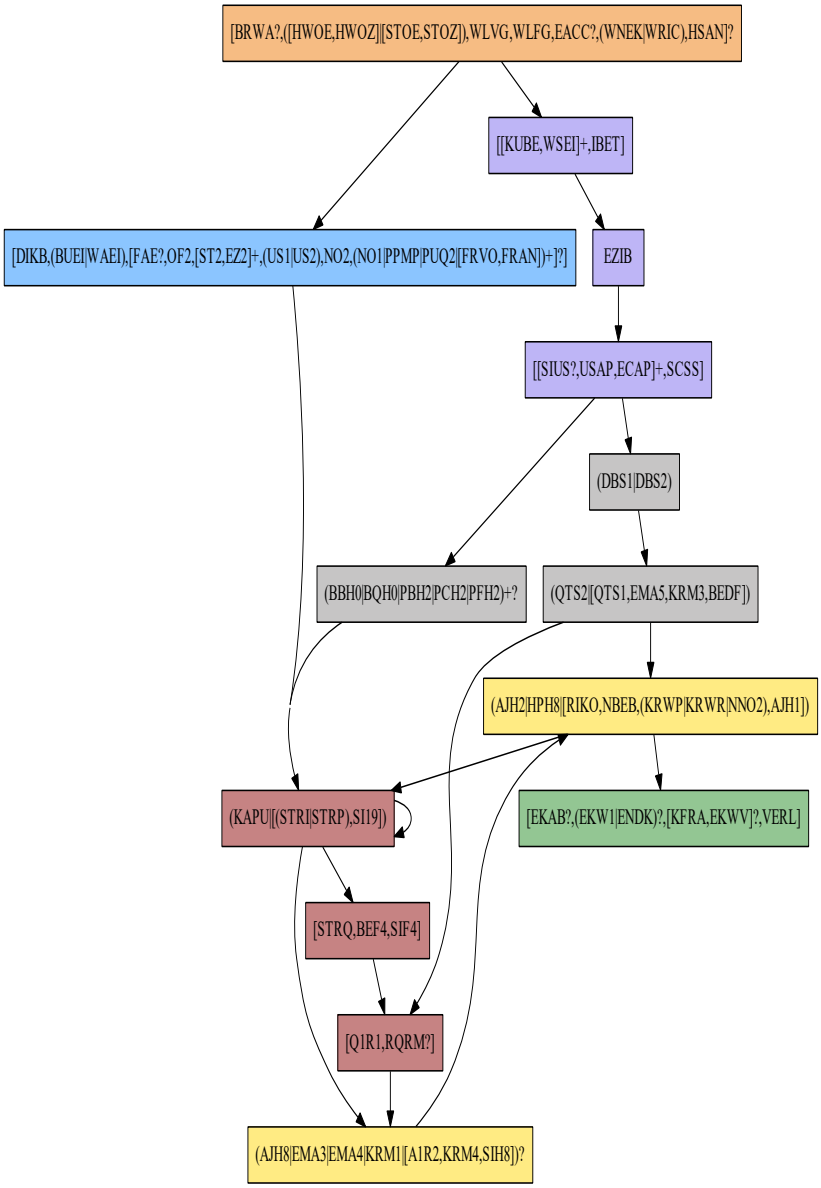


Figure 7.1: [**Rolling Mill Pattern Graph**] Model discovered by ProSimple (with $r = 1.5$) on the *Rolling Mill* dataset. We highlight essential parts of the production process with colors. To further facilitate understanding, we label pattern nodes in Figure 7.2.

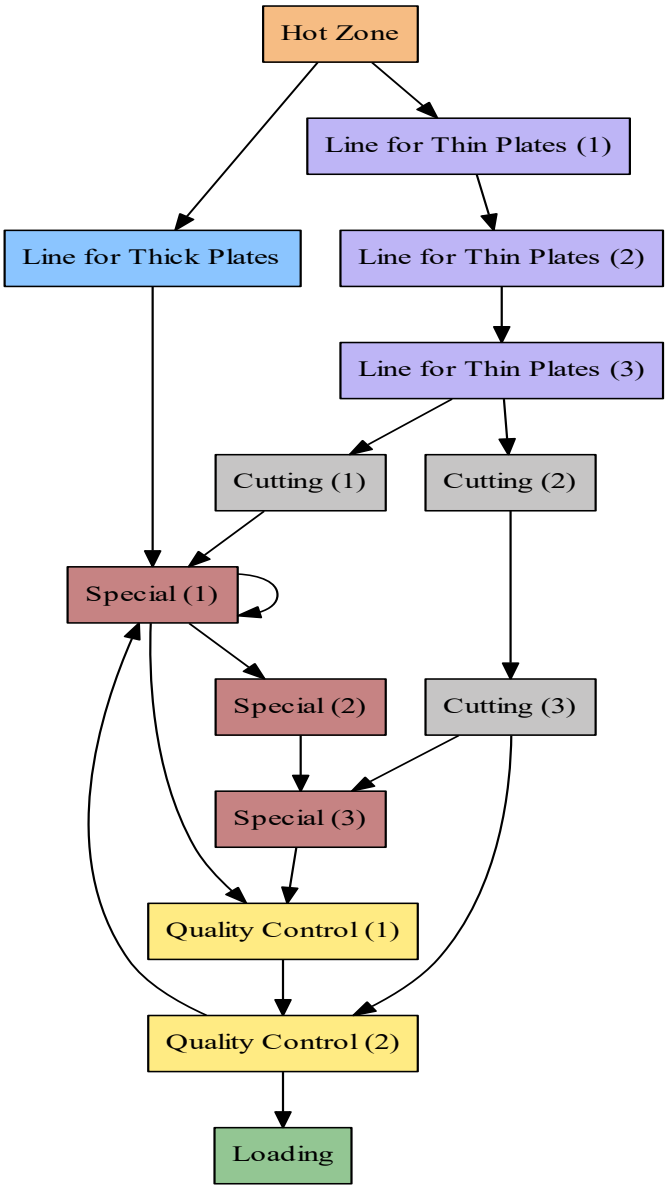


Figure 7.2: **[Simulation model layout]** Relabeled pattern graph from Figure 7.2. We highlight different parts of the process with colors, and give a descriptive label for each node in the graph.

7.2.2 *Predicting Production Steps*

Description alone is great, but what we often actually need is an executable simulation model to study different production scenarios. As we want to use the simulation model for analyzing different possible scenarios with unseen plates, we need to predict the paths of these plates through the rolling mill. Therefore, we learn rules to predict event sequences using ConSequence. We directly map the events in the event sequences to nodes in the pattern graph. For more details of the discovered rules, we refer to our case study in Section 4.6.6.

7.2.3 *Learning Queueing Models*

To complete our simulation model, we need to model the behavior of the individual production stations in the rolling mill. Since we are interested in predicting sojourn and waiting times, we model each process station by a waiting queue. As before, we normally would need a lot of domain knowledge to model the actual behavior, and interviewing domain experts involves the risk of an over-simplified model. Instead, we learn queueing models using CueMin.

To this end, we must choose the right level of abstraction. In the extreme case, we learn a separate queueing model for each event $e \in \Omega$ of the event alphabet. The arrival time of each plate at one event refers to the completion timestamp of the previous event, and the departure time refers to the completion timestamp of the current event. We refer to this model as one *queue per event* (QpE).

