Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns

Joscha Cüppers
joscha.cueppers@cispa.de
CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany

Jilles Vreeken
jv@cispa.de
CISPA Helmholtz Center for Information Security,
Saarbrücken, Germany

ABSTRACT

We study the problem of succinctly summarizing a database of event sequences in terms of *generalized* sequential patterns. That is, we are interested in patterns that are not exclusively defined over observed surface-level events, as is usual, but rather may additionally include *generalized* events that can match a set of events. To avoid spurious and redundant results we define the problem in terms of the Minimum Description Length principle, by which we are after that set of patterns and generalizations that together best compress the data without loss. The resulting optimization problem does not lend itself for exact search, which is why we propose the heuristic Flock algorithm to efficiently find high-quality models in practice. Extensive experiments on synthetic and real-world data show that Flock results in compact and easily interpretable models that accurately recover the ground truth, including rare instances of generalized patterns. Additionally Flock recovers how generalized events within patterns depend on each other, and overall provides clearer insight into the data-generating process than using state of the art algorithms that only consider surface-level patterns.

CCS CONCEPTS

• **Information systems → Data mining**.

KEYWORDS

generalized patterns, pattern set mining, event sequences

ACM Reference Format:
Joscha Cüppers and Jilles Vreeken. 2023. Below the Surface: Summarizing Event Sequences with Generalized Sequential Patterns. In *Proceedings of ACM SIGKDD conference (KDD'23).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Succinctly summarizing a database in easily understandable terms is one of the key problems in data mining. Pattern set mining, where we mine a small sets of patterns that together model the data well, has proven to be particularly successful [9, 30, 33]. Existing methods, however, only consider what we call surface-level patterns. These are patterns that are exclusively defined over observed events, and therewith also only match exact instances in the data.

To illustrate the limitations of surface-level patterns, let us consider a toy example. The two sentences *'the cat meows'* and *'the dog barks'* share only the event *'the'*. Any method that only considers surface-level events would either just report 'the' as a common pattern, or, if they occur frequently often enough in the data report both sentences as patterns, neither of which is particularly useful. In contrast, any human would immediately see that these sentences are both instances of the general statement *'the [pet] [makes noise]'*, and would be annoyed to get a summary that would both explicitly report all variants of this general pattern (e.g. mice squeaking, horses whinnying) as well as fail to report rare instances (e.g. fishes saying blub). For natural language, there exist high-quality word ontologies that we can use to analyse text through a more general lens [1, 12]. However, for event sequence data in general, this is not the case. This raises the question, how can we automatically discover a set of patterns that succinctly describes the data in terms of more general patterns?

In this paper we consider the problem of discovering generalized events and generalized patterns from event sequence data. A generalized event is a symbol that can match different observed events e.g. $\alpha = \{a, b\}$ matches a and b. A generalized pattern is a sequential pattern that is defined over observed and generalized events, e.g. pattern c, α, d matches c, a, d and c, b, d. This more expressive pattern language allows us not only to more effectively summarize event sequence data, but also provide deeper insight as it is less prone to under or over-fitting as compared to a pattern language of surface-level patterns. In this context underfitting means that patterns are either not reported or only partially, overfitting means semantically identical patterns are reported multiple times.

We define the problem of discovering the best set of generalizations and patterns in terms of the Minimum Description Length principle [13]. Loosely speaking, we are after those that together provide the best lossless compression. The search space for this problem is vast, triply-exponential, and is not favourably structured, which is why we propose the Flock algorithm to heuristically mine good models from data. Flock finds high-quality generalizations by considering those events that frequently appear in the same (pattern) context, and finds high-quality generalized patterns by iteratively merging patterns and extending them with discovered generalizations.

Flock aside, very few methods consider sequential patterns beyond surface level patterns, and either require a beforehand known structure [1, 12, 27] or can only model 'generalizations' that are limited to a single location in a pattern [3]. As we will see, methods that only consider surface-level patterns are prone to highly redundant results – after all, they cannot generalize and are hence bound to report every sufficiently frequent variation of a true generating pattern – but also to underfitting, because they only report sufficiently frequent instances rather than the more rare but important variants.

Through an extensive set of experiments and comparisons to a wide array of competitors, we show that our method works well in practice. On synthetic data, we show that Flock recovers surface-level patterns as well as the state of the art, but that it outperforms these competitors by a large margin in recovering generalized patterns and generalizations. On real-world data, we show that the small sets of highly expressive patterns that Flock discovers provide clear insight into the data-generating process that goes far beyond what surface-level patterns can provide.

2 PRELIMINARIES

In this section, we discuss preliminaries and introduce the notation we use throughout the paper.

2.1 Notation

We consider a database D of $|D|$ event sequences. An event sequence $S \in D$ consists of $|S|$ events drawn from an alphabet Ω_o of *observed* events $e \in \Omega_o$. We write $S[j]$ to refer to the j^{th} event in S and $S[j:k]$ to mean a subsequence $S[j]\dots S[k]$. Note, we do not allow multiple events to occur at time point j.

In addition to observed events $e \in \Omega_o$, we also consider *generalized* events $\alpha \in \Omega_g$. Generalized events are special in that they match multiple observed events $e \in \Omega_o$, e.g. $\alpha = \{a, b\}$ will match either a or b. We allow generalizations to be nested, e.g. $\beta = \{\alpha, c\}$ will match any out of a, b, or c. We can flatten a generalized event, $fl(\alpha)$, to obtain all observed events that α can match.

As patterns we consider serial episodes. A serial episode $p \in \Omega^{|p|}$ is a sequence of $|p|$ events over an alphabet $\Omega = \Omega_o \cup \Omega_g$. We say that a sequence S *contains* an instance of a pattern p if there exists a window $S[j:k]$ that matches p. We explicitly allow gaps between the events in p. To avoid spurious matches we consider windows up to a length of $|p| + |p|n$, where n is a user-chosen parameter. The *support* of a pattern in D is the number of unique matches of p, note that one pattern can match multiple times per sequence.

During search we iteratively refine the generalized alphabet Ω_g by adding and removing (events from) generalizations $\alpha \in \Omega_g$. We write $(\alpha, R, \oplus)$ to denote that refinement of Ω_g where we add event set $R \subset \Omega$ to existing generalization $\alpha \in \Omega_g$, or adding a new generalization $\alpha = R$ to Ω_g if $\alpha \notin \Omega_g$. Analogously, we write $(\alpha, R, \ominus)$ whenever we want to remove (events from a) generalization α. Wherever clear from context we do not write the $\oplus$ and $\ominus$. To denote a set of additive refinements for Ω_g we write $\Omega_g^{\oplus}$, and analog $\Omega_g^{\ominus}$ to denote a set of removal refinements.

2.2 Minimum Description Length

The Minimum Description Length (MDL) principle [13] is a computable and statistically well-founded model selection criterium based on Kolmogorov Complexity [17]. For a given model class $\mathcal{M}$, it identifies the best model $M \in \mathcal{M}$ as the one that minimizes the number of bits needed to describe both model and data without loss, or formally, $L(M) + L(D|M)$ with $L(M)$ the length of model M and $L(D|M)$ the length of data D given M. This is known as two-part, or crude MDL—in contrast to one-part, or refined MDL [13], which

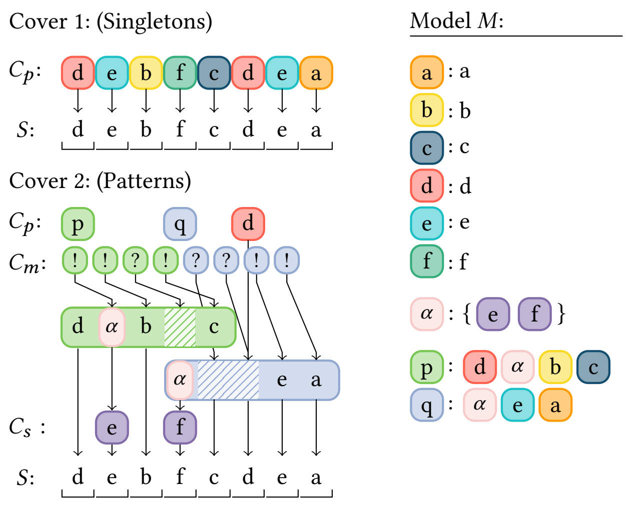

Figure 1: Toy example showing two ways to encode the same sequence S. Cover 1 uses only singletons, while Cover 2 uses the entire model M. A cover C consists of (up to) three different code streams: C_p contains codes for patterns, C_m defines how these interleave, and C_s specifies which observed events $e \in \Omega_o$ the generalized events $\alpha \in \Omega_g$ in the cover map to.

is not computable for arbitrary models. We use two-part MDL because we are particularly interested in the model: the patterns and generalizations. In MDL we are never concerned with materialized codes, we only care about code lengths. To use MDL we have to define a model class $\mathcal{M}$, and encodings for data and model, which we present next.

3 MDL FOR GENERALIZED SEQUENTIAL PATTERNS

We will now define the problem we aim to solve. As model class $\mathcal{M}$ for a dataset D over observed alphabet Ω_o, we consider tuples that define a generalized alphabet Ω_g, and a set of patterns P over $\Omega = \Omega_o \cup \Omega_g$. To ensure that every model $M \in \mathcal{M}$ can validly encode D we require P to always include all singleton patterns, i.e. $P \supseteq \Omega_o$. By MDL, we are interested in that $M \in \mathcal{M}$ that most succinctly describes D without loss.

3.1 Decoding a Sequence

Before we define how we *encode* a dataset given a model M, we first give the intuition on its main components by explaining how to *decode* an already encoded sequence S. We give an example in Fig. 1. In Cover 1, S has been encoded using singleton patterns only. To decode it, we simply iteratively read pattern codes from the pattern stream C_p, and use model M to decode these to the correct events.

