Sequence Partitioning for Process Mining with Unlabeled Event Logs

Michał Walicki[a,1], Diogo R. Ferreira[b,*]

[a] *Institute of Informatics, University of Bergen, Norway*
[b] *IST – Technical University of Lisbon, Portugal*

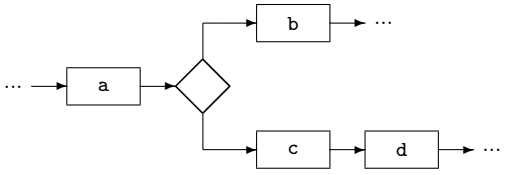
Abstract

Finding the case id in unlabeled event logs is arguably one of the hardest challenges in process mining research. While this problem has been addressed with greedy approaches, these usually converge to sub-optimal solutions. In this work, we describe an approach to perform complete search over the search space. We formulate the problem as a matter of finding the minimal set of patterns contained in a sequence, where patterns can be interleaved but do not have repeating symbols. This represents a new problem that has not been previously addressed in the literature, with NP-hard variants and conjectured NP-completeness. We solve it in a stepwise manner, by generating and verifying a list of candidate solutions. The techniques, introduced to address various subtasks, can be applied independently for solving more specific problems. The approach has been implemented and applied in a case study with real data from a business process supported in a software application.

Keywords: Process mining, sequential pattern mining, sequence partitioning, combinatorics on words.

1. Introduction

A business process is a structured set of activities which can be instantiated multiple times and whose execution is assumed to be recorded in an event log. The goal of process mining [1, 2] is to rediscover the process model from the runtime behavior of process instances recorded in the event log. The event log contains a sequence of entries in the form (*case id*, *task id*) where *case id* identifies the process instance and *task id* specifies the task that has been performed. The sequence of tasks recorded during the execution of a process instance is called a *workflow trace* [3]. Since several process instances may be active simultaneously, the traces for different instances may overlap in time, as illustrated in Figure 1.



case id	*task id*
1	a
2	a
2	c
1	b
3	a
...	...

```
   aacbabadaccddacd
1 a..b............
2 .ac....d........
3 ....ab..........
4 ......a..c..d...
5 ........a.cd....
...
```

Figure 1: An excerpt of a generating process, its event log and the workflow traces.

For the purpose of process mining, the overlapping of workflow traces is not a problem, since each event is associated with the *case id* which clearly identifies the process instance that the event belongs to. A wide range of process mining techniques [4] has been devised to discover process models from such event logs. However, while these event logs can be easily obtained from workflow and case-handling systems, in other

*Corresponding author e-mail: diogo.ferreira@ist.utl.pt
[1] This work was performed while the first author was visiting the Technical University of Lisbon.

This is a preprint version of an article published in Data & Knowledge Engineering, vol.70, no.10, pp.821–841, October 2011.

applications that are not fully process-aware it may become difficult to retrieve event data in that form. If the case id attribute is missing, then the event log becomes an unlabeled sequence of events where it is unknown whether any two events belong to the same process instance or not.

The problem we address in this paper is how to recover the workflow traces from an unlabeled sequence of events. A sample sequence is illustrated in the top-right corner of Figure 1. This sequence of symbols is the result of several workflow traces becoming interleaved in the event log. But since workflow traces essentially repeat the sequential patterns of the business process, in principle it should be possible to identify these repeating patterns in the unlabeled sequence. As in Figure 1, and for reasons to be explained below, we consider patterns without repeating symbols. In addition, for a subsequence to qualify as a pattern, it should have at least 2 symbols and at least 2 occurrences in the sequence.