In the second approach, we learn a queueing model for each node in the pattern graph. The arrival time of each plate at one node refers to the completion timestamp of the last event that happened before the first event that is part of the pattern of the current node. We refer to this model as one *queue per pattern* (QpP). Since QpE involves more queues on a lower abstraction level, it should be able to capture more complex behavior, which may result in higher simulation accuracy. QpP results in fourteen instead of 264 queues, and thus is easier to understand. An open question for the experiments is whether QpE really achieves higher accuracy, or whether the higher model complexity leads to unexpected errors.

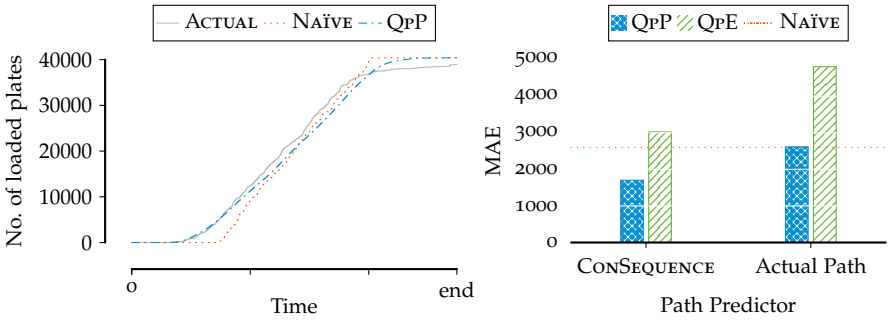


Figure 7.3: **[Cumulated number of loaded plates]** Actual number of loaded plates over time versus number of loaded plates predicted by the Naïve mean sojourn time predictor and by the one queue per pattern (QPP) simulation model (left), and the corresponding mean absolute error of the one queue per pattern (QPP) and one queue per event (QPE) simulation model using ConSequence for path prediction versus using the actual path (right, lower is better).

7.3 EXPERIMENTS

Now, we evaluate our simulation model. We simulate arrival of all plates in the dataset with their known arrival timestamp. Since the queueing models as part of our simulation model contain stochastic behavior, we conduct 100 independent runs.

7.3.1 *Throughput Prediction*

First, we evaluate predictions on production throughput. We average the predicted number of loaded plates at each day across all simulation runs. We compare the resulting cumulated number of loaded plates with the actual data. As a simple prediction baseline, we compare to a model that predicts the loading time of a plate by the sum of its arrival time and the mean time until loading over all plates in the data.

We show the actual cumulated number of loaded plates as well as the prediction by the QPP simulation model and the Naïve baseline on the left of Figure 7.3. We see that QPP is closer to the actual line than Naïve especially at the beginning of the observed time range. At the middle part of the time range, both methods are almost on par with

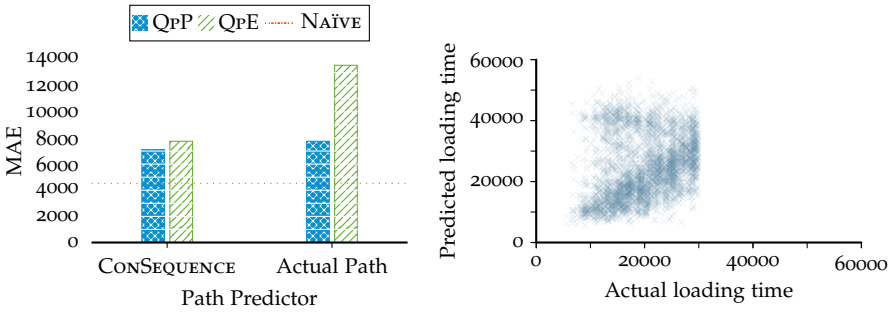


Figure 7.4: [**Evaluating loading time predictions**] Mean absolute error on the predicted loading time for QPP and QPE using CONSEQUENCE for path prediction versus using the actual path in comparison to NAÏVE (left, lower is better), and actual loading time versus predicted loading time for QPP with CONSEQUENCE (right).

a slightly better fit by NAÏVE. Then, QPP gets closer again, before the lines eventually converge.

We report the mean absolute error (MAE) on the number of loaded plates for QPP and QPE on the right of Figure 7.3. Furthermore, we show how the MAE changes when we use the actual event sequence instead of the path predicted by CONSEQUENCE. QPP with CONSEQUENCE has the lowest MAE, and is the only model that beats the NAÏVE baseline. QPP fits the data better than QPE.

7.3.2 *Loading Time Prediction*

Next, we evaluate predictions of the loading time for individual jobs. We show the mean absolute error on the loading time for QPP and QPE in comparison to the NAÏVE mean sojourn time predictor on the left of Figure 7.4. We see that both simulation models perform worse than the baseline. As before, using the actual path instead of the path predicted by CONSEQUENCE leads to a higher error.

To examine the reason why the simulation model QPP predicts well on the average loading time as we saw in the previous section but bad on the loading time of individual plates, we plot actual versus predicted loading time on the right of Figure 7.4. Since the simulation runs empty after no more new plates arrive at the start node, which makes interpreting predictions hard, we only plot results for jobs with

an actual loading time before the last plate arrives. This is why the last actual loading time is around time step 30000. We see that the simulation model overestimates loading time for a large set of plates.

7.4 DISCUSSION

ProSimple, ConSequence and CueMin enable quick creation of simulation models for business processes, from which process analysts and domain experts gain valuable insight of the event flow and queueing behavior. Moreover, the resulting simulation model predicts throughput well. Since all parts of the model consist of interpretable and understandable building blocks, domain experts can modify the model using their process knowledge to increase accuracy. For instance, they can incorporate business rules for production scheduling or maintenance time windows.

As soon as the model gives a sufficiently accurate image of the real process, production planners can analyze different scenarios to avoid bottlenecks and to support investment decisions [53, 57]. Artificial data generated for scenario analysis should follow the rules of real-world data. We see that Moody is able to support this task. In addition, if routing rules depend on event data attributes, Moody can be integrated into the simulation to predict change of those attributes. Furthermore, we see two promising use cases for UrPiLs in process simulation. First, we can replace ConSequence's deterministic path prediction by constraints that allow multiple paths. Second, we can learn constraints for optimization tasks such as scheduling.

Although our short case study on the Rolling Mill dataset unveils promising opportunities for understanding, predicting and optimizing business process using data, we see many interesting remaining challenges and open research questions for future work. First, we see by our experiments that the simulation model is good at predicting behavior of the whole population but bad at predicting times for single process instances. Similarly, using the actual event sequence instead of the sequence predicted by ConSequence should not lead to worse accuracy. While both results require further analysis to better understand the actual reasons, we already see multiple directions to improve the simulation model. Instead of learning queueing models independently, future work should focus on a better integration be-

tween queueing models in the network. To better predict individual instances, we need to improve modeling of the service order. In addition, the current model is not able to explain waiting times, which are not caused by system load.

Finally, we see that the difference between QPP and QPE raises the question of the right level of abstraction. While we ran PROSIMPLE and CUEMIN independently of each other, we think integrating the waiting queue perspective by CUEMIN into process discovery is promising. In other words, a good process model such as a pattern graph should lead to a good network of queueing models.

Traditionally being two separated research fields, we see the ongoing trend of business process simulation and process mining growing together. The majority of existing work focuses on extracting knowledge from event data to support domain experts to develop simulation models [2, 115] or digital twins [102]. Semi-automated modelling approaches [4, 100] further alleviate the manual effort to create those models. Future work faces the challenge to increase the level of automation in creating simulation models and digital twins. Last but not least, there are many state-of-the-art algorithms that solve specific parts of the problem, such as process discovery, control-flow prediction, learning waiting queues, and remaining time prediction. Increasing their availability and usage as well as integrating them into easily applicable solutions is an interesting and promising challenge.