Cover 2 utilizes model M better. We again iteratively read codes from C_p. The first code is for pattern p, and we can immediately append its first event (a 'd') to the decoded sequence. To determine whether there is a gap or not, we read a code from the meta stream C_m. This happens to be a fill-code $\boxed{!}$, meaning we can write the next event of p. This is the generalized event α that can match either e or f. To determine which of these two events we have to emit, we read a code from the specification stream C_s, and proceed accordingly. We then continue as before, reading another fill code, and writing a 'b'. Next, we read a gap-code $\boxed{?}$ from the meta-stream, which informs us that there is a gap in pattern p. To fill this gap, we have to read the next code from the pattern stream. We read the code for pattern q, and hence write its first event to the sequence. We now have two patterns that could emit the next event. We therefore read as many meta codes as there are active patterns. If all of these are gap codes, we read from the pattern stream, and otherwise we emit the next event for that pattern for which we read a fill code. Here, the latter is the case for p, we write the corresponding 'c', and are finished decoding p. To wrap things up, we read the next meta-code for q, which is a gap that we fill according to the next pattern code ('d') and finally read two fill codes for q and hence emit 'e' and 'a', after which we have decoded S without loss.

3.2 Calculating the Encoded Length

Now that we know what we need to encode, we define how many bits these codes should cost.

Encoding the data. We start by defining how to compute the encoded cost of a database D given a model M. Formally, we have

$$L(D|M) = L_{\mathbb{N}}(|D|) + \left(\sum_{S_i \in D} L_{\mathbb{N}}(|S_i|) \right) + L(C_p) + L(C_m) + L(C_s) \quad . \tag{1}$$

We first encode the number of sequences, and then the length of each sequence in the database. We then encode the pattern stream C_p, meta stream C_m, and specification stream C_s. We encode the number and length of the sequences using $L_{\mathbb{N}}$, the MDL-optimal encoding for integers $z \geq 1$ [26]. It is defined as $L_{\mathbb{N}}(z) = \log^* z + \log c_0$ where $\log^* z$ is the expansion $\log z + \log\log z + \cdots$ where we only include the positive terms. To ensure a valid lossless encoding, i.e. one that satisfies the Kraft inequality, we set $c_0 = 2.865064$ [26].

We next discuss the three code streams. We start with the pattern stream C_p, Eq. (2). Because the occurrences of pattern codes in the pattern stream are independent, we encode these using optimal prefix codes. Formally, we have

$$L(C_p) = -\sum_{p \in M} usg(p) \log \left(\frac{usg(p)}{\sum_{p' \in M} usg(p')} \right) \quad , \tag{2}$$

where $usg(p)$ is the number of times the code for p appears in pattern stream C_p. To use optimal prefix codes we will have to explicitly encode the usages in the model.

In contrast, the occurrences of codes in the meta stream C_m, Eq. (3), are dependent on which patterns we are currently decoding, meaning we need to know (many) conditional probabilities. To avoid having to make arbitrary choices on how to explicitly encode these in the model, we propose to use prequential codes [13]. Prequential codes work by assuming an initial usage of $\epsilon = 0.5$ [13] for all possible codes, and updating these counts with every transmitted (resp. received) code. This way we not only ensure that we always have a valid coding distribution, but also achieve asymptotic optimality *without* having to transmit the counts beforehand [13]. Formally, we have

$$L(C_m) = \sum_{p \in P} \left(-\sum_{i=1}^{fills(p)} \log \left(\frac{\epsilon + i}{2\epsilon + i} \right) - \sum_{i=1}^{gaps(p)} \log \left(\frac{\epsilon + i}{2\epsilon + fills(p) + i} \right) \right), \tag{3}$$

where $fills(p)$ and $gaps(p)$ refers to the number of fills resp. gaps of pattern p in meta-stream C_m.

This leaves the encoding of the specification stream C_s, Eq. (4). Because specification codes depend on the context of the generalization at hand, we will again use prequential codes. Generalizations *within* a pattern p, however, can additionally be dependent on *each other.* for example, cats meow, dogs bark. To exploit and reveal such structure, we allow for dependencies between generalizations within a pattern. We provide the details in the model encoding below. For now, what matters is that we encode the specification code for an event $e \in fl(\alpha)$ for the current generalization α of pattern p conditioned on an earlier emitted event d of p. Formally, the length in bits of the entire stream is

$$L(C_s) = \sum_{p \in P} \sum_{\alpha \in p} \sum_{i=1}^{usg(p)} -\log \left(\frac{\epsilon + usg_i(e|d)}{|fl(\alpha)|\epsilon + \sum_{c \in fl(\alpha)} usg_i(c|d)} \right) \quad , \tag{4}$$

where for each pattern $p \in P$ (first sum), and each generalization $\alpha \in p$ (second sum), we encode the surface-event e conditioned on the value of event d using prequential codes (third sum). Note that if a generalization $\alpha \in p$ is *not* dependent on an earlier emitted generalization $\beta \in p$, d will be a fixed constant by which the above becomes a standard unconditional prequential code.

Encoding the model. Next, we define how to compute the encoded cost of a model. We start by defining the encoded cost for the generalized alphabet Ω_g. We have

$$\begin{aligned} L(\Omega_g) = {} & L_{\mathbb{N}}(|\Omega_o|) + L_{\mathbb{N}}(|\Omega_g| + 1) + \\ & \sum_{k=1}^{|\Omega_g|} \left(\log k + \log \binom{k-1}{l} + \log(|\Omega_o'|) + \log \binom{|\Omega_o'|}{m} \right) \quad , \end{aligned} \tag{5}$$

where we first encode the sizes of the observed[1] resp. generalized alphabets using $L_{\mathbb{N}}$. We then encode the generalizations $\alpha \in \Omega_g$ in turn. For each generalization $\alpha_k \in \Omega_g$, we first transmit how many nested generalizations it includes, denoted as l, and then identify which these are using a data-to-model code over the $k-1$ generalizations transmitted so far. We then transmit the number of observed events, denoted as m, in Ω_o that α includes, which are not already defined by its nested generalizations. Once we know this number, encode which events out of Ω_o' these are, where Ω_o' is the set of observed events excluding events already defined by its generalizations, formally $\Omega_o' = \Omega_o \setminus \bigcup_{\beta \in \alpha_k} fl(\beta)$.

Given the generalizations, we can next encode the pattern set P and their respective usages, i.e. the code table, Eq. (6). We first transmit the number $|P'|$ of non-singleton patterns $P' \subset P$, and then the *combined* usage of all patterns. We finally encode each

[1]Note that as the size of Ω_o is constant for any model of the same data, it is unnecessary to include the first term for optimization, but we include it to have a lossless code.

pattern $p \in P'$. We have

$$L(CT) = L_{\mathbb{N}}(|P'|) + L_{\mathbb{N}}(usg(P)) + \log \binom{usg(P) + |\Omega_o| - 1}{|P'| + |\Omega_o| - 1} + \sum_{p \in P'} L(p) \; . \tag{6}$$

To encode a pattern $p \in P'$, Eq. (7), we first transmit its length using $L_{\mathbb{N}}$. We then encode which events and generalizations it includes, and finally for each $\alpha \in p$ we encode whether and if so on which earlier generalization it depends (the plus one corresponds to a dummy symbol that represents independence). Formally,

$$L(p) = L_{\mathbb{N}}(|p|) + |p| \log(|\Omega|) + \sum_{i=1}^{|p|} \log(k+1) \quad , \tag{7}$$

with k is $|\{j \mid p[j] \in \Omega_g, i < j\}|$ if $p[i] \in \Omega_g$ else $k = 0$. By which we have a lossless encoding for model M, $L(M) = L(\Omega_g) + L(CT)$, and data D, by which we can now formally state the problem.

The Minimal Generalized Pattern Set Problem *Given a sequence database D over an event alphabet Ω_o, find the smallest pattern set P and generalization set Ω_g such that the total encoded size*

$$L(D, M) = L(M) + L(D|M)$$

is minimal.

For a given database D over observed alphabet Ω_o there exist exponentially many patterns sets P, exponentially many generalization sets Ω_g, and exponentially many possible covers C. Worst of all, the search space of neither the overall nor of the subproblems exhibits any structure such as (weak) monotonicity or submodularity that we can exploit for our search. Hence, we resort to heuristics.

4 ALGORITHM

To find good models in practice we propose to break the problem into two parts: 1) given a model M find a good cover C, and 2) given a cover C find a good model M. We discuss these in turn.

4.1 Covering the Data

We start by determining a good cover C given a model M. A valid cover is a set of windows that covers each event in database D only once. To find a C that minimizes $L(D|M)$ we first need for each pattern $p \in P$ all windows in D that match p. To find these efficiently, we use an inverted index.

Next, we describe how we find a cover C given a set of windows. Given that there are exponentially many possible covers [3], determining the optimal cover is computationally not feasible, therefore we approach this problem greedily. To this end, we define an order over all windows where we consider window $w_1 > w_2$ if, in order of priority, $|p_1| > |p_2|$, $gaps(w_1) < gaps(w_2)$, $support(p_1) > support(p_2)$, and finally lexicographically, where w_1 is a window of pattern p_1, analogously for w_2. Intuitively, we prefer patterns that cover many events with as few gaps as possible, as these will likely result in the shortest description of D. To find a cover given a window set, we consider each window w. If there does not exist a higher ranked window that conflicts, i.e. overlaps, with w, all windows that conflict with w are discarded. We repeat this process until all conflicts have been resolved, resulting in a valid cover of D. We provide the pseudo-code and details in Appx. A.1.