Under these conditions, the sequence *aacbabadaccddacd* of Figure 1 admits solutions with 2, 3 and 4 patterns. Table 1 lists some of these solutions. Each solution contains a set of patterns, and each pattern has its own multiplicity (number of repetitions). This is written as $\langle p_1^{n_1}, ..., p_k^{n_k} \rangle$, where each p_i is a pattern (sequence of symbols) and n_i is the number of occurrences of pattern p_i in the solution. The "true" solution, i.e. the one that corresponds to the generating patterns, can be found in the first column. Here it can be seen that the sequence admits an alternative solution with 2 patterns. There are many more solutions with 3 and 4 patterns, but in general we will be interested in solutions with a minimal set of patterns, as these can provide a more compact representation of the generating process.

Solutions with 2 patterns	Solutions with 3 patterns	Solutions with 4 patterns
$\langle acd^4, ba^2 \rangle$	$\langle ad^4, bc^2, ca^2 \rangle$	$\langle ad^2, ba^2, ca^2, dc^2 \rangle$
$\langle ab^2, acd^4 \rangle$	$\langle ab^2, acd^2, cda^2 \rangle$	$\langle ac^2, ba^2, cd^2, da^2 \rangle$
	$\langle abdc^2, ad^2, ca^2 \rangle$	$\langle ab^2, ac^2, ad^2, cd^2 \rangle$
	...	...
(2 solutions)	(44 solutions)	(12 solutions)

Table 1: Some of the solutions for the sequence *aacbabadaccddacd*.

1.1. Related work

While there is a host of problems and techniques in the area of sequential pattern mining [5, 6, 7, 8, 9, 10, 11, 12], the present problem does not seem to have been investigated before in this general form. There are some variants completely unrelated to ours, for instance, of partitioning *numerical* sequences into substrings, minimizing some cost function, e.g. [13]. In closer variants, the discovery of sequential patterns is usually bound to constraints [14] such as time windows [15, 16], regular expressions [17], or some domain-specific knowledge [18]. There are also approaches that focus on mining specific patterns or on counting their occurrences [19] but this represents just one of the subtasks in finding the minimal set of patterns that cover a given sequence.

Related problems have been formulated in other areas but they do not seem to match exactly the problem considered here. For instance, in the field of combinatorics on words there is a related problem of how many subsequences (and how long) are needed in order to determine a given sequence uniquely [20]. This has a different focus from our problem, since we want to find all possible sets of patterns in order to pick the minimal solution. An interesting result was reported in [21], allowing to decide if a given regular language can be obtained as the shuffle product of two other regular languages. Unfortunately, as we are dealing with a one-word language (a single sequence), it is obvious that any such language can be obtained by shuffling any of its non-empty subsequences. Viewed as a special case of the problem addressed in [21], our instance is trivial but the algorithm presented there, addressing a related but different problem, does not seem to help solving ours.

Another field posing apparently related questions is bioinformatics. However, the main difference here concerns the fact that every instance of our problem contains a single sequence to be covered completely by a set of patterns, while bioinformatics typically searches for common fragments shared by several sequences. The techniques are therefore quite different and do not seem to be directly transferable.

1.2. Previous work

Among works most closely related to the current one, we can mention [22] which introduced an Expectation-Maximization approach to estimate a Markov model from an unlabeled event log. The approach includes a greedy algorithm that can be used to assign a case id to each event. This greedy algorithm finds a single solution, which is often a local maximum of the likelihood function. Also, the resulting number of patterns depends on the Markov model itself and on the way the greedy algorithm decides where to terminate the occurrence of one pattern and begin another occurrence of the same or a different pattern.

Here, we are interested in traversing the complete search space in order to enumerate all possible solutions with a given number of patterns. We are also concerned with the concept of *minimum description length* (MDL) [23], so we define the optimal solution(s) as the one(s) with a minimal set of patterns. As in the example of Table 1, it is easier to describe the sequence based on 2 patterns rather than 3 or 4 patterns. The principle of MDL has already been used in process mining to evaluate the quality of mined models [24]. Here we use it as a guiding principle while looking for solutions across the entire search space.