7.5 CONCLUSION

In this chapter, we discussed the potential to combine the approaches we proposed in the previous chapters by conducting a short case study on the Rolling Mill dataset. We used PROSIMPLE, CONSEQUENCE and CUEMIN to create a simulation model of the rolling mill. Through two initial experiments, we showed that the model gives promising results in predicting overall production throughput, but is unable to predict satisfyingly for individual plates. From these results, we derived multiple open research questions. Combining business process simulation and process mining is an ongoing trend, and future work will increase the level of automation in creating simulation models.

To make our methods available in their latest version, we publish our code under the Github account of SHS Stahl-Holding-Saar, by which

the author of this thesis is employed, as ready-to-use Python libraries.[1] We hope our methods will help both researchers and practitioners to gain better insight into their business processes and event data.

1 https://github.com/shs-it/?q=prolothar

8

CONCLUSION

Finally, we draw a conclusion from our contributions, discuss limitations and give an outlook on future work and open challenges with regard to the research problems we defined in Chapter 1.

8.1 SUMMARY OF CONTRIBUTIONS

In this dissertation, we addressed how we can use event data to understand, predict and optimize business processes. We formulated and addressed a research problem for each of these three process analysis steps. The first problem was

Problem 1 (Understanding Process Behavior) *Given an event log, discover models summarizing the control-flow of events and how event data changes throughout the process.*

Since real-world processes exhibit complex behavior, where actual process behavior has much larger variance than the behavior expected by domain experts, understanding the control-flow of events is easier said than done. We tackled this problem in Chapter 2. To distinguish between relevant and irrelevant behavior, we formalized the problem of discovering a graphical model summarizing the control-flow of events using the Minimum Description Length (MDL) principle, i. e., we selected the model with the smallest lossless description of the data. As finding the best model is a NP-hard problem, we proposed our greedy algorithm ProSeqo to discover good models in practice.

ProSeqo starts with the directly-follows graph of an event log, i. e., an overfitting model, where each event is a node, and there is a directed edge from one event a to another event b, if b directly follows a at least once in the data. Then, ProSeqo iteratively simplifies the model by removing edges, nodes, and summarizing nodes into event sequence patterns, until MDL tells us to stop. Whenever this results is still too complex for domain experts to understand, we proposed the ProSimple algorithm to remove further edges, until the model satisfies

a user-defined graph density threshold. Experiments on both synthetic and real-world data validated that our approaches are robust to noise and work well in enabling a better understanding of the control-flow.

In Chapter 3, we tackled the event data perspective of the first research problem. How data changes throughout a process is a surprisingly understudied topic. To close this gap, we proposed the MOODY algorithm to discover interpretable if-then rules summarizing how event data changes. Through extensive experiments on both synthetic and real-world data, we empirically showed MOODY finds succinct rules, needs little data for accurate discovery, is robust to sensible amounts of noise, and thus gives valuable insight into data modifications.

After providing novel approaches to better understand a process, we focused on the second research problem, which we defined as

Problem 2 (Predicting Process Behavior) *Given an event log, find models to predict event sequences with their activities, event data and timestamps.*

To tackle this problem, we learned a sparse event-flow graph over the training sequences, and statistically robust rules that use trace attributes to determine which paths to follow, by proposing the CONSEQUENCE algorithm in Chapter 4. Since the event-flow graph and the decision rules are easily human-readable, CONSEQUENCE in contrast to deep neural networks enables truly interpretable event sequence prediction. Through an extensive set of experiments including a case study, we showed our approach produces compact, interpretable and accurate models, is robust against noise and has low sample complexity, which enables applicability on a wide range of real-world datasets.

To manage customer satisfaction and delivery reliability, predicting time as well as detecting and preventing bottlenecks is particularly important in service-oriented and manufacturing processes. Event timestamps often result from waiting times in the process, which we can explain by queueing models. We studied discovering queueing models for interpretable time prediction from data, and proposed the CUEMIN algorithm to tackle this problem in Chapter 5. As we left combining multiple queueing models into a network of waiting queues as future work when proposing CUEMIN, we approached a first solution by combining PROSIMPLE, CONSEQUENCE and CUEMIN in Chapter 7.

Since the final goal of any process analysis is process enhancement, we defined the third research problem as

Problem 3 (Optimizing Process Behavior) *Given an event log, find measurements to improve and optimize process behavior.*

Reducing anomalies increases efficiency and makes process behavior more predictable. Process models as the ones discovered by PROSEQO and PROSIMPLE help domain experts to identify anomalies. Production planners can use queueing models to prevent bottlenecks and optimize throughput. Furthermore, queueing models support investment decisions, such as hiring more employees or buying new machines to increase service capacity. These contributions to process optimization are rather indirect, i. e., they help to gain the necessary understanding and help to create a simulation model of process behavior, which we can use in optimization use cases. In a more direct approach, both constraint programming and AI planning are powerful tools to solve optimization problems such as scheduling. Formulating optimization constraints, however, requires extensive domain knowledge and manual modeling effort. Therefore, we proposed the URPILS algorithm to discover constraints for constraint programming and AI planning from exemplary solutions in Chapter 6.

In summary, we proposed multiple novel approaches to understand, predict, and optimize process behavior using data. Through extensive experiments, we empirically showed that our methods discover interpretable yet accurate models, are robust against noise, and have low sample complexity. Many companies face a disruptive digital transformation, where fast business decisions are necessary but wrong business decisions, such as investing into new production machines, may endanger the survival of the company. By using our methods, domain experts and process owners can make data-driven process optimizations, and build digital twins to test any change to the process on an accurate digital representation before modifying the real process.

8.2 LIMITATIONS

Naturally, each method has its limitations, and we see a lot of interesting research directions to address them. First, PROSEQO, PROSIMPLE and CONSEQUENCE discover good models for processes without concurrent activities. Many real-world processes, however, consist of branches with parallel activities. For instance, in the Sepsis event log, different laboratory tests are conducted concurrently instead of se-

quentially. Detecting concurrent behavior either as a preprocessing step or integrated into the model search of the three methods would lead to better results on more real-world processes.

Since we use the MDL principle, our model selection is based on a well-founded induction by compression theory, resulting in noise-robust approaches and simple yet accurate models. However, MDL is not a magic wand: The encoding we propose includes choices. Different choices may lead to different, and possibly better, models. As a more serious downside, discrete optimization is a bottleneck in all our approaches. While our approaches scale favorably with the number of training instances, discrete optimization impedes fast runtimes on very complex datasets with thousands of different events or event attributes. Runtimes of multiple hours or even days prevent application with real-time requirements. Increasing complexity in the modeling language for methods like ProSeqo, CueMin or UrPiLs to tackle real-world processes with more complex behavior, results in an even larger search space. Thus, we see high potential in optimizing model search. One option is to examine application of state-of-the-art discrete optimization methods specialized on problems with expensive evaluation functions [33]. As an alternative, we could make the search space differentiable by using a neural autoencoder with binary activations [46].

While we evaluated on datasets from several domains such as different service-oriented and manufacturing processes, we yet do not make use of domain knowledge to find better solutions faster. As all our models consist of interpretable building blocks, one option to include such knowledge is by domain experts adapting the model or restricting the model search space. This, however, requires a lot of manual effort, and domain knowledge is not always easily available. To this end, we expect similar processes to behave similar across different companies, such that exploiting ontologies promises to enhance model search [39, 61, 70, 94]. Whenever ontologies are not available or incomplete, natural language processing can be used to extract semantic information from event logs [121].