Algorithm 1: REFINE

```
input  : pattern $p$ and current cover $C$
output : set of candidates $Q$
1 $F \leftarrow$ FREQUENTFOLLOWERS$(p, C)$
2 $Q \leftarrow p \times \{q \in F \mid |q| > 1\}$
3 $Q \leftarrow \{q \in Q \mid \Delta\bar{L}(q) > 0\}$
4 $Q \leftarrow Q \cup$ EXTENDPATTERN$(p, \{q \in F \mid |q| = 1\})$
5 return $Q$
```

4.2 Finding Good Models

Given a sequence database D, our overall goal here is to discover a set of generalizations Ω_g and a set of patterns P that together describe D well. The general idea of our proposed algorithm is to start with an 'empty' model M_0 that only includes the observed events Ω_o and to iteratively and greedily refine this model by adding patterns and generalized events that improve the total encoded length. In each iteration, we generate a set of candidate patterns based on the current model, evaluate these, and if a candidate improves the model, we add it to the model. To avoid getting stuck with stale patterns and generalizations, we clean up at the end of each iteration by flattening generalizations and merging similar patterns. We now explain these steps in detail.

Generating Candidates. A key part of our proposed algorithm is to find improved or *refined* versions of a given pattern p. We provide the pseudocode as Algorithm 1. The general idea for refining a pattern p is to check whether there is any structure in the events and patterns that often occur soon after p in cover C. We then generate candidate patterns by concatenating pattern p with patterns q (line 2) that occur within the maximum number of allowed gaps $n|p|$ (l. 1). We discard all candidates for which we estimate that they will not lead to any gain (l. 3). With $\Delta L(q)$ we denote how many bits we actually save (or lose) by adding q to our model, but as computing this exactly is computationally costly we instead use an efficiently computable *optimistic estimate* $\Delta\bar{L}$ that we define below.

It is relatively straightforward to see how to instantiate the above strategy for refining an existing pattern with a singleton or pattern $q \in P$, as it essentially amounts to counting how often in C every possible q occurs within the maximum window length around p. It is much less clear how to discover good candidate generalizations α, however. The first idea that comes to mind is to 'simply' first use the above strategy to find a model M that only includes patterns over Ω_o, and then to merge those patterns in M that are most similar, replacing the events where they differ with a new generalized event α. While this strategy works to a certain extent, it can only discover the most frequent generalized events and patterns, and will not truly solve the problem at hand.

We therefore propose an improved strategy for discovering generalizations, where for a given pattern p we consider the distribution of when which events happen close to p in C. We provide pseudocode in Algorithm 2. The main idea is that if two or more events a and b often occur within a similar number of time steps after pattern p, they are good candidates to be included in a new generalized event as there is evidence they have a similar contextual (possibly, semantic) relation to p. Specifically, we propose to

Algorithm 2: ExtendPattern

```
input  : pattern $p$ and delay distributions $F$
output : Candidate pattern $p*$
1  $F \leftarrow$ ExtendWithGeneralizations($F$)
2  $e' \leftarrow \arg\max_{e \in F} \frac{counts(e)}{\tilde{E}}$
3  $p' \leftarrow p \times e'$, $p* \leftarrow p$
4  $\Omega_g^{\oplus} \leftarrow \emptyset$
5  $Q \leftarrow [(e, (|p|, e')) \quad | e \in F]$
6  while $\Delta\bar{L}(p') > \Delta\bar{L}(p*)$ increasing do
7      $p* \leftarrow p'$
8      $e, (i, e') \leftarrow$ top($Q$)
9      $p_1 \leftarrow cp_i(p, i{-}1, e)$ if $\tilde{E}{<}\tilde{E}'$ else $cp_i(p, i, e)$
10     if $e' \in \Omega_g$ then
11         $\alpha \leftarrow e'$
12         $\Omega_g^{\oplus\prime} \leftarrow \Omega_g^{\oplus} \cup \{(\alpha_{id}, \{e\})\}$
13     else
14         $\alpha \leftarrow \emptyset$
15         $\Omega_g^{\oplus\prime} \leftarrow \Omega_g^{\oplus} \cup \{(\alpha_{id}, \{e', e\})\}$
16     $p_2 \leftarrow cp_r(p, i, \alpha)$
17     $p' \leftarrow \arg\max_{p \in \{p1, (p2, \Omega_g^{\oplus\prime})\}} \Delta\bar{L}(p)$
18     $\Omega_g^{\oplus} \leftarrow \Omega_g^{\oplus\prime}$ if $p' = p2$
19     update $Q$
20 return $(p*, \Omega_g^{\oplus})$
```

Algorithm 3: Merge

```
input  : Pattern set $P$ and generalization $\Omega_g$, cover $C$
output : Pattern set $P$ and generalization $\Omega_g$, cover $C$
1  $Q \leftarrow []$
2  forall $p \in P$ do
3      $q \leftarrow \arg\max_{\{q \in P| \ |q|=|p|\}}$ overlap between $p$ and $q$
4      if overlap between $p$ and $q > 1$ then $Q.add(p, q)$;
5  forall $p, q \in Q$ do in order of 1. overlap 2.combined usage
6      $p', \Omega_g^{\oplus} \leftarrow$ merge $p$ and $q$
7      if $\Delta L(p') > 0$ then
8          $P \leftarrow (P \cup \{p'\}) \setminus \{p, q\}$
9          apply $\Omega_g^{\oplus}$ to $\Omega_g$
10         replace all $p$ and $q$ with $p'$ in $C$
11 return $P, \Omega_g, C$
```

generate candidate generalized events based on the similarity of the distributions of delays between pattern p and occurrences of events $e \in \Omega$. A delay distribution of event $e \in \Omega$ relative to pattern p captures how often and how many times steps after p event e occurs; technically we implement this non-parametrically using a histogram with one bin per time step. We construct these delay distributions in line 1 of Alg. 1 for all $q \in \Omega$ that occur within the $n|p|$ time steps after p in C.

To maximize the chance that the resulting generalization will improve the overall cost, we start extending a pattern p with the event e' that has the highest delay probability mass, i.e. occurs frequently with the same delay after p, formally $e' = \arg\max_{e \in F} counts(e)/\tilde{E}$, where $\tilde{E}$ is the median delay between p and e (l. 2-3). Next, we seek if there are other events that together with the just added e' would instead form a promising generalization α. We do this by testing events $e \in F$ in order of the most similar delay distribution to e' normalized by frequency. That is, $W1(E', E)/counts(e)$ where E' and E refer to the respective distributions, and $W1$ is the Wasserstein distance [32] between two delay distributions, defined as

$$W1(E1, E2) = \min_N \sum_{i \in E1} \sum_{j \in E2} n_{i,j} d_{i,j} \quad ,$$

where $n_{i,j}$ refers to the probability mass that has to be moved between i and j and $d_{i,j}$ to the distance. By $counts(e)$ we refer to the frequency within $n|p|$ time steps after p.

This may once again seem like a sound strategy, but comes with the problem that events that are next to each other will have similar delay distributions. That means we have to additionally test whether event e is truly part of a generalization together with e' or is simply a regular neighbor to that event or generalization. We therefore, evaluate the gain in compression by either extending the generalization with a new event (l. 16) or by extending the pattern directly with the event left or right (l. 9). Note with $cp_i(p, i, e)$ we create a new pattern where event e is inserted behind the i^{th} event of pattern p, and analog with $cp_r(p, i, \alpha)$ we create a new pattern where the i^{th} event is replaced by α. Given both extensions, we take the one for which our estimated gain is higher (l. 17). After each added event we update the priority queue. Here we described the procedure of extending a pattern by adding events or patterns to the back; we do the same with preceding events and patterns. This concludes the description of how we create a set of pattern candidates and generalization candidates given a pattern and the current cover.

Fine-Tuning Candidates. The overall algorithm takes the candidates generated above, and tests, in order of estimated gain, for addition to the model. Testing a candidate for addition involves computing $L(D, M)$ and therewith a new cover C'. That is, we now know where and which instances of p are used, and can take advantage of that information and further refine the pattern to minimize the total encoded length. We do so by pruning the generalized events to only include those instances that are used in a way that aids compression. To this end, we test for each event in $\Omega_g^{\oplus}$ whether removing it will improve the gain in bits (PruneGen). As we allow for gaps in the occurrences, we can further take advantage of the cover C' by extending pattern p with events that frequently occur in these gaps (RefineInterleaving). This includes generalized events that are built from multiple observed events that occur in the gaps of p. We provide the pseudocode for both procedures, as well as a more detailed description, in Appx. A.4.

Simplifying the Model. By iteratively adding more specialized patterns and generalizations to the model, previously discovered patterns and generalizations may no longer positively contribute to the MDL score. We therefore prune the model after each iteration by merging similar patterns and by flattening generalizations. We discuss these in turn.

Algorithm 4: Flatten

```
input  : Pattern set $P$ and generalization $\Omega_g$
output : Pattern set $P$ and generalization $\Omega_g$
1 forall $\alpha \in \Omega_g$ do
2     if $\alpha$ used in just one other $\beta \in \Omega_g$ then
3         extend $\beta$ with $fl(\alpha)$
4         forall $p \in P$ do
5             replace $\alpha$ with $\beta$ in $p$
6         $\Omega_g \leftarrow \Omega_g \setminus \{\alpha\}$
7         if $L(D, M)$ did not decrease then
8             revert all changes
9 return $P, \Omega_g$
```

We provide pseudocode of the merge procedure as Algorithm 3. We consider merging two patterns p and q if they have the same length (l. 3) and have an overlap of at least two events (l. 4) where we say an event overlaps if $p[i] = q[i]$. To prioritize pattern mergers likely to improve compression and meaningful generalizations, we merge patterns in order of overlap and combined pattern usage, both decreasing (l. 5). When merging two patterns we create for all events where $p[i] \neq q[i]$ a new generalization $\alpha = \{p[i], q[i]\}$. Through this process it is possible to create the same generalization twice, if that happens we replace all instances with the same generalization and delete the other one. Since the new pattern will match all windows that the source patterns matched, we do not have to recompute a new cover and can directly compute by how many bits our encoding will change (l. 7). If we have a positive gain we keep the new pattern, and the corresponding generalization, and discard the two source patterns (l. 8).