A similar aim was proposed in [25], which introduced a trie representation which we apply also in our solution. The approach of [25], however, using Knuth's Algorithm X, is hardly scalable to sequences longer than 30-40 symbols, while the new techniques in the present paper have been applied successfully to sequences more than 10 times longer.

1.3. Practical aim

The practical aim of this study is to allow process mining of realistic examples where the length of the input sequence would have from a few hundred to a few thousand events, with not more than 10-15 different events. In such scenarios one could be looking for a partition of the input sequence into a set of up to 5-10 patterns although, in many cases, fewer patterns will suffice. This represents a huge computational effort as any approach to this problem can be expected to scale exponentially with the length of the input sequence or, at least, of the size of alphabet. This may be the reason why the problem remained neglected, in spite of its relevance for practical applications. Our solution shows that even if it is NP-hard (which remains an open hypothesis), the exponent of its complexity involves only the size of the alphabet. Hence, the algorithm can be expected to scale reasonably to longer sequences, as long as the size of the alphabet remains relatively small. This is, in fact, the case in practice, and we will see that other theoretical worst cases do not occur under usual circumstances. An implementation of the proposed algorithm has been used to analyze data from real-life problems and the results are reported in the case study section.

1.4. Structure of the paper

Section 2 introduces the problem and notational conventions, and presents the main algorithm whose refinement leads to the final solution. It consists of three subroutines described, respectively, in the following Sections 3, 4 and 5. The correctness of the introduced algorithms is proved and upper bounds on the worst case complexity are analyzed. Section 6 adds a few remarks about the optimizations used in the actual implementation and comments on possible adjustments of the algorithm for handling other versions of the problem. Section 7 presents a case study performed on a dataset obtained from a real-world business process, and analyzed using our implementation, and Section 8 concludes the paper. All proofs are gathered in the Appendix.

2. Preliminaries

Subsection 2.1 introduces some notational conventions. Of particular importance is the concept of *multiset*, which is used throughout the paper. Subsection 2.2 formulates precisely the problem and its variants, and 2.3 sketches the algorithm forming the basis for the approach developed in subsequent sections.

2.1. Notation and conventions

$\mathbf{N}$ denotes the natural numbers and $\mathbf{N}_+$ the positive ones.

We assume that a sequence has an alphabet Σ with a (relatively small) number of symbols $E = |\Sigma|$. $|X|$ denotes the cardinality of X when it is a set and the length of X when it is a sequence. Typically, the considered alphabet is relative to a given sequence S and contains only the symbols $\Sigma(S)$ occurring in S. A pattern (over Σ) is any sequence $p \in \Sigma^+$ and it occurs in another sequence S if S contains p as a *subsequence* (not necessarily as a substring).[2]

Two occurrences of a pattern are disjoint (DO = disjoint occurrence) if they are disjoint subsequences. For instance, the sequence $S = abacabac$ can be seen as consisting of two DOs of $abac$, but it can also be seen as consisting of two DOs of ab and two of ac or else of two DOs of ac and two of ba. We denote these respective facts by $S \in \bigotimes\langle abac^2\rangle$ or $S \in \bigotimes\langle ab^2, ac^2\rangle$ or $S \in \bigotimes\langle ac^2, ba^2\rangle$, where $\langle p_1^{n_1}, ..., p_k^{n_k}\rangle$ denotes the multiset of patterns $p_1, ..., p_k$, each p_i with the respective number n_i of occurrences, and $\bigotimes$ denotes their shuffle product (all interleavings of the argument patterns).

For any sequence S (examples will be given for $T = abacbc$ with $|T| = 6$ and $\Sigma(T) = \{a, b, c\}$) we define:

1. S_i with $1 \leq i \leq |S|$ is the symbol at position i in $S = S_1S_2...S_{|S|}$. Negative indexing is used to refer to sequence positions counted from the end, i.e. S_{-1} denotes the last element, S_{-2} the one before last, etc.

