Mining Easily Understandable Models from Complex Event Logs

Boris Wiegand[°•] Dietrich Klakow[•] Jilles Vreeken[*]

Abstract

We consider the problem of discovering accurate, yet easily understandable graph-based models from complex event sequence data. 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. Current approaches do not solve this problem satisfyingly, as their modeling language is too restricted to capture complex behavior or they return models that are still too difficult to understand.

We formalize the problem in terms of the Minimum Description Length (MDL) principle, by which we say that the best model provides the shortest lossless description of the data. The resulting problem is NP-hard, and hence we propose the greedy PROSEQO algorithm to discover good models from data. PROSEQO iteratively simplifies the current description by removing nodes, edges, and applying patterns, until MDL tells us to stop. For whenever this result is still too complex, we propose PROSIMPLE, which iteratively removes further edges until we satisfy a user-specified threshold.

Through an extensive set of experiments, we show both methods perform very well in practice. They return simple models that reconstruct the ground truth well, need only little data to do so, are robust against noise, and scale well. A case study shows that, unlike the state of the art, we discover easily understandable models that capture the key aspects of the data generation process.

1 Introduction

Suppose we are given a database of event sequences. How can we discover high quality yet easily understandable models of the data generating process? As an example, consider the production 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 model of their process, these are idealized and do not necessarily match reality in which human decisions, machine breakdowns, bottlenecks, etc. all have their effects. Having a high-quality model of what is *actually* going on hence does not only allow valuable insight, the chance to optimize, but also answering what-if questions.

Real-world event data exhibits complex behaviors, such as sequences, branches, loops, optionals, and combinations thereof, and hence 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

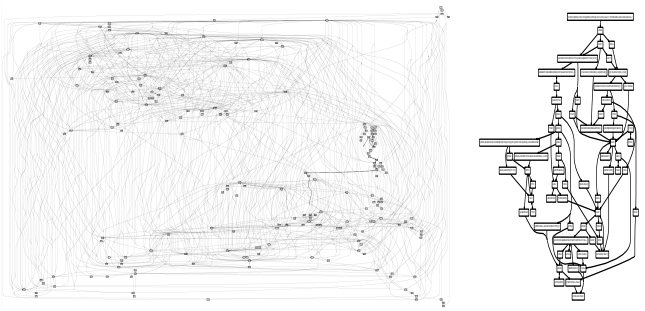


Figure 1: Example on real data. Left, the directly-follows-graph, and right, the result of PROSIMPLE, for the *Rolling Mill* production log of steel producer Dillinger.

Fig. 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. While it shows some structure, the graph is heavily cluttered: our domain experts could barely make out the bigger picture, let alone gain any non-trivial understanding.

Trying to make sense of event data is a classic problem, and is studied by both the pattern mining and process discovery communities. Existing solutions, however, do not quite solve the problem. While pattern mining methods [4, 9, 25] are very good for discovering and summarizing non-trivial behavior, these only return loose collections of local structures, rather than a global model for the data. Process discovery [12, 2] on the other hand produces global models, but these tend to look similarly complex and hard-to-understand as the directly-follows-graph in Fig. 1.

In this paper we combine the best of both worlds. We propose to discover easily understandable models from complex event logs in the form of *pattern graphs*. Simply put, these are directed graphs with 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 that provides 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

[°] SHS - Stahl-Holding-Saar GmbH & Co. KGaA, Dillingen, Germany.
boris.wiegand@stahl-holding-saar.de

[•] Saarland University, Saarbrücken, Germany.
dietrich.klakow@lsv.uni-saarland.de

[*] CISPA Helmholtz Center for Information Security, Germany.
jv@cispa.de

stop. For whenever this result is still too complex, i.e. has too many edges, we additionally 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. We show they reconstruct the ground truth well with little data needed, are robust against various types of noise, and scale well. On real 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 Fig. 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.

The main contributions of this paper 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 resp. discover good and simple models from data,
(d) evaluate via a large set of experiments,
(e) and make all code and data available.

The remainder of the paper is structured as usual.