As process mining is a huge, fast-emerging and multi-faceted research field, we could not address all of its subtopics in this dissertation. Before we can analyze event data, we first have to collect it. Data acquisition of event logs is a non-trivial task, which often requires domain knowledge [133], and thus is an obstacle for process mining

projects in practice. To further increase industrial adoption of process mining, we consider simplifying collecting and accessing high-quality, well-documented event data a key factor.

In all our contributions, we assumed a static event log. However, real-world processes usually are not static. Therefore, it would be interesting to extend our methods to continuous learning and concept drift detection [21, 131] on event stream data [165].

We focused on events that belong to exactly one sequence and all events are on the same abstraction level. Event data, however, appears in many variants. In object-centric event logs [1, 3], each event refers to the change of the state of one or multiple objects, where objects as in object-oriented programming belong to different classes, and there are relationships between objects. As an example, consider an order that belongs to one customer and consists of multiple items, and multiple items of different customers can be transported by the same truck. In multi-level event logs [81], events belong to different abstraction levels, such that one event embraces a set of lower-level events.

In summary, to make our approaches applicable to even more real-world processes and use cases, potential future work comprises extending the modeling language to fit more complex behavior, optimizing model search, using domain knowledge from ontologies, and supporting different types of event logs.

8.3 OUTLOOK

As process mining and event log analysis are evolving fast, we conclude this dissertation with an outlook based on current developments. Since real-world process event data is rarely publicly available, contains nondeterministic behavior, and often comes from very special domains, evaluating tasks such as process model discovery, sequence prediction, and anomaly detection in a reproducible and sound manner is challenging. Rehse and Fettke [122] report significant evaluation problems, such as unrepresentative datasets, no proper comparison to other methods, deficient metrics, and a lack of reproducibility. The community addresses these issues by developing and studying new quality metrics [60, 111, 157] as well as creating new and larger benchmark datasets [79, 111]. Therefore, we expect that sound and re-

producible benchmarking of analyzing and predicting event log data becomes easier and more standardized.

Improved reproducible benchmarking enables intensified publishing on AI and data mining conferences, such that researchers can profit from advanced knowledge across communities. We already observe that process mining quickly adopts recent developments and trends from the AI community, such as graph neural networks [26, 141] or large language models [18, 19, 44, 62]. Vice versa, process mining offers challenging problems of great industrial interest, and thus we expect AI and data mining researchers to put a stronger focus on pattern mining for process event sequences, next event prediction, event sequence anomaly detection, and other process mining related tasks.

The dynamics of a currently ongoing digital transformation [73] force companies to adapt their business model and innovate faster than ever before. Companies that do not transform rapidly enough risk to fall behind and fail in competition [56]. Digital twins, i. e., accurate digital representations of the real world, facilitate process design, monitoring, simulation and optimization, and thus support succeeding in the digital transformation [143]. However, creating and maintaining digital twins for existing processes requires both a lot of manual effort and domain knowledge. Since recent work highlights the potential of process mining for digital twins [12, 78, 104], process mining may lead to a wider adoption of digital twins in established service-oriented and manufacturing processes.

As we showed in this dissertation, our methods discover accurate yet easily understandable models from event logs, which domain experts can use to alleviate the effort of creating digital twins. We hope that further research will even more facilitate understanding, predicting and optimizing business processes.

BIBLIOGRAPHY

[1] Wil van der Aalst. *Process Mining – Data Science in Action*. 2nd ed. Springer, 2016.

[2] Wil van der Aalst. "Process mining and simulation: A match made in heaven!" In: *Proceedings of the Summer Simulation Multi-Conference (SummerSim), Bordeaux, France*. 2018. URL: https://dl.acm.org/doi/10.5555/3275382.3275386.

[3] Wil van der Aalst. "Object-Centric Process Mining: Dealing with Divergence and Convergence in Event Data." In: *Proceedings of the 17th International Conference on Software Engineering and Formal Methods (SEFM), Oslo, Norway*. 2019, pp. 3–25.

[4] Waleed Abohamad, Ahmed Ramy, and Amr Arisha. "A hybrid process-mining approach for simulation modeling." In: *Proceedings of the 2017 Winter Simulation Conference (WSC), Las Vegas, NV*. IEEE. 2017, pp. 1527–1538.

[5] Arya Adriansyah. "Aligning observed and modeled behavior." PhD thesis. TU Eindhoven, 2014.

[6] Rakesh Agrawal and Ramakrishnan Srikant. "Mining sequential patterns." In: *Proceedings of the 11th International Conference on Data Engineering (ICDE), Taipei, Taiwan*. 1995, pp. 3–14.

[7] Diego Aineto, Sergio J. Celorrio, and Eva Onaindia. "Learning action models with minimal observability." In: *Journal of Artificial Intelligence (AIJ)* 275 (2019), pp. 104–137.

[8] Tim Andersen and Tony Martinez. "NP-completeness of minimum rule sets." In: *Proceedings of the 10th International Symposium on Computer and Information Sciences (ISCIS), Kuşadası, Turkey*. 1995, pp. 411–418.

[9] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein generative adversarial networks." In: *Proceedings of the 34th International Conference on Machine Learning (ICML), Sydney, Australia*. 2017, pp. 214–223.

[10] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty. "A review of learning planning action models." In: *The Knowledge Engineering Review* 33 (2018), e20.

[11] Adriano Augusto, Raffaele Conforti, Marlon Dumas, and Marcello La Rosa. "Split Miner: Discovering Accurate and Simple Business Process Models from Event Logs." In: *Proceedings of the 17th IEEE International Conference on Data Mining (ICDM), New Orleans, LA*. 2017, pp. 1–10.

[12] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Mathias Weske. "Process-aware digital twin cockpit synthesis from event logs." In: *Journal of Computer Languages* 70 (2022), p. 101121.

[13] Roman Barták. "Constraint programming: In pursuit of the holy grail." In: *Proceedings of the 8th Annual Conference of Doctoral Students (WDS), Prague, Czech Republic*. 1999, pp. 555–564.

[14] Ron Begleiter, Ran El-Yaniv, and Golan Yona. "On prediction using variable order Markov models." In: *Journal of Artificial Intelligence Research* 22 (2004), pp. 385–421.

[15] Mohamed-Bachir Belaid, Nassim Belmecheri, Arnaud Gotlieb, Nadjib Lazaar, and Helge Spieker. "GEQCA: Generic Qualitative Constraint Acquisition." In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*. 2022, pp. 3690–3697.

[16] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. *Global constraint catalog, 2nd edition (revision a)*. Tech. rep. Swedish Institute of Computer Science, 2012.

[17] Alessandro Berti and Wil van der Aalst. "Reviving Token-based Replay: Increasing Speed While Improving Diagnostics." In: *Proceedings of the International Workshop on Algorithms and Theories for the Analysis of Event Data*. 2019, pp. 87–103.

[18] Alessandro Berti and Mahnaz S. Qafari. *Leveraging Large Language Models (LLMs) for Process Mining (Technical Report)*. 2023. arXiv: 2307.12701.

[19] Alessandro Berti, Daniel Schuster, and Wil van der Aalst. *Abstractions, Scenarios, and Prompt Definitions for Process Mining with LLMs: A Case Study*. 2023. arXiv: 2307.02194.

[20] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. "Constraint acquisition via partial queries." In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), Beijing, China*. 2013, pp. 475–481.