2. $S[i : j]$ is the substring $S_iS_{i+1}...S_{j-1}S_j$ and $S_i = S[i : i]$ (example: $T[1 : 3] = aba$, $T[4 : 4] = T_4 = c$). For $j > i$, $S[j : i]$ is the empty sequence ϵ.

3. $Q \sqsubseteq S$ denotes the *subsequence* partial order. The set of subsequences of S is $\text{Subs}(S) = \{X \mid X \sqsubseteq S\}$ (example: $\{aba, abc, abb, bac\} \subset \text{Subs}(T)$). $Q \preceq S$ denotes the *prefix* partial order, and the set of prefixes of S is $\text{Pref}(S) = \{X \mid X \preceq S\}$ (example: $aba \in \text{Pref}(T)$ while $abc \notin \text{Pref}(T)$)

4. For $a \in \Sigma : occ(a, S) = \{i \mid S_i = a\}$ is the set of occurrences and $noc(a, S) = |occ(a, S)|$ is the number of occurrences of symbol a in sequence S (example: $occ(b, T) = \{2, 5\}$ and $noc(b, T) = 2$)

5. An occurrence of a pattern p in S is any subsequence $y \sqsubseteq S$ where $p = y$ (example: $T_1T_2T_6$ and $T_3T_5T_6$ are two, non-disjoint, occurrences of $p = abc$ in T). When referring to a single pattern p, we use p_i to represent the i-th symbol of the pattern. When referring to a set of patterns $\{p_1, p_2, ..., p_k\}$ we use p_i to represent a pattern in that set.

6. For a pattern p and a sequence S, we denote by $noc(p, S)$ the maximal number of disjoint occurrences (MDOs) of p in S (example: $noc(abc, T) = 2$ since there are two DOs of abc in T: $T_1T_2T_4$ and $T_3T_5T_6$; however, note that it is possible to choose less than 2 occurrences of abc in T if one picks $T_1T_2T_6$)

7. A multiset M over a set $\mathbb{S}$ is a function $M \in \mathbf{N}^{\mathbb{S}}$ that assigns a number to each element of $p \in \mathbb{S}$, called its *multiplicity* in M. We denote the multiplicity of an $p \in \mathbb{S}$ in M by $mlt(p, M)$, and write $p \in M$ whenever $mlt(p, M) > 0$ and $p \notin M$ whenever $mlt(p, M) = 0$. The empty multiset, denoted $\langle\rangle$, is one with $\forall_{p\in\mathbb{S}} : p \notin \langle\rangle$, i.e., $mlt(p, \langle\rangle) = 0$.

A multiset M is usually written explicitly as $\langle p_1^{n_1}, ..., p_k^{n_k}\rangle$ where $n_i = mlt(p_i, M)$ (example: in the multiset $M = \langle ab^2, bc^3\rangle$ we have $mlt(ab, M) = 2$ and $mlt(bc, M) = 3$).

8. For a multiset M the shuffle operator $\bigotimes M$ returns the set of all sequences that can be produced by interleaving the elements of M, where each element $p \in M$ enters as $mlt(p, M)$ copies of pattern p (examples: $\bigotimes\langle ab^1, bc^1\rangle = \{abbc, bcab, bacb, abcb, babc\}$ and $\bigotimes\langle ab^2\rangle = \{abab, aabb\}$). A multiset M covers the sequence S if $S \in \bigotimes M$. In this case, we say that M is a *solution* for S. (Examples: $T \in \bigotimes\langle abc^2\rangle$, $T \in \bigotimes\langle abc^1, acb^1\rangle$, $T \in \bigotimes\langle ab^1, ba^1, c^2\rangle$, $T \in \bigotimes\langle ac^1, b^1, ba^1, c^1\rangle$, etc.) We let $\bigotimes\langle\rangle = \{\epsilon\}$.