2 Preliminaries

Before we formalize the problem, we introduce notation and preliminary concepts that we will use throughout the paper.

2.1 Notation We consider databases of *event sequences*. Such a database D consists of $n = |D|$ sequences. A sequence $S \in D$ consists of $m = |S|$ events drawn from a finite length alphabet $\Omega = \{a, b, \ldots\}$. We write $S[i]$ to refer to the i^{th} event in S. We denote the empty string as ϵ.

We will model using directed graphs $G = (V, E)$, where each node corresponds to a pattern. The simplest patterns are *singletons* $X \in \Omega$. 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.

All logarithms are to base 2, and we use $0 \log 0 = 0$.

2.2 MDL The Minimum Description Length principle (MDL) [19, 10] is a practical version of Kolmogorov Complexity [13]. Both embrace the slogan Induction by Compression. We use the MDL principle for model selection.

By MDL, the best model is the model that gives the best lossless compression. More specifically, given a set of models $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimizes $L(M) + L(D \mid M)$, in which $L(M)$ is the length in bits of the description of M, and $L(D \mid M)$ is the length of the data when encoded with model M. Simply put, we are interested in that model that best compresses the data without loss. MDL as described above is known as two-part MDL, or crude MDL; as opposed to refined MDL. In refined MDL model and data are encoded together [10]. We use two-part MDL because we are specifically interested in the model: that pattern graph that best describes the data. In MDL we are only concerned with code lengths, not actual code words.

Next, we formalize our problem in terms of MDL.

3 MDL for Pattern Graphs

As models we consider *pattern graphs*. A pattern graph is a directed and possibly cyclic graph $G = (V, E)$ where the nodes correspond to patterns. In addition, a valid pattern graph has two special nodes, the empty string ϵ and an end-of-sequence character $\hookleftarrow$ that resp. serve as source v_s and sink v_e. As we are specifically interested in easily understandable models M, 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 $S \in \Omega^m$ 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, which choices to make, etc, 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 S as well as the covers of it of three different models M_1, M_2, M_3 in Fig. 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 should 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 there are two edges we can follow: we 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 S without loss.

In the second example, we start again at v_s, emit ϵ after reading $\boxed{\rightarrow}$, and arrive at a *sequence* pattern. We emit its first element (a) after reading the next $\boxed{\rightarrow}$, and arrive at its second element ($b|c$). To decide whether to emit b, or c, we

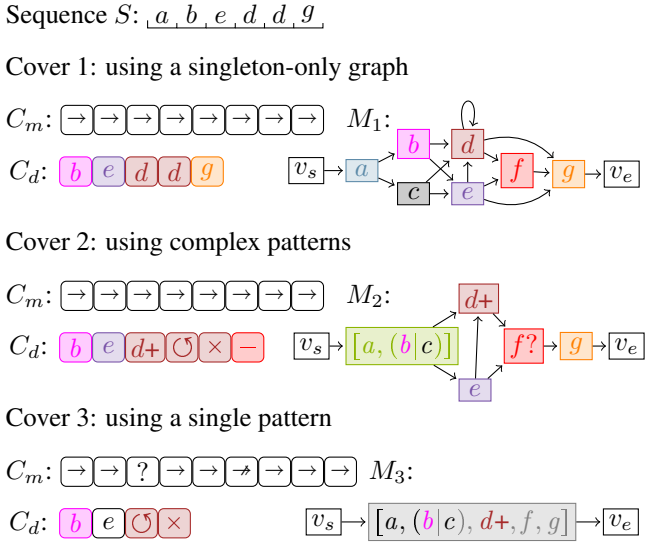


Figure 2: Toy example. Sequence S as covered by three different models M_1, M_2, M_3.

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 have to 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 or not 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 happens to be 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 should move to the next element but not emit. The final → takes care of g, and we have again losslessly decoded S.

3.1 Defining the Score The above examples illustrate how we can model event sequences S 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.

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 using the MDL-optimal encoding for integers $z \geqslant 1$ [20], and then proceed to encode the code streams C_m and C_d. $L_{\mathbb{N}}$ 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.