Next, we discuss how we simplify the generalizations. We provide the pseudocode as Algorithm 4. If a generalization α is used in only one other generalization β, we consider merging it with its parent(s) (l. 3), meaning we add all events $e \in fl(\alpha)$ to β and replace α with β in all patterns $p \in P$ (l. 5). Similar to above, as the updated patterns match the same positions as before we can compute the total encoded size without recomputing the cover. If we obtain a gain, we keep the change, otherwise we revert (l. 8).

As both types of simplification steps can create new candidates for the other, we call Merge and Flatten alternating until convergence (Appx. A.4, Alg. 9). The Simplify algorithm can also be applied to post-process the results of traditional sequential pattern miners in order to reveal generalizations from surface-level patterns. We will use it as such in the experiments to permit a comparison to the state of the art.

Estimating Gains. Exact computation of our MDL score requires computing the cover, which is a computationally costly operation. Rather than always relying on the exact score, we use an *optimistic estimator* $\Delta\overline{L}(p, \Omega_g^{\oplus})$ of the gain in compression where possible. We can estimate the gain $\Delta\overline{L}$ by breaking it down into two parts: the cost of pattern p in the model and the change to the encoding of the data by the updated model. The bits needed to describe a new pattern or generalization can be computed efficiently as shown in Sec. 3, and no estimation is necessary; when extending an existing

Algorithm 5: Flock

```
input  : sequence database $D$ over alphabet $\Omega_o$
output : pattern set $P$, generalization set $\Omega_g$
1  $O,\ P \leftarrow \Omega_o,\ \Omega_g \leftarrow \emptyset,\ C \leftarrow D,\ gain \leftarrow +\infty$
2  while $gain > 0$ do
3      $PQ \leftarrow [],\quad gain \leftarrow 0$
4      foreach $p \in O$ do
5          $PQ.addAll(\text{Refine}(p, C))$
6      $O \leftarrow \emptyset$
7      while not $PQ.empty()$ do in order of $\Delta\overline{L}(p', \Omega_g^{\oplus})$
8          $p', \Omega_g^{\oplus}, p \leftarrow \text{top}(PQ)$
9          $gain', C' \leftarrow \Delta L(p, \Omega_g^{\oplus})$
10         if $gain' > 0$ then
11             $\Omega_g^{\oplus}, C \leftarrow \text{PruneGen}(p', \Omega_g^{\oplus}, C')$
12             $p', \Omega_g^{\oplus}, C \leftarrow \text{RefineInterleaving}(p', \Omega_g^{\oplus}, C)$
13             $O \leftarrow O \cup \{p', p\}$
14             apply $\Omega_g^{\oplus}$ to $\Omega_g$
15             $P \leftarrow P \cup \{p'\}$
16             $gain \leftarrow \max(gain, gain')$
17     $P, \Omega_g, C \leftarrow \text{Simplify}(P, \Omega_g, C)$
18 return $\text{Prune}(P, \Omega_g, C)$
```

generalization we simply consider the difference in encoding cost between the old generalization and extended generalization. We propose to optimistically estimate the encoded cost of the data given the updated model, by using the usage statistics of the previous cover. To give the intuition, suppose we create a new pattern p by concatenating q and r, we then estimate the usage of p by the minimum usage of q and r, and estimate the new usages of these patterns by subtracting exactly that amount. We then simply compute $L(D|M)$ with these estimated usages. We give further details on how we estimate the new usages and how to compute $\Delta\overline{L}(p', \Omega_g^{\oplus})$ in Appx. A.3.

4.3 The Flock Algorithm

Now that we have seen all the individual parts, we can explain the Flock algorithm in detail.[2] The general idea is to start with an empty pattern set P and an empty generalization set Ω_g and iteratively add patterns and generalizations until adding new patterns no longer improves our model M. We give the pseudocode in Algorithm 5. To keep track of patterns we want to extend to a more refined version we maintain a set O, and initialize O with all singletons (line 1). At each iteration, we refine all $p \in O$ to pattern candidates (l. 5). A candidate base on pattern p consists of two parts, a more specific pattern p' and a generalization extension $\Omega_g^{\oplus}$. All pattern candidates are added to a priority queue that we order by the estimated gain.

Next, we test each candidate. If the candidate gives us an actual gain, we fine-tune the candidate (PruneGen, l. 11, and RefineInterleaving, l. 12) and add it to the model (l. 15). To allow for different refinements of source p and further refinements of pattern

[2]The name Flock comes from the expression 'birds of a feather flock together', which was the inspiration for how we search for generalizations.

p' we add both to the open set O. Iteratively adding candidates to the model does not necessarily result in the most succinct representation, therefore we simplify the model after each iteration (l. 17). As patterns can become superfluous as more specific patterns are added, we test for each pattern whether removing it will decrease the total number of bits, before returning the model (l. 18).

Complexity. Finally, we consider the time complexity of FLOCK. In the worst case, we have to find all windows for all possible sequential patterns over alphabet Ω to cover the data. We can find all windows of a pattern p in $O(||D||^2)$ [3], where $||D||$ denotes the total number of events in D. To cover the data, in the worst case, we have to sort all windows, $O(||D|| \log ||D||)$ and check each window against all already selected $O(||D||\ |C|)$. Combined, this gives us an overall complexity of $O(||D||\ |\mathcal{F}|(||D|| + \log(||D||) + |C|))$.

5 RELATED WORK

While not technically pattern mining, research in Natural Language Processing (NLP) on finding synonyms, computing similarities, and constructing ontologies over words are related to our work. Extensive ontologies have been constructed that capture relationships between words [21] but only little work exists on automatically discovering high-quality ontologies directly from data. Neural networks have been shown to produce embeddings that place semantically similar words close to each other [5, 20] yet unlike our patterns these embeddings do not allow for a straightforward interpretation. In the experiments we will compare FLOCK to WORD2VEC with resp. to the bag-of-words and skip-gram architecture [20].

Process mining is more closely related to sequential pattern mining as it also considers event sequences. Instead of mining insightful patterns, it however focuses on discovering process models with explicit temporal semantics for reconstructing sequences from start to finish [35]. As these processes can get very complex, methods have been proposed to abstract sub-processes into high-level activities [14, 29, 31], whereas we aim to find generalizations over individual events. The key difference to process mining is that we focus on event sequence data in general, and are interested in patterns that characterize these sequences without requiring that every sequence has been produced by the same process.

Our work fits within the rich history of sequential pattern mining. Traditionally, the field focused on discovering *all* frequent patterns [16, 19, 34]. Such methods can be adapted to only report patterns that match predefined constraints, including or-structures through regular expressions [11, 22, 23], in contrast to our method the specific or-structures have to be provided beforehand and are not discovered. As reporting *all* frequent patterns often results in overly large and highly redundant results, modern approaches instead focus on discovering patterns that are either significant with regard to some null-hypothesis [15, 24, 25] or discovering sets of patterns that together generalize the data well [9, 30]. For the latter, the MDL criterion has been particularly successful [2, 10, 30]. Out of these approaches, we compare to SKOPUS [24], SQS [30], ISM [9].

Mining *generalized* sequential patterns has been studied in the seminal work of Srikant and Agrawal [27, 28] they however require a taxonomy and suffer from the well-known pattern explosion of frequent pattern mining. Closer to our method are the proposals of Grosse and Vreeken [12] and Beedkar and Gemulla [1] who both study the problem of summarizing data *given* an ontology. We, on the other hand, aim to discover both the generalizations and patterns without prior knowledge.

SQUISH [3] comes closest to our approach, as it is able to discover patterns that include or-structures and follows a similar MDL based approach. SQUISH discovers or-structures in a post-processing step where it combines discovered pattern instances that are exactly the same except for one event. In contrast, we jointly search for generalizations and generalized patterns, by which we can identify much richer (more subtle) generalizations. We allow generalizations to be re-used between patterns, as well as explicitly model dependencies between generalizations within a pattern in order to obtain highly informative models. We compare to SQUISH [3] in the experiments.

6 EXPERIMENTS

In this section, we empirically evaluate FLOCK on synthetic and real-world data. We implemented FLOCK in C++ and provide the source code for research purposes, along with the used datasets in the supplementary material.[3] We compare to SQS [30], SQUISH [3], ISM [9], SKOPUS [24], and WORD2VEC [20].

As SQS, ISM, and SKOPUS only consider surface-level events, we post-process their results using the SIMPLIFY Algorithm, to extract generalized patterns and generalized events from their results. We consider both the bag-of-words and skip-gram defined versions of WORD2VEC [20], clustering the embedding using DBSCAN [7], merging events part of the same cluster into a new generalized event, and finally apply SQS on the resulting data to find generalized sequential patterns. As per default, we set the gap parameter of FLOCK to $n = 10$, in Appx. B.3 we provide a sensitivity analysis that shows that FLOCK is robust against the parameter choice. For data with very low structure we see that a low n produces better results. We give a more detailed description of the experimental setup, and an ablation study on parts of the algorithm, in Appendix B.