9. The operations $M \oplus M'$ and $M \ominus M'$ denote multiset addition and subtraction, whose effect is, for every $p \in \mathbb{S} : mlt(p, M \oplus M') = mlt(p, M) + mlt(p, M')$ and $mlt(p, M \ominus M') = mlt(p, M) \dot{-} mlt(p, M')$ where

[2]We will consider almost exclusively more specific patterns consisting of at least two symbols and with all symbols distinct.

$m \dot{-} n$ is the usual subtraction when $m \geq n$ and 0 otherwise (examples: $\langle ab^2, bc^3\rangle \oplus \langle bc^2\rangle = \langle ab^2, bc^5\rangle$ and $\langle ab^2, bc^3\rangle \ominus \langle ab^4\rangle = \langle bc^3\rangle$).

10. For a multiset M, $\mathrm{Symb}(M)$ is the multiset with the number of occurrences of each symbol in any member of $\bigotimes M$, i.e. if a is a symbol then $mlt(a, \mathrm{Symb}(M)) = \sum_{p \in M} noc(a, p) \cdot mlt(p, M)$ (example: $\mathrm{Symb}(\langle ab^2, bc^3\rangle) = \langle a^2, b^5, c^3\rangle$). For a sequence S, $\mathrm{Symb}(S)$ is the multiset with the number of occurrences of each symbol in S (example: $\mathrm{Symb}(T) = \langle a^2, b^2, c^2\rangle$)

11. A multiset of sets of symbols, usually denoted as $\langle \underline{p}_1^{n_1}, ..., \underline{p}_k^{n_k}\rangle$, is one where each element $\underline{p}_i$ (e.g. $\underline{p}_i = \{a, b, c\}$) represents any pattern that can be built as a permutation of the given symbols (e.g. one of $\{abc, acb, cab, cba, bca, bac\}$). When the context is clear, the shorthand notation $\underline{p}_i = \{abc\}$ is used.

2.2. The problem and its variants

The most general formulation of the sequence partitioning problem is as follows:

Problem 2.1. General Sequence Partitioning, GSP
INPUT: a sequence S and a set of patterns P over $\Sigma(S)$
OUTPUT: partitions of S into a minimal number of patterns $X \subseteq P$.

In practice, the set P is often given implicitly and in the instances of the problem addressed in this paper, the user does not specify this set.

A straightforward, inefficient solution could be as follows. Identify various DOs of all patterns, combine them in all possible ways, and see if any of such combinations covers the sequence exactly. E.g.:

	a	b	a	c	a	b	a	c
ab	+	+			+	+		
	+	+	+			+		
abc	+	+	+	+		+		+
ac	+		+	+				+
			+	+			+	+

The first two rows give two distinct DOs of the pattern ab, the third two DOs of abc and the last two, examples of DOs of ac. The first row, with two DOs of ab and the last one, with two of ac, give a minimal partition with the shown patterns. One recognizes this as an optimization version of the NP-complete exact set cover problem (ESC) [26].[3] But even to arrive at this representation, we have to first generate all DOs of various patterns. Already here one can encounter exponential complexity, as this requires listing not only distinct patterns but also their actual occurrences. E.g., in the sequence $aaabbbcccdddabcee \in \bigotimes\langle abcd^2, abce^2\rangle$, one can choose any 2 of the first 3 a's, any 2 of the first 3 b's and any 2 of the first 3 c's to match with the two d's. Then, any remaining abc can be matched with the e's. There are thus 3^3 choices of the DOs of $abcd$ or, equivalently, of $abce$. Generally, given a sequence S with length $n = |S|$, a pattern of length k may have $(\frac{n}{k})^k$ DOs in S. E.g., the sequence $S = x_1x_1x_2x_2...x_kx_k$ contains 2^k DOs of the pattern $x_1x_2...x_k$, with $k = \frac{n}{2}$. Requiring the list of all such possibilities adds an intractable dimension to the already difficult problem.

We therefore do not ask for the actual occurrences of patterns but only for their multiplicites. We also restrict the admissible patterns, which is independently motivated by the origins of the problem as described in the introduction. We consider only patterns with no repeated symbols, which have length at least 2 and which are true *patterns*, i.e. occur at least twice. This restriction leads to the following problem.