According to Shannon Entropy, the length in bits of the optimal prefix-free code for an event x is $-\log P(x)$, which follows the intuition that the more frequent an event the shorter its code should be. However, this requires knowledge of the distribution of events beforehand.

To avoid any arbitrary choices in the model encoding, we use prequential codes [10] to encode the model and disambiguation streams. Prequential codes 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 [5].

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 $\Omega_m = \{\rightarrow, \nrightarrow, ?\}$, we have

$$L(C_m) = -\sum_{i=1}^{|C_m|} \log \frac{usg_i(C_m[i] \mid S[j]) + \epsilon}{\sum usg_i(\cdot \mid S[j]) + \epsilon} \ ,$$

where we encode whether we have to emit, skip, or fill a gap, conditioned on what event $S[j] \in \Omega$ we encoded right before message $C_m[i]$. Initializing with standard choice $\epsilon = 0.5$, and $usg_0(\cdot|\cdot) = 0$, 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 messages in C_p correspond to nodes in the model, and which nodes $v \in G$ are possible depends on that node v_k we are currently at. That is, we have

$$L(C_p) = -\sum_{i=1}^{|C_p|} \log \frac{usg_i(C_p[i] \mid v_k) + \epsilon}{\sum usg_i(\cdot \mid v_k) + \epsilon} \ .$$

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

$$L(C_g) = -\sum_{i=1}^{|C_g|} \log \frac{usg_i(C_g[i] \mid v_k) + \epsilon}{\sum usg_i(\cdot \mid v_k) + \epsilon} \ .$$

Finally, the messages in C_s are dependent both on the last decoded symbol $S[j]$ and what node v_k we are at, i.e.

$$L(C_s) = -\sum_{i=1}^{|C_s|} \log \frac{usg_i(C_s[i] \mid S[j], v_k) + \epsilon}{\sum usg_i(\cdot \mid S[j], v_k) + \epsilon} \ .$$

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

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

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 ($\mathcal{T} = \{singleton, sequence, choice, loop, optional\}$) 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 [13], which is an index over a canonically ordered set of all directed graphs of $|V|$ nodes and $|E|$ edges. The only further details needed to specify 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.

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

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

3.2 Formal Problem Formulation

With the above definitions, we can now formally define the problem at hand.

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, for a given model there exist exponentially many covers, and finding the optimal cover is equivalent to aligning Petri nets and sequences which has known to be NP-hard [6].

Hence, we resort to heuristics.

Algorithm 1: COVER

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

```
v ← v_s;
foreach event e ∈ S do
    if pattern v can cover e then
        add codes to C to encode e with v;
    else if pattern u ∈ M that can cover s and there
      exists a path P from v to u then
        add codes to C to skip remainder of v;
        add codes to C to skip all p ∈ P up to u;
        add codes to C to encode e with u;
        v ← u;
    else
        add codes to C to encode e as [?];
return C
```

4 Algorithm

To find good solutions to the Minimal Pattern Graph Problem in practice, we split the problem into two, and propose greedy algorithms to resp. 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.

4.1 Computation of a Good Cover

To compute $L(D \mid M)$ we need a good cover C. As computing the optimal cover is NP-hard, we take a greedy approach. The intuition is that 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 S with patterns in the model, by which we ensure a linear runtime w.r.t. to $|S|$.

We give the pseudo-code as Algorithm 1. In a nutshell, whenever there exists a pattern $u \in M$ that can cover the next event $e \in S$, 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, the shortest path problem reduces to a breadth-first-search with runtime complexity $O(|V| + |E|)$. The maximal number of nodes in the graph is bounded by $|\Omega|$. Therefore, covering all sequences in D has runtime complexity $O(n \cdot m \cdot (|\Omega| + |E|))$. The cover encoding using prequential codes is order-invariant, hence, the cover algorithm is also sequence-order-invariant.

4.2 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 pseudo-code as Algorithm 2.