6.1 Synthetic Data

To evaluate how well FLOCK recovers the ground truth we consider synthetic data. We sample, uniformly at random, databases D consisting of 100 sequences S, each of length 200, over an alphabet Ω_o of size 500. We additionally consider between 0 to 6 generalized events $\alpha \in \Omega_g$, each of which, unless stated otherwise, consists of five observed events $e \in \Omega_o$ that are sampled uniformly at random. We plant patterns of length 10. We ensure 10% of all planted instances are interleaved in the data, meaning the next pattern starts before the last one ends. We ensure that each planted instance does not collide (overwrites) with an earlier planted instance.

All results on synthetic data are averaged over ten independently generated datasets. In terms of runtime FLOCK is comparable to SQS and SQUISH; for all reported experiments all three finish within seconds to minutes. The competing methods take much longer: for SKOPUS, which is a top-k method, we had to set k to 50 and limit the pattern length to 10 to keep the run-time under 24h.

We evaluate the reported pattern sets with standard $F1$ score. As to not only reward exact recovery of planted patterns, we map planted patterns to reported patterns. We allow a pattern to map to at most one other pattern, and only map it when it is a sub or

[3]https://eda.rg.cispa.io/prj/flock/

super-sequence of the other. We do the same for generalizations, where we allow a mapping if the planted is a subset of reported generalization, or vice versa. For both, we pick the mapping that results in the maximum number of pairs.

First, as a sanity check, we run Flock on data without any structure, i.e. no planted patterns. Flock correctly reports no patterns. Next, we consider the setting where we start with 30 independent patterns without any generalization, and in each subsequent experiment we replace five of these with one generalized pattern p, where one event in p is a generalized event. Colloquially speaking, we answer the question: "How well does Flock pickup generalized patterns compared to surface level patterns?". We show the results in Figure 2a. We observe in the initial setting without any generalization we are on par with Sqs and Squish, the *Word2Vec* based approaches are next best, while Ism and Skopus perform worst. However, as we increase the number of generalized patterns, Flock maintains a high $F1$ score throughout while the score of all other methods decreases significantly.

Next, we consider a more difficult setting. We sample five generalized events and plant 5 patterns each containing 2 generalizations. Note that this means different patterns share the same generalization. To investigate performance under decreasing support, we decrease the total number of planted pattern instances from 400 to 50 in steps of 50. This setup aims to answer the question: "How frequent have patterns to be to be discovered?". We show the results in Figure 2b. We observe that Flock beats the other methods by a wide margin, with Sqs and Squish in second place. Flock performs very well up to 100 planted instances, at 50 the score drops significantly; as an individual instance of a pattern on expectation then only occurs 0.4 times this is unsurprising.

To test how Flock behaves when the pattern frequency stays the same, but the individual instances get less frequent, we consider the case where we increase the number of events per generalization (cf. Figure 2c). We observe a very wide margin to all other methods. An increase in generalization size only has a very small effect on Flock's ability to recover the planted patterns.

Finally, we evaluate the quality of the reported generalizations (cf. Figure 2d). To do so we generate data containing two patterns, sharing one generalization. To see how well we recover the generalization if some events are much less frequent, we decrease the usage of events in the generalization linearly to zero, and in each subsequent experiment we increase the number of events per generalization. With that, we aim to answer the question: "How accurately does a generalized event get recovered?". We again report the $F1$ score, this time computed over how well the individual events within the generalization are recovered. We omit Skopus from this experiment as a single run did not terminate within 24h. We see that Flock recovers the generalization well, while Squish and Sqs do well in a simpler setting, they are however not robust against larger generalizations, unlike Flock.

On the synthetic data experiments we have seen that Flock outperforms all other methods clearly. In some simple settings Sqs and Squish are on par with Flock. The Word2Vec approaches only do reasonably well on data with no or very few generalizations, inspecting the results this is likely mostly due to Sqs. Ism and Skopus do worst throughout the experiments.

		Flock		Sqs		Squish	
Dataset	$\|\Omega_o\|$	$\|P\|$	$\|\Omega_g\|$	$\|P\|$	$\|\Omega_g\|$	$\|P\|$	$\|\Omega_g\|$
ECG	200	4	1	128	12	88	6
Short-ECG	200	3	2	3	0	4	1
BPI-2015	192	189	53	400	6	525	35
Rolling Mill	836	195	56	430	35	554	73
Moby	10276	239	1	231	0	202	26
JMLR	3845	466	5	580	0	480	87

Table 1: Results on real datasets. We given alphabet size $|\Omega_o|$, and the number of reported patterns $|P|$, and number of generalization $|\Omega_g|$, for each method. Overall we observe that Flock reports fewer patters and more generalizations, making the model easier to interpret.

6.2 Real-World Data

To evaluate if Flock finds meaningful structure in real-world data, we test Flock on five distinct datasets, electrocardiograms (*ECG*),[4] a business event log (*BPI-2015*),[5] a rolling mill production log (*Rolling Mill* [35]), and two text datasets (*JMLR* and *Moby* [30]). We compare Flock to the two best performing competitors, Sqs and Squish. Since the *ECG* and text datasets have a very low amount of structure we set $n = 2$. We run all three methods on each dataset and report the number of discovered patterns and generalizations in Table 1.

First, we consider the *ECG* dataset, we note that compression rates are similar (Appx. B.4) but there is a big difference in the number and quality of patterns. Flock can capture the key structure in just four patterns, while Sqs reports 128 patterns and Squish reports 88 patterns. As this dataset contains enough events such that each individual instance is still strongly represented we reduced the number of events drastically, from 100k to 3k (*Short-ECG*). We find that Sqs and Squish report shorter patterns than Flock, Flock is able to discover generalized events enabling it to find longer patterns over this extended alphabet.

The next two datasets we consider are event logs (*BPI-2015* and *Rolling Mill*), characterized by strongly repetitive and structured behavior. Flock finds patterns that describe a more general behavior which we do not observe for the other methods. To demonstrate that Flock discovers generalizations with strong dependencies between each other we show a pattern discovered on the *Rolling Mill* dataset in Figure 3. The instance of α has a strong influence on β while β determines the value of γ and finally the value of γ has a strong influence on the value of δ, see zoom box. This pattern covers 14 possible instances within one pattern, including rare instances where the usual procedure is not followed.

Finally, to see how well Flock handles settings with large alphabets we consider text data. We consider a set of abstracts from the *JMLR* journal, and the novel *Moby* Dick by Herman Melville. For the *JMLR* dataset, we see that Flock reports fewer patterns than Sqs and Squish while capturing the same amount of structure, allowing for a more interpretable representation (see Table 1). We

[4]https://physionet.org/content/stdb/1.0.0/
[5]https://data.4tu.nl/collections/BPI_Challenge_2015/5065424/1

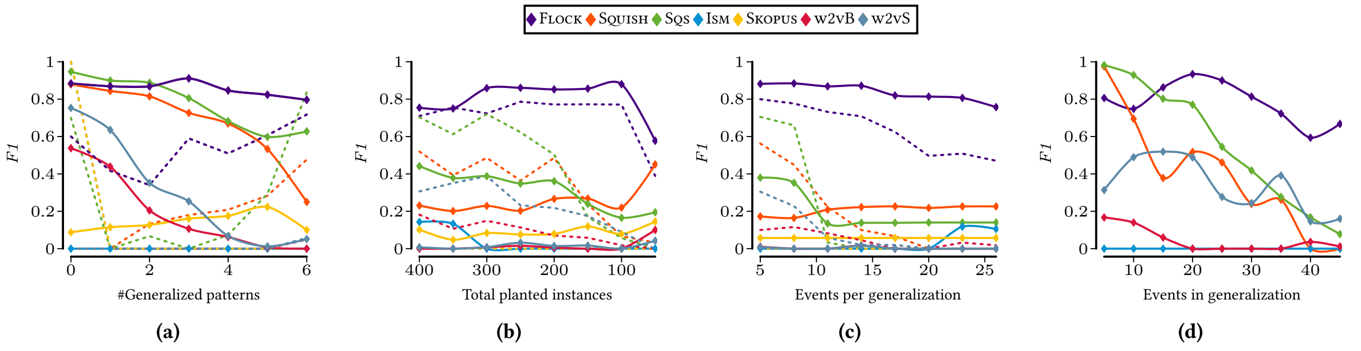

Figure 2: F1 score for recovery of planted patterns (solid line, Fig. a-c) on synthetic data over (a) number of patterns containing one generalized event, (b) total number of planted pattern instances, and (c & d) number of observed events per generalized event. F1 score for recovery of generalized events (dotted line, Fig. a-c). In Plot (d) we evaluate (F1 score) the recovery of events per generalization. Overall we see that Flock beats the other competitors by a wide margin.

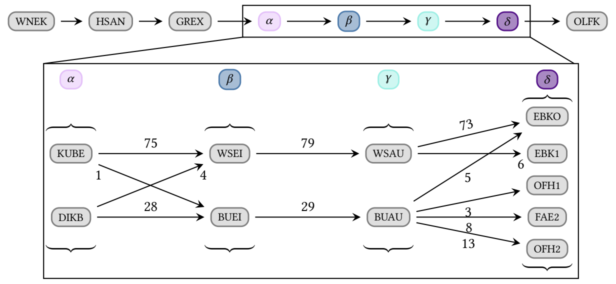

Figure 3: Example pattern discovered by Flock on the *Rolling Mill* dataset, of length 8 out of which 4 are generalized events ($\alpha, \beta, \gamma, \delta$). We see the value of the previous generalization strongly influences the next generalization (see zoom box): e.g. if α has value "kube", β has value "wsei" in 75 of 76 cases.

show a selection of the patterns discovered by Flock on *JMLR*, *Moby* and *BPI-2015* in Appendix B.4.