[21] R.P. Jagadeesh Chandra Bose, Wil van der Aalst, Indrė Žliobaitė, and Mykola Pechenizkiy. "Dealing with concept drifts in process mining." In: *IEEE Transactions on Neural Networks and Learning Systems* 25.1 (2013), pp. 154–171.

[22] Lawrence Brown, Noah Gans, Avishai Mandelbaum, Anat Sakov, Haipeng Shen, Sergey Zeltyn, and Linda Zhao. "Statistical analysis of a telephone call center: A queueing-science perspective." In: *Journal of the American Statistical Association* 100.469 (2005), pp. 36–50.

[23] Kailash Budhathoki, Mario Boley, and Jilles Vreeken. "Discovering Reliable Causal Rules." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Virtual Event*. 2021, pp. 1–9.

[24] Manuel Camargo, Marlon Dumas, and Oscar González-Rojas. "Learning accurate LSTM models of business processes." In: *Proceedings of the 17th International Conference on Business Process Management (BPM), Vienna, Austria*. 2019, pp. 286–302.

[25] Allan Cheng, Javier Esparza, and Jens Palsberg. "Complexity results for 1-safe nets." In: *Proceedings of the 11th International Conference on Foundations of Software Technology and Theoretical Computer Science, Bombay, India*. 1993, pp. 326–337.

[26] Andrea Chiorrini, Claudia Diamantini, Alex Mircoli, and Domenico Potena. "Exploiting instance graphs and graph neural networks for next activity prediction." In: *Proceedings of the 3rd International Conference on Process Mining (ICPM), Eindhoven, The Netherlands*. 2021, pp. 115–126.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. 3rd ed. The MIT Press, 2009.

[28] Christopher Coulombe and Claude-Guy Quimper. "Constraint Acquisition Based on Solution Counting." In: *28th International Conference on Principles and Practice of Constraint Programming (CP), Haifa, Israel*. 2022.

[29] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience New York, 2006.

[30] Pierluigi Crescenzi and Gianluca Rossi. "On the Hamming distance of constraint satisfaction problems." In: *Theoretical Computer Science* 288.1 (2002), pp. 85–100.

[31] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. "Counting models for 2SAT and 3SAT formulae." In: *Theoretical Computer Science* 332.1-3 (2005), pp. 265–291.

[32] D. J. Daley. "General customer impatience in the queue GI/G/1." In: *Journal of Applied Probability* 2.1 (1965), pp. 186–205.

[33] Aryan Deshwal, Syrine Belakaria, Janardhan Rao Doppa, and Alan Fern. "Optimizing discrete spaces via expensive evaluations: A learning to search framework." In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), New York, NY*. 2020, pp. 3773–3780.

[34] Boudewijn van Dongen. *BPI Challenge 2015*. 2017. DOI: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.

[35] Boudewijn van Dongen. *BPI Challenge 2017*. 2017. DOI: 10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[36] Boudewijn van Dongen. *BPI Challenge 2019*. 2019. DOI: 10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1.

[37] Bogdan Doytchinov, John Lehoczky, and Steven Shreve. "Real-time queues in heavy traffic with earliest-deadline-first queue discipline." In: *The Annals of Applied Probability* (2001), pp. 332–378.

[38] Nan Du, Hanjun Dai, Rakshit Trivedi, Utkarsh Upadhyay, Manuel Gomez-Rodriguez, and Le Song. "Recurrent marked temporal point processes: Embedding event history to vector." In: *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), San Francisco, CA*. 2016, pp. 1555–1564.

[39] Simon Eichele, Knut Hinkelmann, and Maja Spahic-Bogdanovic. "Ontology-Driven Enhancement of Process Mining With Domain Knowledge." In: *Proceedings of the AAAI 2023 Spring Symposium on Challenges Requiring the Combination of Machine Learning and Knowledge Engineering (AAAI-MAKE), San Francisco, CA*. 2023.

[40] Patrick Favre-Bulle. *Density Image Converter Tool for Android, iOS, Windows and CSS*. `https://github.com/patrickfav/density-converter` (accessed on 01-08-2023). 2020.

[41] Usama Fayyad and Keki Irani. "On the handling of continuous-valued attributes in decision tree generation." In: *Machine Learning* 8 (1992), pp. 87–102.

[42] William Feller. *Introduction to Probability Theory and Its Applications*. 3rd ed. Vol. 1. Wiley, 1968.

[43] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. "The Model Counting Competition 2020." In: *ACM Journal of Experimental Algorithmics (JEA)* 26 (2021), pp. 1–26.

[44] Hans-Georg Fill, Peter Fettke, and Julius Köpke. "Conceptual modeling and large language models: impressions from first experiments with ChatGPT." In: *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling* 18 (2023), pp. 1–15.

[45] Jonas Fischer and Jilles Vreeken. "Sets of robust rules, and how to find them." In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Würzburg, Germany*. 2019, pp. 38–54.

[46] Jonas Fischer and Jilles Vreeken. "Differentiable pattern set mining." In: *Proceedings of the 27th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Virtual Event*. 2021, pp. 383–392.

[47] Michael Foster, John Derrick, and Neil Walkinshaw. "Reverse-Engineering EFSMs with Data Dependencies." In: *Proceedings of the 33rd IFIP International Conference on Testing Software and Systems (ICTSS), Virtual Event*. 2021, pp. 37–54.

[48] Jaroslav Fowkes and Charles Sutton. "A subsequence interleaving model for sequential pattern mining." In: *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), San Francisco, CA*. 2016, pp. 835–844.

[49] Esther Galbrun, Peggy Cellier, Nikolaj Tatti, Alexandre Termier, and Bruno Crémilleux. "Mining periodic patterns with a MDL criterion." In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Dublin, Ireland*. 2018, pp. 535–551.

[50] Albert Gatt and Emiel Krahmer. "Survey of the state of the art in natural language generation: Core tasks, applications and evaluation." In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 65–170.

[51] Thomas Grisold, Jan Mendling, Markus Otto, and Jan vom Brocke. "Adoption, use and management of process mining in practice." In: *Business Process Management Journal* 27.2 (2021), pp. 369–387.

[52] Peter Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[53] Randolph W. Hall. *Queueing Methods: For Services and Manufacturing*. Prentice-Hall, 1990.

[54] Oliver Handel and André Borrmann. "The relationship between the waiting crowd and the average service time." In: *Proceedings of the 11th International Conference on Traffic and Granular Flow (TGF), Delft, The Netherlands*. 2015, pp. 209–216.

[55] Peter Hart, Nils Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[56] Thomas Hess, Christian Matt, Alexander Benlian, and Florian Wiesböck. "Options for formulating a digital transformation strategy." In: *MIS Quarterly Executive* 15.2 (2016).

[57] Vlatka Hlupic and Stewart Robinson. "Business process modelling and analysis using discrete-event simulation." In: *Proceedings of the 30th Winter Simulation Conference (WSC), Washington, DC*. 1998, pp. 1363–1369.

[58] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[59] Laurent Hyafil and Ronald Rivest. "Constructing optimal binary decision trees is NP-complete." In: *Information Processing Letters* 5.1 (1976), pp. 15–17.

[60] Ashish T.S. Ireddy and Sergey V. Kovalchuk. "An Experimental Outlook on Quality Metrics for Process Modelling: A Systematic Review and Meta Analysis." In: *Algorithms* 16.6 (2023).

[61] Wirat Jareevongpiboon and Paul Janecek. "Ontological approach to enhance results of business process mining and analysis." In: *Business Process Management Journal* 19.3 (2013), pp. 459–476.