Algorithm 2: PROSEQO

input : sequence database D
output: model M for D

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

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 $e^* \leftarrow \arg\min_{e \in E} L(D, M \ominus e)$;
4 $M \leftarrow M \ominus e^*$;
5 $M \leftarrow$ PROSEQO(D, M) ;
6 **return** M;

We start from the directly-follows-graph (line 1), an overfit pattern graph where all singleton events are nodes, i.e. $V = \Omega$, and we have edges $(v, u) \in E$ whenever in any sequences in D, u happens right after v. We then 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 gain (ln. 4–5), and update model and candidate transformations whenever we manage to improve over the current model (ln. 6–7).

As 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'. Every edge $(a, b) \in E$ generates a candidate sequence $[a, b]$. Nodes with the same predecessor v generate choice candidates, loop candidates are created from loops in the pattern 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. To gain efficiency, we do not re-generate the entire candidate set A in every iteration, but rather update it: we remove those transformations from A 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 the 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 $O\left(n \cdot m \cdot (|\Omega| + |E|)^2 \cdot |\Omega|\right)$.

4.3 Discovering Simple Models with PROSIMPLE

What if the MDL-optimal model, or its approximation by PROSEQO, is still too complex for a human? 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 pseudo-code as Algorithm 3. First, we simply run PROSEQO on D, which gives us a $M \in \mathcal{M}$ (line 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). 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 $M \in \mathcal{M}^{(r)}$ (line 5). We refer to this method as PROSIMPLE.

The edge removal part of PROSIMPLE has runtime complexity $O(|E|^2 \cdot n \cdot m \cdot (|\Omega| + |E|))$, because we can remove up to $|E|$ edges, and for each iteration we have to compute the cover to get the score for $|E|$ different models.

5 Related Work

Discovering structure from event sequences is a classic research topic [1, 14]. Earlier proposals focused on efficient discovery of all frequent subsequences with or without gaps [28, 16], resulting in overly many and highly redundant patterns: the pattern explosion. Attention hence shifted to reducing redundancy via closures [23, 22], statistical testing [21, 17], or a pattern set mining approach [24, 8].

Subsequences and serial episodes can model interesting behavior, but only have limited expressive power. There exist proposals that can additionally model parallel behavior [23], choices [4], or periodicity [9], but each only extends in one direction. Petri nets [18] can jointly model loopy, alternative as well as concurrent behavior, but existing approaches [25] 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 [26], 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 [27, 12, 2]. 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 overgeneralize and obtain only low precision [2, 26].

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.

6 Experiments

In this section, we empirically evaluate PROSEQO and PROSIMPLE on synthetic and real-world data. We implemented both methods in Python. For reproducibility, we make both code and synthetic data generators available for research purposes.[1] All experiments were executed 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 [2] and IMF [12] are process discovery algorithms, whereas SQUISH [4] 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 [12, 2].

6.1 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 fixed start and end symbols. 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 also outputs the sequential pattern $[\text{START}, a_1]$. While IMF correctly identifies that all 10 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.

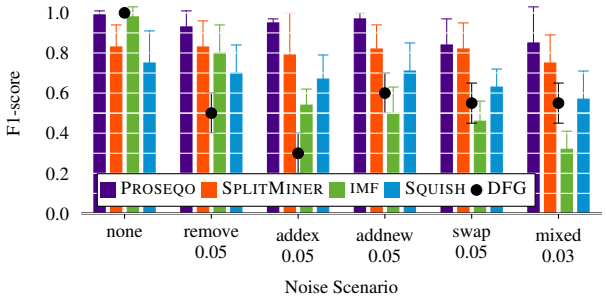


Figure 3: [**PROSEQO can handle noise**] F1-scores on directly-follows-relations for resp. no, 5% remove, 5% addex, 5% addnew, 5% swap, and 3% of all noise types simultaneously, for PROSEQO, SPLITMINER, IMF, SQUISH, and the trivial (DFG) graph that PROSEQO departs from.

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. [11] 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 random 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 neighbouring pair of events, we swap their order with a given probability.