7 DISCUSSION

The experiments on real-world data show that Flock performs well in practice. It recovers surface-level patterns as well as the state of the art, but additionally is also able to recover ground-truth generalizations, generalized patterns, as well as the dependencies between generalizations within patterns. The models that Flock discovers are smaller, less redundant, and the more expressive patterns it discovers provide clear insight into the data-generating process.

It is worth commenting on interpretability. A single surface-level pattern is arguably easier to interpret than a generalized pattern, and if matching the ground truth, so is a set of surface-level patterns compared to a set of generalized patterns. One of the key strengths of Flock is that its MDL objective will automatically determine if it is better to model the data at hand with surface-level or generalized patterns; for the former, the experiments show that it is as able as Sqs [30], Ism [9], and Squish [3] in discovering true surface-level patterns, while it is unique in its capability to discover generalizations that allow it to show the forest for the trees.

As good as its results are, we do see ample opportunity to improve Flock further. We currently penalize gaps uniformly per pattern. It may well be, however, that for a given pattern gaps occur exclusively between the second and third event; extending the pattern language and MDL encoding accordingly would provide valuable additional insight. Second, we see enormous chances in formulating the problem in differentiable rather than as a combinatorial term. Fischer and Vreeken [8] recently showed how to discover sets of item sets using a binarized auto-encoder, and it would be a break-through to achieve the same for sequential pattern mining as this would allow considering datasets and alphabets that are orders of magnitude larger than the current state of the art. Third, we consider it very interesting to explore how to incorporate background knowledge in the form of a given ontology, a set of generalizations, i.e. sets of surface-level events that we know or expect should behave similarly, or a similarity matrix over events. To a certain extent, our current problem formulation already allows for this: we can initialize Flock with a model M that includes the corresponding generalizations. It is, however, not immediately clear how to best continue from there; should Flock consider these given generalizations as immutable parts that cannot be pruned, or as a suggestion that can be refined?

8 CONCLUSION

We considered the problem of summarizing an event sequence database with generalized sequential patterns. To that end, we introduced the concepts of generalized events and generalized sequential patterns. To find succinct and non-redundant models, we formalized the problem using the Minimum Description Length principle, and presented the efficient Flock algorithm to find good pattern sets in practice.

Experiments on synthetic and real world data showed that Flock works well in practice and provides insight beyond what is possible with existing surface-level pattern mining methods, even with post-processing. To further improve Flock we plan, as future work, to study how to best incorporate background knowledge as well as how to scale up through continuous optimization based search.

REFERENCES

[1] Kaustubh Beedkar and Rainer Gemulla. 2015. LASH: Large-Scale Sequence Mining with Hierarchies. In *PODS*. Melbourne Victoria Australia, 491–503.

[2] Roel Bertens, Jilles Vreeken, and Arno Siebes. 2016. Keeping it Short and Simple: Summarising Complex Event Sequences with Multivariate Patterns. In *KDD*.

[3] Apratim Bhattacharyya and Jilles Vreeken. 2017. Squish: Efficiently Summarising Event Sequences with Rich Interleaving Patterns. In *SDM*. 11.

[4] Joscha Cüppers, Janis Kalofolias, and Jilles Vreeken. 2022. Omen: Discovering Sequential Patterns with Reliable Prediction Delays. *KAIS* 64, 4 (2022), 1013–1045.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[6] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (April 1972), 248–264.

[7] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, Vol. 96. 226–231.

[8] Jonas Fischer and Jilles Vreeken. 2021. Differentiable pattern set mining. In *KDD*. 383–392.

[9] Jaroslav Fowkes and Charles Sutton. 2016. A Subsequence Interleaving Model for Sequential Pattern Mining. In *KDD*.

[10] Esther Galbrun. 2022. The Minimum Description Length Principle for Pattern Mining: A Survey. *Data Min. Knowl. Disc.* 36, 5 (2022), 1679–1727.

[11] Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2002. Mining sequential patterns with regular expression constraints. *IEEE Trans Knowl Data Eng* 14, 3 (2002), 530–552.

[12] Kathrin Grosse and Jilles Vreeken. 2017. Summarising Event Sequences Using Serial Episodes and an Ontology. *DMNLP* (2017), 16.

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

[14] Christian W Günther and Wil MP Van Der Aalst. 2007. Fuzzy Mining–Adaptive Process Simplification Based on Multi-Perspective Metrics. In *BPM*. Springer.

[15] Steedman Jenkins, Stefan Walzer-Goldfeld, and Matteo Riondato. 2022. SPEck: Mining Statistically-Significant Sequential Patterns Efficiently with Exact Sampling. *Data Min Knowl Disc* 36, 4 (July 2022), 1575–1599.

[16] Srivatsan Laxman, P. S. Sastry, and K. P. Unnikrishnan. 2007. A fast algorithm for finding frequent episodes in event streams. In *KDD* (San Jose, California, USA). 410–419.

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

[18] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: A Novel Symbolic Representation of Time Series. *Data Min. Knowl. Disc.* 15, 2 (2007), 107–144.

[19] Heikki Mannila. 1997. Discovery of Frequent Episodes in Event Sequences. *Data Min. Knowl. Disc.* (1997), 31.

[20] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[21] George A Miller. 1995. WordNet: A Lexical Database for English. *Commun. ACM* 38, 11 (1995), 39–41.

[22] Jian Pei, Jiawei Han, and Wei Wang. 2002. Mining sequential patterns with constraints in large databases. In *CIKM*. 18–25.

[23] Jian Pei, Jiawei Han, and Wei Wang. 2007. Constraint-based sequential pattern mining: the pattern-growth methods. *JIIS* 28, 2 (2007), 133–160.

[24] François Petitjean, Tao Li, Nikolaj Tatti, and Geoffrey I. Webb. 2016. Skopus: Mining Top-k Sequential Patterns under Leverage. *Data Min. Knowl. Disc.* 30, 5 (Sept. 2016), 1086–1111.

[25] Sam Pinxteren and Toon Calders. 2021. Efficient Permutation Testing for Significant Sequential Patterns. In *SDM*. SIAM, 19–27.

[26] Jorma Rissanen. 1983. A Universal Prior for Integers and Estimation by Minimum Description Length. *Annals Stat.* 11, 2 (1983), 416–431.

[27] Ramakrishnan Srikant and Rakesh Agrawal. 1996. Mining Sequential Patterns: Generalizations and Performance Improvements. In *International Conference on Extending Database Technology*. Springer, 1–17.

[28] Ramakrishnan Srikant and Rakesh Agrawal. 1997. Mining Generalized Association Rules. *Future Generation Computer Systems* 13, 2-3 (Nov. 1997), 161–180.

[29] Yaguang Sun and Bernhard Bauer. 2016. A Graph and Trace Clustering-based Approach for Abstracting Mined Business Process Models:. In *ICEIS*. Rome, Italy.

[30] Nikolaj Tatti and Jilles Vreeken. 2012. The Long and the Short of It: Summarizing Event Sequences with Serial Episodes. In *KDD*. ACM, 462–470.

[31] Boudewijn F van Dongen and Arya Adriansyah. 2009. Process Mining: Fuzzy Clustering and Performance Visualization. In *BPM*. Springer, 158–169.

[32] Cédric Villani. 2009. *Optimal Transport: Old and New.* Number 338 in Grundlehren Der Mathematischen Wissenschaften. Springer, Berlin.

[33] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. 2011. KRIMP: Mining Itemsets that Compress. *Data Min. Knowl. Disc.* 23, 1 (2011), 169–214.

[34] Jianyong Wang and Jiawei Han. 2004. BIDE: Efficient Mining of Frequent Closed Sequences. In *ICDE*. 79–90.

[35] Boris Wiegand, Dietrich Klakow, and Jilles Vreeken. 2021. Mining Easily Understandable Models from Complex Event Logs. In *SDM*. 10.

Algorithm 6: GreedyCover

```
input  : set of windows $W$ and $D$
output : list of ordered windows covering $D$
1  $i \leftarrow 1$
2  sort $W$ by $j_1$
3  while $i < |W|$ do
4      $gw \leftarrow i$
5      $gws \leftarrow gw + 1$
6      while $W[gws].first \leq W[gw].last$ do
7          if $W[gws] > W[gw]$
            and $W[gws]$ in conflict with $W[gw]$ then
8              $gw \leftarrow gws$
9          $gws \leftarrow gws + 1$
10     if $gw = i$ then $i \leftarrow i + 1$ ;
11     $wr \leftarrow i$
12     while $W[wr].first \leq W[gw].last$ do
13         if $W[wr]$ in conflict with $W[gw]$ then
14             remove $wr$ from $W$
15         $wr \leftarrow wr + 1$
16 return $W$
```

A ALGORITHM

A.1 Cover

Given a Model M, we want to find the shortest description of sequence database D given model M, i.e. that cover C that minimizes $L(D|M)$. To this end we need for each pattern $p \in CT$, all subsequences $S[j_1, j_2, \ldots, j_{|p|}]$ where p matches, that is window set W.

In the main body of the paper, we explained what a valid cover is and what it means for two windows to be in conflict. Here we describe how we actually find a valid cover C.

We show pseudocode in Algorithm 6. Given a set of windows W we first sort W by j_1. The general idea is now to move a pointer i from left to right through W where all windows to the left of i are conflict free whereas to the right we still have to resolve all conflicts. We iterate over the window list W while maintaining three pointers, the first pointer i points to the lowest non conflict free window. The second one, greatest window gw, is used to keep track of the current greatest window after i, that is not in conflict with a greater window. The third one, greater window searcher gws, is used to find conflicting greater window then the current greatest window gw.