Problem 2.2. SP'
INPUT: a sequence S, patterns $P \subseteq \{p \in \Sigma(S)^+ \mid |p| > 1,\ noc(p, S) > 1,\ \forall_{1 \leq i < j \leq |p|} : p_i \neq p_j\}$
OUTPUT: partitions of S into a minimal number of patterns $X \subseteq P$

[3] ESC asks to cover an input set by selecting from a number of available subsets, so that each element in the input set is covered by exactly one subset.

This problem, asking only for the minimal number of patterns from P needed to partition S, is still NP-hard and, consequently, so are the more complex variants asking for actual patterns or their multiplicities. This can be shown, e.g., by a reduction from the edge-coloring problem (Fact 9.1 in Appendix), which is possible because SP' allows to choose freely a subset P of patterns to be considered. If we disallow that, the possibility of such a reduction is not obvious, and we obtain our main problem:

Problem 2.3. Sequence Partitioning, SP

INPUT: a sequence S
OUTPUT: partitions of S into a minimal number of patterns $X \subseteq P$
where $P = \{p \in \Sigma(S)^+ \mid |p| > 1,\ noc(p, S) > 1,\ \forall_{1 \le i < j \le |p|} : p_i \neq p_j\}$

The central feature of this instance, distinguishing it from the previous ones, is that the patterns P to be used are not specified (beyond their general format) but have to be mined from the sequence. This limits the flexibility in formulating the questions and gives a special case of SP', whose NP-hardness remains an open issue. We address the enumeration version of the problem, asking for the partitions specified by the used patterns and their multiplicities, but not for the positions at which different patterns occur. An important contribution of this work consists in obtaining the former without generating the latter.

We will approach the above problem by solving a more specific version, where the number of patterns to be used is part of the input.

Problem 2.4. Sequence k-Partitioning, SP[k]

INPUT: a sequence S, an integer $0 < k \le |S|/4$
OUTPUT: partitions (if any) of S into k patterns from P
where $P = \{p \in \Sigma(S)^+ \mid |p| > 1,\ noc(p, S) > 1,\ \forall_{1 \le i < j \le |p|} : p_i \neq p_j\}$

The limit $k \le |S|/4$ is due to the restriction to patterns of length at least 2 and with at least 2 occurrences. Our algorithm for SP is based on iterating a solution to SP[k] with increasing k until it produces a solution. The complexity analysis shows that the exponent involves the size $E = |\Sigma|$ of the alphabet (possibly, as the factor limiting the number of distinct patterns by $\sum_{i=1}^{i=E} \frac{E!}{i!}$). Although this may quickly become prohibitive, the fact that the alphabet is typically small as compared to the size of the sequence opens the possibility of analyzing non-trivial, real-life data sets. Section 7 provides an example of such analysis.[4]

2.3. The main algorithm

Consider a naive algorithm that tries to generate solutions to Problem 2.3 by running once through sequence S and generating all possible multisets at each position in the sequence. For example, for the sequence $T = abacbc$, at the first position one would start with $\langle a^1 \rangle$; at the second position, one would get either $\langle ab^1 \rangle$ or $\langle a^1, b^1 \rangle$; at the third position, the possibilities are $\langle a^1, ab^1 \rangle$, $\langle a^1, ba^1 \rangle$, and $\langle a^2, b^1 \rangle$; and so on. Once we reach the end of the sequence, the solutions (if any) can be found among the resulting set of multisets. Let $V(S, i)$ be the set of multisets at each position i in the sequence S. The building of $V(S, i)$ as the algorithm proceeds through the successive positions i of the sequence S is defined inductively as follows:

$$\begin{aligned} V(S,1) &= \{S_1\} \\ V(S,i) &= \textstyle\bigcup_{X \in V(S,i-1)} ch(X, S_i),\ \textit{where} \\ ch(X,a) &= \{(X \ominus \langle x^1 \rangle) \oplus \langle xa^1 \rangle \mid x \in X \ \wedge\ a \notin x\} \cup \{X \oplus \langle a^1 \rangle\} \end{aligned} \tag{2.5}$$