As a metric of success, we consider the F1 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 Fig. 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

[1] https://eda.mmci.uni-saarland.de/proseqo/

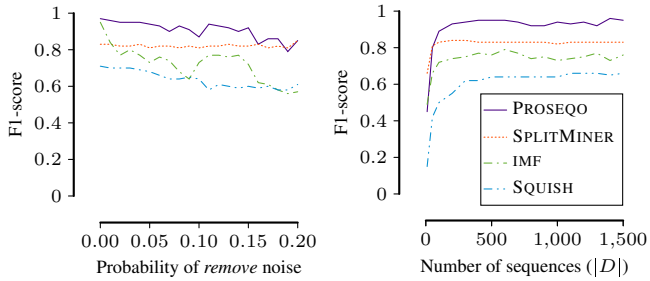


Figure 4: [**PROSEQO is robust**] F1-scores for varying amounts of *remove* noise (left) and for varying number of sequences with 5% remove noise (right).

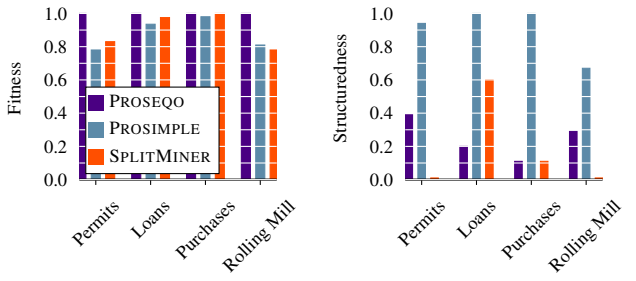


Figure 5: [**PROSIMPLE mines well-fitting yet understandable models**] Fitness (left, higher is better) and structuredness (right, higher is better) for PROSEQO, PROSIMPLE with $r = 1.5$ and SPLITMINER on four real-world datasets.

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 investigate robustness against varying levels of *remove* noise, and for varying number of samples $S \in D$ for a fixed amount of *remove* noise of 5%. We again generate 20 models per setting and report the average result in Fig. 4. We see that PROSEQO performs favorably, its F1 scores only dropping slightly for up to 20% noise, whereas it only needs a 100 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 these two methods in the remainder of this section.

6.2 Real-World Data Next, we evaluate on four real-world event logs. Three stem from the publically available Business Process Intelligence Challenge. *Permits* contains event data for building permit applications of a Dutch municipality. *Loans* corresponds to the recording of a loan application process of a Dutch financial institute. *Purchases* 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 1.

Because we do not know the ground truth, we cannot compute F1 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 $S = 1$.

We measure fitness by converting the models to Petri nets [18] and computing the corresponding score [3]. 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. We compare to SPLITMINER. We run each of the methods, and give the results in Fig. 5 and Table 1.

We first consider Fig. 5 and see that 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 1, we again see that PROSIMPLE creates by far the simplest models. The loss of fit is revealed by the relative compression $L\%$. A much simpler model cannot explain all the behavior in the data and thus $L(D \mid M)$ increases a lot. Even though the fit of the data might look bad for PROSIMPLE, it is an approximation of the best fit of a model with degree ratio $r = 1.5$. We perform a parameter-sensitivity analysis in the supplementary.[2]

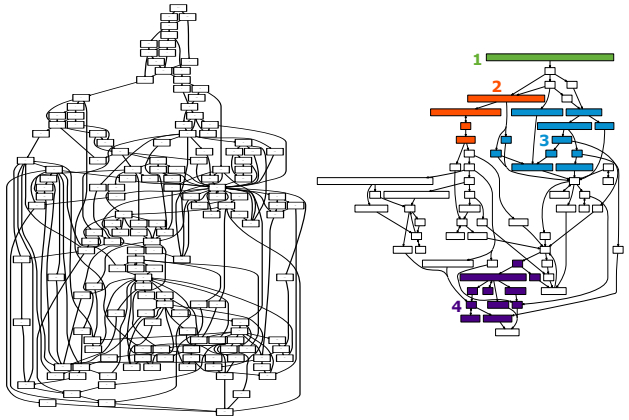
The importance of model simplicity becomes even more clear when we visually inspect the models. In Fig. 6 we show the models discovered by PROSIMPLE and SPLITMINER for the *Rolling Mill* data. While the latter is already much more structured than the trivial directly-follows graph, it is still a bowl 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.