[62] Urszula Jessen, Michal Sroka, and Dirk Fahland. *Chit-Chat or Deep Talk: Prompt Engineering for Process Mining*. 2023. arXiv: 2307.09909.

[63] Apratim Bhattacharyya and Jilles Vreeken. "Efficiently summarising event sequences with rich interleaving patterns." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Houston, TX*. 2017, pp. 795–803.

[64] Norman L. Johnson, Samuel Kotz, and Adrienne W. Kemp. *Univariate discrete distributions*. John Wiley & Sons, 2005.

[65] Toon Jouck and Benoît Depaire. "PTandLogGenerator: A Generator for Artificial Event Data." In: *Proceedings of the 14th International Conference on Business Process Management (BPM) Demonstration Track, Rio de Janeiro, Brazil*. CEUR, 2016, pp. 23–27.

[66] Andrew Keith, Darryl Ahner, and Raymond Hill. "An order-based method for robust queue inference with stochastic arrival and departure times." In: *Computers & Industrial Engineering* 128 (2019), pp. 711–726.

[67] J. F. C. Kingman. "The first Erlang century – and the next." In: *Queueing Systems: Theory and Applications* 63.1-4 (2009), pp. 3–12.

[68] Thomas Kittsteiner and Benny Moldovanu. "Priority auctions and queue disciplines that depend on processing time." In: *Management Science* 51.2 (2005), pp. 236–248.

[69] Eva L. Klijn and Dirk Fahland. "Performance mining for batch processing using the performance spectrum." In: *Proceedings of the 17th International Conference on Business Process Management (BPM), Vienna, Austria*. 2019, pp. 172–185.

[70] Dino Knoll, Julian Waldmann, and Gunther Reinhart. "Developing an internal logistics ontology for process mining." In: *Proceedings of the 12th Conference on Intelligent Computation in Manufacturing Engineering, Gulf of Naples, Italy*. 2019, pp. 427–432.

[71] Samuel Kolb, Sergey Paramonov, Tias Guns, and Luc De Raedt. "Learning constraints in spreadsheets and tabular data." In: *Machine Learning* 106 (2017), pp. 1441–1468.

[72] Tuukka Korhonen and Matti Järvisalo. "Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters." In: *27th International Conference on Principles and Practice of Constraint Programming (CP), Virtual Event*. 2021, 8:1–8:11.

[73] Marcin Kotarba. "Digital transformation of business models." In: *Foundations of Management* 10.1 (2018), pp. 123–142.

[74] Thomas Krismayer. "Automatic Mining of Constraints for Event-based Systems Monitoring." PhD thesis. Johannes Kepler University Linz, 2020.

[75] Mohit Kumar, Samuel Kolb, and Tias Guns. "Learning Constraint Programming Models from Data Using Generate-And-Aggregate." In: *28th International Conference on Principles and Practice of Constraint Programming (CP), Haifa, Israel*. 2022.

[76] Mohit Kumar, Samuel Kolb, Stefano Teso, and Luc De Raedt. "Learning MAX-SAT from Contextual Examples for Combinatorial Optimisation." In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI), New York, NY*. 2020, pp. 4493–4500.

[77] Mohit Kumar, Stefano Teso, Patrick De Causmaecker, and Luc De Raedt. "Automating personnel rostering by learning constraints using tensors." In: *Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI), Portland, OR*. 2019, pp. 697–704.

[78] Mahesh Kumbhar, Amos HC Ng, and Sunith Bandaru. "A digital twin based framework for detection, diagnosis, and improvement of throughput bottlenecks." In: *Journal of Manufacturing Systems* 66 (2023), pp. 92–106.

[79] Johannes Lahann, Peter Pfeiffer, and Peter Fettke. "LSTM-based anomaly detection of process instances: benchmark and tweaks." In: *Proceedings of the 4th International Conference on Process Mining (ICPM), Bozen-Bolzano, Italy*. Springer. 2022, pp. 229–241.

[80] Chris van der Lee, Emiel Krahmer, and Sander Wubben. "Automated learning of templates for data-to-text generation: comparing rule-based, statistical and neural methods." In: *Proceedings of the 11th International Conference on Natural Language Generation (INLG), Tilburg, The Netherlands*. 2018, pp. 35–45.

[81] Sander J.J. Leemans, Kanika Goel, and Sebastiaan J. van Zelst. "Using multi-level information in hierarchical process mining: Balancing behavioural quality and model complexity." In: 2020, pp. 137–144.

[82] Sander Leemans, Dirk Fahland, and Wil van der Aalst. "Discovering block-structured process models from event logs containing infrequent behaviour." In: *Business Process Management Workshops*. Vol. 171. Lecture Notes in Business Information Processing. Springer, 2014, pp. 66–78.

[83] Massimiliano de Leoni and Wil van der Aalst. "Aligning Event Logs and Process Models for Multi-Perspective Conformance Checking: An Approach Based on Integer Linear Programming." In: *Proceedings of the 11th International Conference on Business Process Management (BPM), Beijing, China*. Springer, 2013, pp. 113–129.

[84] Massimiliano de Leoni and Felix Mannhardt. *Road Traffic Fine Management Process*. 2015. DOI: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

[85] Vladimir Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals." In: *Soviet Physics – Doklady*. Vol. 10. 8. 1966, pp. 707–710.

[86] Dafna Levy. *Production Analysis with Process Mining Technology*. NooL – Integrating People & Solutions. 2014. DOI: 10.4121/uuid:68726926-5ac5-4fab-b873-ee76ea412399.

[87] Ming Li and Paul Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.

[88] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. "Automatic generation of software behavioral models." In: *Proceedings of the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany*. 2008, pp. 501–510.

[89] Felix Mannhardt. *Sepsis Cases - Event Log*. 2016. DOI: 10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[90] Felix Mannhardt, Massimiliano De Leoni, Hajo A. Reijers, and Wil Van Der Aalst. "Balanced multi-perspective checking of process conformance." In: *Computing* 98 (2016), pp. 407–437.

[91] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. "Discovering Frequent Episodes in Sequences." In: *Proceedings of the 1st ACM International Conference on Knowledge Discovery and Data Mining (KDD), Montreal, Canada*. 1995, pp. 210–215.

[92] Alexander Marx and Jilles Vreeken. "Telling cause from effect by local and global regression." In: *Knowledge and Information Systems* 60.3 (2019), pp. 1277–1305.

[93] Alexander Marx and Jilles Vreeken. "Formally justifying MDL-based inference of cause and effect." In: *Proceedings of the AAAI Workshop on Information-Theoretic Methods for Causal Inference and Discovery (ITCI), Virtual Event*. 2022.

[94] Ana Karla Alves de Medeiros, Wil van der Aalst, and Carlos Pedrinaci. "Semantic process mining tools: Core building blocks." In: *Proceedings of the European Conference on Information Systems (ECIS), Galway, Ireland*. 2008.

[95] Nijat Mehdiyev and Peter Fettke. "Explainable Artificial Intelligence for Process Mining: A General Overview and Application of a Novel Local Explanation Approach for Predictive Process Monitoring." In: *Computing Research Repository* abs/2009.02098 (2020).

[96] Tao Meng and Kai-Wei Chang. "An Integer Linear Programming Framework for Mining Constraints from Data." In: *Proceedings of the 38th International Conference on Machine Learning (ICML), Virtual Event*. 2021, pp. 7619–7631.