We initialize i with 0 such that it points to the first window in W, at this point this is, trivially, also the greatest window we have seen so far. We now search for a greater window that overlaps with gw, line 6. If we find one we update our gw pointer. Once we can't find a greater window we remove all windows that overlap with gw. To this end, we start again at position i and increase a window remover pointer wr until the first position of $W[wr]$ is larger than the last position of $W[gw]$, at this point we can be sure to have considered all windows that might overlap. If i points to the greatest window we increase i by one. Note i might overlap with gw in this case we can still simply remove it, i then just points to the next window, which is exactly what we want. We repeat this process until i points to the last window in W, at this point each event in the database D is covered by exactly one window.

Window Search. To efficiently find all windows given a pattern p we use an inverted index. We add triples (i, j, k) to a set O, where i refers to the sequence j to the next event $S_i[j]$ to be tested against $p[k]$. We initialize O with all $(i, j+1, 2)$ where $S_i[j] = p[1]$. We increase j until $S_i[j] = p[k]$, this means we found the next event in S_i that matches the next unmatched event in p. Hence we increment the j and k pointer by one, $(i, j+1, k+1)$, and additionally add $(i, j+1, k)$ to O. To illustrate why we add the second case, where we do not increase k, consider the case with sequence $abbc$ and pattern abc, to capture both windows, i.e. the one with the first and the second b we need one instance to move over b without matching it.

We continue this process until $k = |p| + 1$, at this point we found a window w that matches p and can remove the triple form O. We also remove a triple from O if the respective windows grows larger than the maximum window length of $|p| + n|p|$. We continue this process until O is empty, at this point we have found all windows of p.

A.2 Generalization refinement example

Example of applying a set of generalized refinements $\Omega_g^{\oplus}$ to a set of generalized events Ω_g. Let us consider the following case we have $\Omega_g^{\oplus} = \{(\alpha_{id}, \{c, d\}), (\gamma_{id}, \{\alpha, e, f\})\}$ and $\Omega_g = \{\alpha, \beta\}$ where $\alpha = \{a, b\}$ and $\beta = \{g, h\}$. We now apply $\Omega_g^{\oplus}$ to Ω_g. Since Ω_g already contains a generalization α we extend it with the specified events, hence $\alpha = \{a, b, c, d\}$. The generalization γ on the other hand not does not yet exist, we hence add a new generalization γ to Ω_g, hence $\Omega_g = \{\alpha, \beta, \gamma\}$ where $\gamma = \{\alpha, e, f\}$. As $\Omega_g^{\oplus}$ does not specific any additions for β it is not affected.

A.3 Optimistic Gain Estimation

In this section, we will explain how we compute the gain estimation $\Delta\bar{L}(p, \Omega_g^{\oplus})$. That is we want to estimate by how our encoding cost $L(D, M)$ changes by adding pattern p to CT and applying a set of generalization refinements $\Omega_g^{\oplus}$ to Ω_g.

The cost of encoding a new pattern, or generalized event, can be directly computed as shown in Section 3. However, when extending a generalization i.e adding new elements to an existing generalization, we have to update the cost of this generalization. To compute the difference in cost we subtract the cost of the old generalization and add the cost of the new one. Since we allow generalization to contain other generalization, and we encode these differently, we also have to do so here. So when extending an existing generalization α with k' new generalization $\beta_1, \ldots, \beta_{k'}$ and m' new events. With that, we estimate the increased cost as,

$$\begin{aligned}\Delta\bar{L}(\alpha) = & -\log\binom{i-1}{k} + \log\binom{i-1}{k+k'} \\ & -\log(|\Omega_o^*|) + \log(\Omega_o') \\ & -\log\binom{|\Omega_o^*|}{m} + \log\binom{\Omega_o'}{m+m'}\end{aligned}$$

where Ω_o^* are those observed events not defined by the old nested generalizations $\beta \in \alpha$, i.e. $\Omega_o^* = \Omega_o \setminus \bigcup_{\beta \in \alpha_i} fl(\beta)$ and Ω_o' those observed events not defined by the extended generalizations i.e. $\Omega_o' = \Omega_o^* \setminus \bigcup_{j=1}^{k'} fl(\beta_j)$, k is the number of generalizations in α and m the number of events in α before the extension. This covers the estimated cost on the model side.

Next, we consider the difference in the cover cost. To estimate the gain of a new model we have to estimate how the cost of the C_p, C_m and C_s changes. To avoid recovering the data for each candidate we estimate the effect a pattern extension, new generalization, or extending a generalization has on the usage of all patterns and singletons. We will later explain how we estimate these usages for all cases. Given the estimated usage changes for all patterns, we can compute the difference in bits needed for the pattern code stream C_p,

$$\Delta \bar{L}(C_p) = \sum_{p \in P \cup \{p'\}} usg_{new}(p) * \log(usg_{new}(p)) - usg_{old}(p) * \log(usg_{old}(p))$$

For the specification sequence. Given an estimation how often each pattern instance will be used for each pattern, we can create specification sequence $C_{s\ new}$ representing the new specification, and with $C_{s\ old}$ we denote the old specification sequence.

As we can't compute the pattern structure at this point we assume all generalization to be independent, to more accurately estimate the gain we make the same assumption for the existing patterns. The difference in encoding cost is then,

$$\Delta \bar{L}(C_s) = L(C_{s\ new}) - L(C_{s\ old}) \quad .$$

As we do not have a good way to estimate the number of needed gaps and fill codes we do not estimate the difference for the meta stream. All operations we consider create a new pattern based on an existing pattern. Importantly we do not remove the existing one i.e. the old one is still in the P. We infer a frequency estimate on the number of instances we count in Algorithm 1. Next, we consider all different refinement cases and describe how we estimate the new frequencies for the specific cases.

Case 1 (New pattern): This case we already explained in the paper and include it here, in extended form, for completeness. Creating one new pattern p^* with p_1 and p_2 where p_2 is another pattern. From F we get c, how often p_2 follows p_1. The usage of our new pattern $p_1 \times p_2$ is simply c, while we reduce the usage of p_1 and p_2 by c. Hence $usg_{new}(p^*) = c$ and $usg_{new}(p_1) = usg_{old}(p_1) - c$, $usg_{new}(p_2) = usg_{old}(p_1) - c$. If we extend a p_1 by more than one singleton c is simply the minimum count out of all extensions, we then subtract c from all extensions. The frequency of how often events of a generalization are used we adjust proportionally to the pattern frequency. That is for each event $e \in fl(\alpha)$ where generalization α in the patterns p_1, respectively p_2 we adjust the usage to $usg_{new}^{p_1}(e) = usg_{old}^{p_1}(e) - \frac{c\ usg_{old}^{p_1}(e)}{usg_{old}(p_1)}$. Usage of same e in the new pattern p^* is $usg_{new}^{p^*}(e) = \frac{c\ usg_{old}^{p_1}(e)}{usg_{old}(p_1)}$.

Case 2 (New pattern, extended with Generalization): Next, we consider the case where we extend a pattern with an existing or newly formed generalization α. To estimate the frequency of the patterns, we can treat this case analog to Case 1, c is now just the sum of all counts of events $e \in fl(\alpha)$. The usage of $e \in \alpha$ is set to the count of e, whereas existing generalizations are adjusted as in Case 1.

Algorithm 7: PruneGen

```
input  : pattern $p$, generalization extensions $\Omega_g^\oplus$, cover $C$
output : pruned generalization extensions $\Omega_g^\oplus$, updated cover $C$
1 forall $(\alpha_{id}, R) \in \Omega_g^\oplus$ do in order of increasing $usg(e)$
2     if $\Delta\bar{L}(p', \Omega_g^\oplus \setminus (\alpha_{id}, R)) > \Delta\bar{L}(p', \Omega_g^\oplus)$ then
3         $\Omega_g^\oplus \leftarrow \Omega_g^\oplus \setminus (\alpha_{id}, R)$
4 return $\Omega_g^\oplus, C$
```

Case 3 (Extending an existing Generalization): Extending an existing generalization with new elements does effect all patterns that use this generalization. The effects are hence not contained to the pattern we refine. First we just consider p^* which contains a generalization α which we extend with event e. We increase the the frequency of pattern p^* by the number of counts of e. Of course we reduce the usage of e by the same amount. The estimated usage of event e in p^* is naturally also the same.

To estimate the usage of other patterns p' that contain α. We search for all windows of p' and assume that all *new* windows are used. That is all windows that did not match p^* before the extension of α with e. From the windows we can directly infer the respective frequency, which we use as a usage estimation.

A.4 Pattern search

In the main part of the paper, we describe the main search procedure, in this section we will expand on that explanation, providing more details.

Generalization pruning. We show pseudocode in Algorithm 7. When we initially build new generalization and extend existing ones we estimate the usage of the individual elements in the generalization. Once we have actually computed a cover we have actual usage counts. Hence we test, for each added event e, if under these counts we still get a gain when adding e.

Refine Interleaving. Before adding pattern p to our model, we search within the gaps of p for possible generalization. We provide pseudocode in Algorithm 8. At this point, we have a cover C that includes pattern p. We count for all gaps of pattern p how frequent each event is in each respective gap. The frequency of all events between the $i-1$ and the i^{th} event in pattern p is given by $c_i = counts[i]$. We refine one gap at a time, we start with the gap with the most frequent event. Per gap, we test all events in order of frequency, if we add a second event we create a new generalized event that matches the first and second event, analog for the third, forth, etc. event.

Simplify. The Simplify algorithm simply calls Merge and Flatten until convergence. This algorithm is applicable to any kind of pattern set P. The required generalized alphabet Ω_g is simply the empty set, and the cover C is computed using GreedyCover.