6.3 Case Study Using the PROSIMPLE model in Fig. 6, we can highlight four parts of the Dillinger rolling mill that are understandable to everyone. The process starts with so-

[2] https://eda.mmci.uni-saarland.de/proseqo/

				SplitMiner			ProSeqo				ProSimple, $r = 1.5$			
Data	n	$\hat{m}$	$\lvert\Omega\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	t	$\lvert V\rvert$	$\lvert E\rvert$	$L\%$	t	$\lvert V\rvert$	$\lvert E\rvert$	$L\%$	t
Permits	1199	46	291	683	1441	33s	181	33071	98.2	50h	42	49	129.7	51h
Loans	31509	17	26	53	71	1s	22	135	98.6	2h25m	5	5	130.4	2h28m
Purchases	2000	9	28	68	135	4s	26	163	97.4	4m	6	6	103.3	4m
Rolling Mill	1000	28	191	427	817	9s	135	934	98.3	2h12m	68	101	132.7	2h30m

Table 1: [**Results on real-world data**] We give the base statistics, number of sequences $n = |D|$, number of unique events $|\Omega|$, and average sequence length $\hat{m}$, and results of SplitMiner, ProSeqo, and ProSimple with $r = 1.5$ on four real world datasets. Given are the number of nodes ($|V|$) and edges ($|E|$) of the discovered models. For ProSeqo and ProSimple we additionally give the relative compression ($L\%$, lower is better) and runtime.



(a) SplitMiner (b) ProSimple

Figure 6: [**ProSimple mines detailed yet easily understandable models**] Result of SplitMiner (left) and ProSimple (with $r = 1.5$, right) on the *Rolling Mill* data.

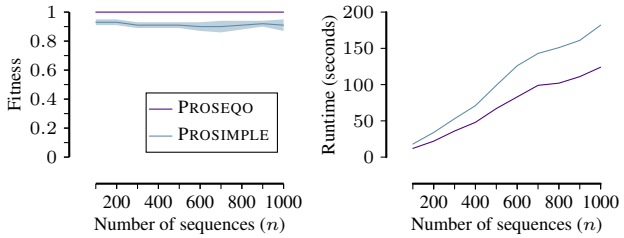


Figure 7: [**ProSeqo and ProSimple scale favourably**] Scalability of ProSeqo and ProSimple on *Loans* in terms of average fitness over 10 runs with standard deviation (left) and runtime in seconds (right) for varying subsample size.

called *slabs*, cast steel cuboids. The green source node (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 (4) mainly consists of quality checks and corrections.

Not only does the model ProSimple discovered provide a high-level overview of the control-flow of the rolling mill process, it also contains details on low-level behavior that allow for, among others, an anomaly analysis. First, our domain experts questioned some of the modeled behavior as violating their expectation. These all turned out to be due to relatively rare but re-occurring deviations from the normal behavior such as e.g. machine failures, as well as alternative routing due to high workload in parts of the plant. For example, if needs be, thicker plates can 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 product quality and reduced production loss.

6.4 Scalability Finally, we report in Figure 7 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 they scale linearly in the number of sequences.

7 Discussion

Although both methods 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. On the topic of scalability, it is worthwhile to investigate whether it is possible to formalize accurate and easy-to-compute estimates [7], such that we can be more informed during the search.

We see many applications of pattern graphs, including anomaly detection, 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 re-done in pure counterfactual terms [15].

8 Conclusion

We studied the problem of discovering accurate yet easily understandable models from complex event sequence data. 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 parameter 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.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, pages 3–14. IEEE, 1995.

[2] A. Augusto, R. Conforti, M. Dumas, and M. La Rosa. Split Miner: Discovering accurate and simple business process models from event logs. In *ICDM*, pages 1–10. IEEE, 2017.

[3] A. Berti and W. van der Aalst. Reviving token-based replay: Increasing speed while improving diagnostics. In *ATAED*, pages 87–103, 2019.