[97] Azadeh S. Mozafari Mehr, Renata M. de Carvalho, and Boudewijn van Dongen. "Detecting privacy, data and control-flow deviations in business processes." In: *Proceedings of the 33rd International Conference on Advanced Information Systems Engineering (CAiSE), Virtual Event*. 2021, pp. 82–91.

[98] Timo Nolle, Alexander Seeliger, and Max Mühlhäuser. "BINet: multivariate business process anomaly detection using deep learning." In: *Proceedings of the 16th International Conference on Business Process Management (BPM), Sydney, NSW, Australia*. 2018, pp. 271–287.

[99] Barry O'Sullivan. "Automated modeling and solving in constraint programming." In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI), Atlanta, GA*. 2010, pp. 1493–1497.

[100] Laura Johanna Oberle and Han van der Aa. "DDPS: A Project Methodology for Data-Driven Process Simulation." In: *Proceedings of Americas Conference on Information Systems (AMCIS), Panama City, Panama*. 2023, p. 1767. URL: https://aisel.aisnet.org/amcis2023/sig_dsa/sig_dsa/13.

[101] César Ojeda, Kostadin Cvejoski, Bodgan Georgiev, Christian Bauckhage, Jannis Schuecker, and Ramses J. Sanchez. "Learning Deep Generative Models for Queuing Systems." In: *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*. 2021, pp. 9214–9222.

[102] Alef B. Oliveira, Cleiton Santos, Eduardo Loures, and Eduardo A. P. Santos. "Design principles of digital twin: a process mining driven approach." In: *Proceedings of the 11th International Conference on Production Research Americas (ICPR), Curitiba, Brazil*. Springer. 2022, pp. 385–392.

[103] Fernando Otero and Alex Freitas. "Improving the Interpretability of Classification Rules Discovered by an Ant Colony Algorithm: Extended Results." In: *Evolutionary Computation* 24.3 (2016), pp. 385–409.

[104] Gyunam Park and Wil van der Aalst. "Realizing a digital twin of an organization using action-oriented process mining." In: *Proceedings of the 3rd International Conference on Process Mining (ICPM), Eindhoven, The Netherlands*. 2021, pp. 104–111.

[105] Vincenzo Pasquadibisceglie, Annalisa Appice, Giovanna Castellano, and Donato Malerba. "Using convolutional neural networks for predictive process analytics." In: *Proceedings of the 1st International Conference on Process Mining (ICPM), Aachen, Germany*. 2019, pp. 129–136.

[106] Tomasz P. Pawlak and Krzysztof Krawiec. "Automatic synthesis of constraints from examples using mixed integer linear programming." In: *European Journal of Operational Research* 261.3 (2017), pp. 1141–1157.

[107] Judea Pearl. *Causality: Models, Reasoning and Inference*. 2nd ed. Cambridge University Press, 2009.

[108] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. "Mining sequential patterns by pattern-growth: The prefixspan approach." In: *IEEE Transactions on Knowledge and Data Engineering* 16.11 (2004), pp. 1424–1440.

[109] François Petitjean, Tao Li, Nikolaj Tatti, and Geoffrey Webb. "Skopus: Mining top-k sequential patterns under leverage." In: *Data Mining and Knowledge Discovery* 30 (2016).

[110] Carl Adam Petri. "Kommunikation mit Automaten." PhD thesis. Institut für instrumentelle Mathematik Bonn, 1962.

[111] Peter Pfeiffer, Johannes Lahann, and Peter Fettke. "The Label Ambiguity Problem in Process Prediction." In: *Proceedings of the 4th International Conference on Process Mining (ICPM), Bozen-Bolzano, Italy*. 2022, pp. 37–44.

[112] Émilie Picard-Cantin, Mathieu Bouchard, Claude-Guy Quimper, and Jason Sweeney. "Learning parameters for the sequence constraint from solutions." In: *22nd International Conference on Principles and Practice of Constraint Programming (CP), Toulouse, France*. 2016, pp. 405–420.

[113] Mirko Polato, Alessandro Sperduti, Andrea Burattin, and Massimiliano de Leoni. "Time and activity sequence prediction of business process instances." In: *Computing* 100.9 (2018), pp. 1005–1031.

[114] George Pólya. "Eine Wahrscheinlichkeitsaufgabe in der Kundenwerbung." In: *Zeitschrift für Angewandte Mathematik und Mechanik* 10.1 (1930), pp. 96–97.

[115] Mahsa Pourbafrani and Wil van der Aalst. "Extracting process features from event logs to learn coarse-grained simulation models." In: *Proceedings of the 33rd International Conference on Advanced Information Systems Engineering (CAiSE), Virtual Event*. Springer. 2021, pp. 125–140.

[116] Steven D. Prestwich, Eugene C. Freuder, Barry O'Sullivan, and David Browne. "Classifier-based constraint acquisition." In: *Annals of Mathematics and Artificial Intelligence* 89 (2021), pp. 655–674.

[117] Steven Prestwich. "Unsupervised Constraint Acquisition." In: *Proceedings of the 33rd International Conference on Tools with Artificial Intelligence (ICTAI), Virtual Event*. 2021, pp. 256–262.

[118] Hugo Proença, Peter Grünwald, Thomas Bäck, and Matthijs van Leeuwen. "Robust subgroup discovery: Discovering subgroup lists using MDL." In: *Data Mining and Knowledge Discovery* 36.5 (2022), pp. 1885–1970.

[119] Hugo Proença and Matthijs van Leeuwen. "Interpretable multiclass classification by MDL-based rule lists." In: *Journal of Information Science* 512 (2020), pp. 1372–1393.

[120] Michael Rapp, Eneldo L. Mencía, Johannes Fürnkranz, Vu-Linh Nguyen, and Eyke Hüllermeier. "Learning gradient boosted multi-label classification rules." In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Virtual event*. 2020, pp. 124–140.

[121] Adrian Rebmann and Han van der Aa. "Enabling semantics-aware process mining through the automatic annotation of event logs." In: *Information Systems* 110 (2022). DOI: 10.1016/j.is.2022.102111.

[122] Jana-Rebecca Rehse and Peter Fettke. "Process mining crimes – a threat to the validity of process discovery evaluations." In: *Lecture Notes in Business Information Processing*. Vol. 329. Springer, 2018, pp. 3–19.

[123] Jorma Rissanen. "Modeling by shortest data description." In: *Automatica* 14.1 (1978), pp. 465–471.

[124] Jorma Rissanen. "A Universal Prior for Integers and Estimation by Minimum Description Length." In: *The Annals of Statistics* 11.2 (1983), pp. 416–431.

[125] Ronald Rivest. "Learning Decision Lists." In: *Machine Learning* 2.3 (1987), pp. 229–246.

[126] Robert Robinson. "Counting labeled acyclic digraphs." In: *New Directions in the Theory of Graphs* (1973), pp. 239–273.

[127] V.I. Rodionov. "On the number of labeled acyclic digraphs." In: *Discrete Mathematics* 105.1 (1992), pp. 319–321.

[128] Cynthia Rudin, Benjamin Letham, Ansaf Salleb-Aouissi, Eugene Kogan, and David Madigan. "Sequential event prediction with association rules." In: *Proceedings of the 24th Annual Conference on Learning Theory (COLT), Budapest, Hungary*. 2011, pp. 615–634.

[129] Thomas L. Saaty. *Elements of Queueing Theory: With Applications*. McGraw-Hill Book Company, 1961.

[130] Ashwin Sah, Mehtaab Sawhney, David Stoner, and Yufei Zhao. "The number of independent sets in an irregular graph." In: *Journal of Combinatorial Theory, Series B* 138 (2019), pp. 172–195.