Algorithm 8: RefineInterleaving

```
input  : pattern $p$, generalization extensions $\Omega_g^{\oplus}$, cover $C$
output : refined pattern $p'$, extended generalization
         extensions $\Omega_g^{\oplus}$, updated cover $C$
1  for $i = 0; i < |p| - 1; i++$ do
2      $counts[i] \leftarrow$ get frequencies for all events between $p[i]$
         and $p[i+1]$
3  sort $counts$ by max $counts[i]$
4  forall $c \in counts$ do
5      sort $c$ by frequncy in decending order
6      forall $e \in c$ do
7          $p', \Omega_g^{\oplus\prime} \leftarrow$ extend $p$ and $\Omega_g^{\oplus}$ with $e$
8          if $\Delta \bar{L}(p', \Omega_g^{\oplus\prime}) > 0$ then
9              $p, \Omega_g^{\oplus} \leftarrow p', \Omega_g^{\oplus\prime}$
10 return $p, \Omega_g^{\oplus}, C$
```

Algorithm 9: Simplify

```
input  : Pattern set $P$ and generalization $\Omega_g$, cover $C$
output : Pattern set $P$ and generalization $\Omega_g$, cover $C$
1  while $P$ changes do
2      $P, \Omega_g, C \leftarrow$ Merge$(P, \Omega_g, C)$
3      $P, \Omega_g \leftarrow$ Flatten$(P, \Omega_g)$
4  return $P, \Omega_g, C$
```

Algorithm 10: Prune

```
input  : Pattern set $P$ and generalization $\Omega_g$, cover $C$
output : Pattern set $P$ and generalization $\Omega_g$, cover $C$
1  forall $p \in P$ do
2      if $L(D, P) > L(D, P \setminus \{p\})$ then
3          $P \leftarrow P \setminus \{p\}$
4          update $C$ and $\Omega_g$ accordingly
5  return $P, \Omega_g, C$
```

Pruning. The pruning step is only done once, as a final step. We test for each pattern if removing it improves the model. If it does we remove it otherwise we keep it.

Implementation Details. After we test 100 candidates back-to-back without any actual gain we break the current search loop, i.e. move on to the next iteration. In preliminary experiments, this did not have an effect on the results.

A.5 Dependence structure

When adding a pattern to our model we compute its dependency structure. Meaning we choose for each generalized event that earlier generalization that minimizes the specification cost for this generalization. This is, essentially, equivalent to picking that generalization with the lowest conditional entropy. More precisely when the penalty of the prequential encoding is ignored the encoding cost is equivalent to the conditional entropy. The specification cost of one generalization without the epsilon. $\sum_{i=1}^{usage(p)} - \log \left(\frac{usg(e|d)}{\sum_{c \in fl(\alpha)} usg(c|d)} \right)$ The inner part is just the probability of $p(e|d)$. Using Bayes' theorem we can transform that into $\frac{p(e,d)}{p(d)}$, summing over all usages is equivalent to summing over all combinations each weighted by its joined probability. Making it equivalent to the conditional entropy, defined as $H(Y \mid X) = -\sum_{x \in X, y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)}$. In practice however we directly compute the bits needed by the prequential encoding, which is asymptotical equivalent to picking that generalization with the lowest entropy. In practice it can be cheaper to encode a specifications of a generalization independently of any previous generalization, hence we also allow that.

B EXPERIMENTS

In this section we give further details about the experiment setup, evaluation and additional examples reported by Flock.

B.1 Experiment Setup

Skopus reports the top-k patterns, where k is a hyperparameter. In preliminary experiments we tested setting k to the number of all reported instances of Flock. The number instances of a generalized pattern p are the unique instances that match p. In the setting we considered these where between 100 and 200 instances. Skopus did not terminate with 24h therefore we set k to 50 and limited the pattern length to 10.

The word2vec architecture [20] has been shown to be good at training an embedding where items that occur in similar contexts are placed close to each other, while originally proposed for text the architecture lends itself to arbitrary sequences over a discrete alphabet. In short, we train an embedding into 4 dimensions using the word2vec architecture [20] with a window size of 10.

To extract generalizations from this embedding, we use DBScan with a maximum distance of $\epsilon = 0.2$ and with a minimum number of samples of 3. Next, we replace elements in D contained within a cluster with a new generalized event, representing that cluster. Finally, we apply Sqs on the modified dataset. The groups found by DBScan we consider our generalizations and the pattern found by Sqs are our patterns. We tuned the hyperparameters to produce clusters as close as possible to the true planted generalizations while also trying to avoid spurious clusters. We did this on synthetic generated data where each generalization was only used within one pattern.

The *ECG* dataset is based on the first record (id 300.1) of the MIT-BIH ST Change Database.[6] We subsampled the record, replacing each 5 subsequent values with their average, and transformed the result into a relative sequence by replacing each value with the difference to the previous value. Finally, using SAX [18] we discretize the sequence to 200 events.

On the datasets *ECG*, *Short-ECG*, *Moby*, and *JMLR* we run Flock with gap parameter $n = 2$, in B.3 we show that small values of n works better for datasets with a low amount of structure.

B.2 Evaluation Metric

As metric to evaluate success we consider a mapping between planted and discovered patterns [4]. Where we map each reported

[6] https://physionet.org/content/stdb/1.0.0/

	Patterns	Generalizations
BPI-2015	• send conf. receipt, enter senddate ack., δ, forward to the competent authority, β, γ, enter senddate procedure conf.	α = {forward to the competent authority, enter senddate ack. }
	• send confirmation receipt, α, β	β = {regular procedure without MER, start WABOprocedure }
		γ = {regular procedure without MER, send procedure confirmation }
		δ = {OLO messaging active, send conf. receipt, application received, term. on requ., appl. is stakeholder, no permit or only notification }
Moby	• funni, sporti, gami, jesti, joki, hoki, poki, lad	α = {call, year, never, long, ship, more, minut, centuri, period }
	• α, ago	
JMLR	• support, vector, machin, svm	α = {blind, nonlinear, specialis }
	• princip, compon, analysi	β = {matrix, contrast, neumann, classic, displai, shannon, intercept, distanc, bregman, near, leibler, liebler }
	• α, sourc, separ	
	• β, diverg	

Table 2: Selection of found patterns on real-world datasets. We see that Flock discovers rich generalization and patterns that use multiple generalization. On the *BPI-2015* dataset, we see two patterns using the same generalization.

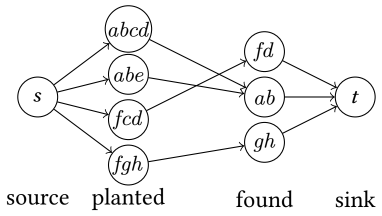

Figure 4: Toy example of how we match discovered to planted patterns. A discovered pattern ab will be matched to either $abcd$ or abe, but not both; maximizing the flow gives us the number of recovered patterns.

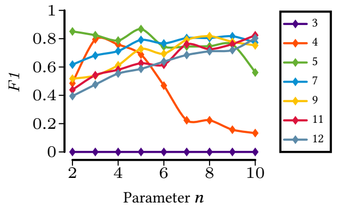

Figure 5: Parameter sensitivity analysis: On seven datasets with pattern lengths from 3 to 12, each pattern includes two generalization. We see Flock is robust against the parameter choice. The dataset with a pattern length of 4 stands out, with half the events being generalization true patterns become hard to distinguish from random correlations.

pattern to at most one planted pattern and to each planted pattern to at most one reported pattern. To find the maximum number of pairs we reformulate the problem as a flow network optimization problem. We connect each planted pattern to our source and each found pattern to the sink. We connect a planted to a found pattern if the found pattern is a subsequence of the planted one or vice versa. The capacity of all links is set to one. We then compute the maximum flow [6] which maximizes the number of planted, found pairs. This setup ensures that we match at most one found pattern to a planted pattern and vice versa. In Figure 4 we give a toy example of such a flow network. We do the analog with generalizations.

B.3 Parameter Analysis

Flock comes with one user parameter n the factor of how many gaps we allow relative to the pattern length. We consider a setting with patterns of different lengths, 3 to 12, each pattern contains two generalization. We report the $F1$ of Flock given the respective parameter n.

				Flock		Sqs	Squish
Dataset	$\lvert D\rvert$	$\lvert\lvert D\rvert\rvert$	$\lvert\bar{S}\rvert$	$\%L$	t	$\%L$	$\%L$
ECG	1	107395	107395	97.2	17k	96.7	98.9
Short-ECG	1	3384	3384	98.2	1	98.3	98.7
BPI-2015	1199	51867	43.36	79.1	43k	60.7	63.2
Rolling Mill	1000	51390	51.39	63.2	1k	49.9	52.9
Moby	1	105719	105719	99.3	0.6k	99.3	99.4
JMLR	788	75646	96	96.7	0.7k	96.6	97.6

Table 3: Extended results on real datasets. We given number of sequences, $|D|$, total number of events $||D||$, and average sequence length $|\bar{S}|$. For each method we give the relative length under our model. For Flock we provide the runtime in seconds.

B.4 Real-World Pattern Examples

In Table 2 we show a selection of patterns and generalization reported by Flock on real-world datasets. Note, the rich generalization β of the *JMLR* dataset was discovered with the default gap parameter of $n = 10$.

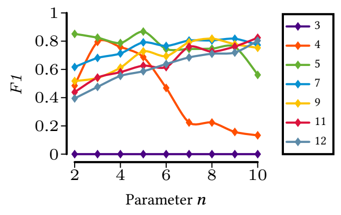

Figure 6: Ablation study on synthetic with subroutines turned off. We observe that without RefineInterleaving and Simplify the $F1$ score decreases by a large margin.

B.5 Ablation Study

Flock consists of several subroutines, we evaluate the impact of disabling RefineInterleaving and Simplify. We evaluate the performance on synthetic data, we use the same setting as for the experiment shown in Figure 2(b), with 400 planted pattern instances. We report the $F1$ score. The results show that RefineInterleaving and Simplify provide key functionality that greatly improves performance.