The number of multisets grows exponentially through the sequence, since each multiset X may originate as many as $|X| + 1$ children, and each child may contain more subsequences than its own parent, so the rate of children per parent also grows through the sequence. Most of these multisets are, of course, unnecessary and we will solve the problem by limiting their number. The algorithm (2.5) can be used very efficiently, for

[4] Since we are dealing with enumeration problem, the classical complexity analysis is not necessarily the most appropriate tool for evaluating the solution. Various proposals, e.g., [27, 28, 29, 30] have not yet resulted in a standard way to classify the complexity of such problems. We therefore do not pursue any detailed complexity analysis of the optimization and enumeration aspects.

instance, when we are only looking for DOs of some given patterns. When checking if the sequence S can be covered by DOs of a single pattern p, the following underlined modifications (collecting, at each point, only the encountered prefixes of p) make the algorithm linear. (A variation of this algorithm was used in [19] for determining $noc(p, S)$.)

$$\begin{array}{rcl} V(S,p,1) & = & \{S_1\} \underline{\cap \text{ Pref}(p)} \\ V(S,p,i) & = & \bigcup_{X \in V(S,p,i-1)} ch(X,p,S_i), \ \textit{where} \\ ch(X,p,a) & = & \big(\{(X \ominus \langle x^1 \rangle) \oplus \langle xa^1 \rangle \mid x \in X\} \cup \{X \oplus \langle a^1 \rangle\}\big) \underline{\cap \text{ Pref}(p)} \end{array} \tag{2.6}$$

So, while efficient for checking DOs of specified patterns, the algorithm is heavily exponential for generating the relevant patterns. Our main Algorithm 2.7 utilizes an optimized version of (2.6) but precedes it by an efficient generation of the potential candidates. It returns the set of all solutions for the input sequence, if there is any, and the empty set otherwise.

Algorithm 2.7 MinCover(S:Seq)

Output: all minimal solutions, if any such exists, and $\varnothing$ otherwise

1: $T := BuildTrie(S)$ ..... // Algorithm 3.5
2: **for each** $k := 1...|S|/4$ **do**
3: $\quad Cand := Candidates(S,k)$ ..... // Algorithm 4.3
4: $\quad R := \varnothing$
5: $\quad$ **for each** $M \in Cand$ **do**
6: $\quad\quad$ **if** $\langle\rangle \in Verify(S,M)$ **then** $R := R \cup \{M\}$ ..... // Algorithm 5.1
7: $\quad$ **if** $R \neq \varnothing$ **then return** R
8: **return** $\varnothing$

The algorithm begins by building a trie T, which contains all patterns occurring in S, and allows to identify the number of DOs of each pattern and symbol. The rest of the algorithm uses T and does not require the use of S except to verify candidate solutions. Algorithm 4.3 generates the potential candidates for solutions using only T. Using a simple algorithm for solving systems of linear equations, it excludes a significant number of irrelevant combinations of patterns. This generation is iterated starting with only 1 candidate pattern and increasing the number of patterns when the lower number does not yield any solution. The iteration stops thus at the lowest number of patterns. Given a candidate multiset M, an optimized version of *Verify*, Algorithm 5.1, decides if M covers S.

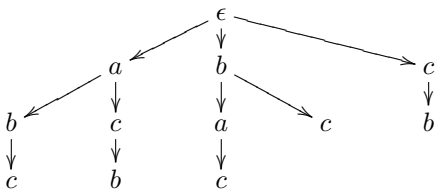
Each of the three main components of Algorithm 2.7 is of independent interest as it can be applied for solving the more specific, yet frequently encountered, subproblem. Section 3 describes the trie representing the sequence and the algorithm to build it. Section 4 describes the algorithm to generate the candidate multisets and to limit their number. Verification of the candidates, yielding the final solutions, is described in Section 5.