[131] Denise Maria Vecino Sato, Sheila Cristiana de Freitas, Jean Paul Barddal, and Edson Emilio Scalabrin. "A survey on concept drift in process mining." In: *ACM Computing Surveys* 54.9 (2021), pp. 1–38.

[132] Stefan Schönig, Claudio Di Ciccio, Fabrizio M. Maggi, and Jan Mendling. "Discovery of multi-perspective declarative process models." In: *Proceedings of the 14th International Conference on Service-Oriented Computing (ICSOC) Banff, Canada*. 2016, pp. 87–103.

[133] Günther Schuh, Andreas Gützlaff, Sven Cremer, Seth Schmitz, and Arian Ayati. "A Data Model to Apply Process Mining in End-to-End Order Processing Processes of Manufacturing Companies." In: *Proceedings of the 2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), Virtual Event*. 2020, pp. 151–155.

[134] Marco Bjarne Schuster, Boris Wiegand, and Jilles Vreeken. *Data is Moody: Discovering Data Modification Rules from Process Event Logs*. 2023. arXiv: 2312.14571.

[135] José Á Segura-Muros, Juan Fernández-Olivares, and Raúl Pérez. *Learning Numerical Action Models from Noisy Input Data*. 2021. arXiv: 2111.04997.

[136] José Á Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. "Discovering relational and numerical expressions from plan traces for learning action models." In: *Applied Intelligence* 51.11 (2021), pp. 7973–7989.

[137] Arik Senderovich, J. Christopher Beck, Avigdor Gal, and Matthias Weidlich. "Congestion graphs for automated time predictions." In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI), Honolulu, HI*. 2019, pp. 4854–4861.

[138] Arik Senderovich, Sander J. J. Leemans, Shahar Harel, Avigdor Gal, Avishai Mandelbaum, and Wil van der Aalst. "Discovering queues from event logs with varying levels of information." In: *Proceedings of the 14th International Conference on Business Process Management (BPM), Rio de Janeiro, Brazil*. 2016, pp. 154–166.

[139] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. "Queue mining – predicting delays in service processes." In: *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE), Thessaloniki, Greece*. 2014, pp. 42–57.

[140] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. "GANAK: A Scalable Probabilistic Exact Model Counter." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Macao, China*. 2019, pp. 1169–1176.

[141] Dominique Sommers, Vlado Menkovski, and Dirk Fahland. "Process discovery using graph neural networks." In: *Proceedings of the 3rd International Conference on Process Mining (ICPM), Eindhoven, The Netherlands*. 2021, pp. 40–47.

[142] Mate Soos and Kuldeep S. Meel. "BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting." In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI), Honolulu, HI*. 2019, pp. 1592–1599.

[143] Fei Tao, Bin Xiao, Qinglin Qi, Jiangfeng Cheng, and Ping Ji. "Digital twin modeling." In: *Journal of Manufacturing Systems* 64 (2022), pp. 372–389.

[144] Nikolaj Tatti. "Significance of Episodes Based on Minimal Windows." In: *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM), Miami, FL*. 2009, pp. 513–522.

[145] Nikolaj Tatti and Boris Cule. "Mining Closed Episodes with Simultaneous Events." In: *Proceedings of the 17th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), San Diego, CA*. 2011, pp. 1172–1180.

[146] Nikolaj Tatti and Boris Cule. "Mining Closed Strict Episodes." In: *Data Mining and Knowledge Discovery* 25 (2012).

[147] Nikolaj Tatti and Jilles Vreeken. "The Long and the Short of It: Summarizing Event Sequences with Serial Episodes." In: *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Beijing, China*. 2012, pp. 462–470.

[148] Niek Tax, Natalia Sidorova, Wil van der Aalst, and Reinder Haakma. "LocalProcessModelDiscovery: Bringing Petri nets to the pattern mining world." In: *Proceedings of the 39th International Conference on Application and Theory of Petri Nets and Concurrency, Bratislava, Slovakia*. 2018, pp. 374–384.

[149] Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. "Predictive business process monitoring with LSTM neural networks." In: *Proceedings of the 29th International Conference on Advanced Information Systems Engineering (CAiSE), Essen, Germany*. 2017, pp. 477–492.

[150] Farbod Taymouri, Marcello La Rosa, and Sarah Erfani. "A Deep Adversarial Model for Suffix and Remaining Time Prediction of Event Sequences." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Virtual Event*. 2021, pp. 522–530.

[151] Dimosthenis C. Tsouros and Kostas Stergiou. "Efficient multiple constraint acquisition." In: *Constraints* 25.3-4 (2020), pp. 180–225.

[152] Leslie G. Valiant. "The Complexity of Enumeration and Reliability Problems." In: *SIAM Journal on Computing* 8.3 (1979), pp. 410–421.

[153] Neil Walkinshaw and Mathew Hall. "Inferring Computational State Machine Models from Program Executions." In: *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC*. 2016.

[154] Neil Walkinshaw, Ramsay Taylor, and John Derrick. "Inferring extended finite state machine models from software executions." In: *Empirical Software Engineering* 21.3 (2016), pp. 811–853.

[155] Pieter Wartenhorst. "N parallel queueing systems with server breakdown and repair." In: *European Journal of Operational Research* 82.2 (1995), pp. 302–322.

[156] A. J. M. M. Weijters and Joel Tiago S. Ribeiro. "Flexible Heuristics Miner (FHM)." In: *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM), Honolulu, HI*. 2011, pp. 310–317.

[157] Jan Martijn E.M. van der Werf, Artem Polyvyanyy, Bart R. van Wensveen, Matthieu Brinkhuis, and Hajo A. Reijers. "All that Glitters Is Not Gold: Towards Process Discovery Techniques with Guarantees." In: *Proceedings of the 33rd International Conference on Advanced Information Systems Engineering (CAiSE), Virtual Event*. 2021, pp. 141–157.

[158] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Mining easily understandable models from complex event logs." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Virtual Event*. 2021, pp. 244–252.

[159] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Discovering Interpretable Data-to-Sequence Generators." In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI), Virtual Event*. 2022, pp. 4237–4244.

[160] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "Why Are We Waiting? Discovering Interpretable Models for Predicting Sojourn and Waiting Times." In: *Proceedings of the SIAM International Conference on Data Mining (SDM), Minneapolis, MN*. 2023, pp. 352–360.

[161] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. "What are the Rules? Discovering Constraints from Data." In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI), Vancouver, Canada*. 2024.

[162] Aileen P. Wright, Adam T. Wright, Allison B. McCoy, and Dean F. Sittig. "The use of sequential pattern mining to predict next prescribed medications." In: *Journal of Biomedical Informatics* 53 (2015), pp. 73–80.

[163] Lincen Yang and Matthijs van Leeuwen. "Truly unordered probabilistic rule sets for multi-class classification." In: *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Grenoble, France*. 2022, pp. 87–103.

[164] Mohammed J. Zaki. "SPADE: An efficient algorithm for mining frequent sequences." In: *Machine Learning* 42.1-2 (2001), pp. 31–60.

[165] Sebastiaan J. van Zelst. "Process mining with streaming data." PhD thesis. TU Eindhoven, 2019.

[166] Junping Zhou, Minghao Yin, and Chunguang Zhou. "New worst-case upper bound for #2-SAT and #3-SAT with the number of clauses as the parameter." In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI), Atlanta, GA*. 2010, pp. 217–222.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`https://bitbucket.org/amiede/classicthesis/`

Final Version as of December 27, 2023 (`classicthesis` version 4.4).