[4] A. Bhattacharyya and J. Vreeken. Efficiently summarising event sequences with rich interleaving patterns. In *SDM*, pages 795–803. SIAM, 2017.

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

[6] M. de Leoni and W. van der Aalst. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *BPM*, pages 113–129. Springer, 2013.

[7] J. Fischer and J. Vreeken. Sets of robust rules, and how to find them. In *ECML PKDD*, pages 38–54. Springer, 2019.

[8] J. Fowkes and C. Sutton. A subsequence interleaving model for sequential pattern mining. In *KDD*, pages 835–844, 2016.

[9] E. Galbrun, P. Cellier, N. Tatti, A. Termier, and B. Crémilleux. Mining periodic patterns with a MDL criterion. In *ECML PKDD*, pages 535–551. Springer, 2018.

[10] P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[11] T. Jouck and B. Depaire. PTandLogGenerator: A generator for artificial event data. In *BPM Demo Track*, pages 23–27. CEUR, 2016.

[12] S. Leemans, D. Fahland, and W. van der Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. In *BPM-Workshops*, pages 66–78. Springer, 2013.

[13] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.

[14] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *KDD*, pages 210–215. AAAI, 1995.

[15] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2nd edition, 2009.

[16] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE TKDE*, 16(11):1424–1440, 2004.

[17] F. Petitjean, T. Li, N. Tatti, and G. Webb. Skopus: Mining top-k sequential patterns under leverage. *Data Min. Knowl. Disc.*, 30, 2016.

[18] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik Bonn, 1962.

[19] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(1):465–471, 1978.

[20] J. Rissanen. A universal prior for integers and estimation by minimum description length. *Annals Stat.*, 11(2):416–431, 1983.

[21] N. Tatti. Significance of episodes based on minimal windows. In *ICDM*, pages 513–522, 2009.

[22] N. Tatti and B. Cule. Mining closed episodes with simultaneous events. In *KDD*, pages 1172–1180, 2011.

[23] N. Tatti and B. Cule. Mining closed strict episodes. *Data Min. Knowl. Disc.*, 2011.

[24] N. Tatti and J. Vreeken. The long and the short of it: Summarizing event sequences with serial episodes. In *KDD*, pages 462–470. ACM, 2012.

[25] N. Tax, N. Sidorova, W. van der Aalst, and R. Haakma. Localprocessmodeldiscovery: Bringing petri nets to the pattern mining world. In *ATPN*, pages 374–384. Springer, 2018.

[26] W. van der Aalst. *Process Mining – Data Science in Action*. Springer, second edition, 2016.

[27] A. Weijters and J. Ribeiro. Flexible heuristics miner (FHM). In *CIDM*, pages 310–317. IEEE, 2011.

[28] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Mach. Learn.*, 42(1-2):31–60, 2001.

A Appendix

Here, we include supplementary material which could not be part of our main paper.

A.1 Parameter Sensitivity of PROSIMPLE In Fig. 8, we show the influence of the degree ratio parameter r on the result of PROSIMPLE. The lower r, the more edges are removed from the output of PROSEQO. Hence, a lower value of r leads to a more structured, easier understandable model but a lower fit on the data. For the *Rolling Mill* data, $r = 7.0$ does not remove any edges and thus outputs a model with the same fitness and structuredness as PROSEQO alone. A value of $r = 1.5$ trades relatively little fitness for a lot of model simplicity (structure).

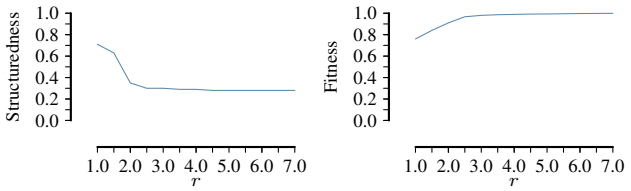


Figure 8: [**Varying degree ratio on PROSIMPLE**] Structuredness (left) and Fitness (right) for PROSIMPLE with varying degree ratio r on the *Rolling Mill* dataset. $r = 7.0$ is equivalent to PROSEQO.