3. The trie – the number of DOs of patterns

The trie, as used here, was introduced in [25] but we present it to make the paper self-contained. It is built for an easy identification of patterns and the number of their disjoint occurrences in the sequence. It stores also detailed information about the actual occurrences of every pattern. This information is of potential use for other purposes, but our algorithms do not utilize it in full, since it is the main source of complexity.

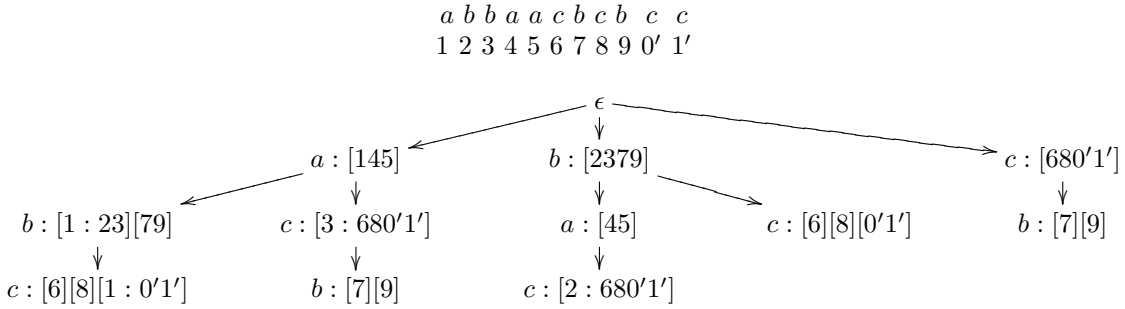
The trie is implemented as a tree with nodes labeled by symbols (siblings are always labeled by different symbols). Each path from the root identifies a unique pattern. The trie can be regarded as a dictionary with patterns as keys and nodes accessed by means of the pattern key (e.g., $T[ba]$, starting at the root ϵ, finds first the node b and then its child a.)

Example 3.1. *For the sequence $S = abbaacbcbcc$, the general structure of the trie T is as follows:*



As data, each node contains a list of relevant occurrences of its symbol. An edge from parent to child $(x : X) \to (y : Y)$ means that for every occurrence of y in the list Y there is a preceding occurrence of x in the list X. This information is stored in more detail, as explained using the following example.

Example 3.2 (3.1 contd).



The list of symbol occurrences at each node (except the children of the root, which identify occurrences of single symbols) is divided into sublists, e.g. $T[bc]$ into $[6][8][0'1']$. Such a cut of a child list $[Y_1...Y_j][Y_{j+1}...Y_l]$ happens after every maximal $1 \leq j \leq l$ such that, in the parent list $X_1X_2...X_k$, there is some X_i with $X_i < Y_j < X_{i+1}$. For example, the cut after 6 at $T[bc]$ appears because of ...37... at $T[b]$, with $3 < 6 < 7$. The meaning of such a cut is that a choice of a child symbol y occurring up to position Y_j requires a choice of the parent symbol x occurring up to position X_i. E.g., choosing $c : 6$ and trying to complete (backwards) the pattern bc, one must choose either $b : 2$ or $b : 3$, but not $b : 7$.

The numeral preceding such a sublist, as in $[2 : 680'1']$, indicates the upper bound on the number of DOs of the pattern, by telling the number of matching earlier occurrences of the preceding symbol. The $[2 : 680'1']$ at $T[bac]$ says that there are at most two occurrences of bac; this is due to the fact that there are only 2 occurrences of ba before position 6. The lack of such a numeral means implicitly that it is equal to the length of the subsequence, i.e., $[1 : 23][79]$ is the same as $[1 : 23][2 : 79]$.[5]

Each node $T[p]$, containing the last symbol of pattern p, stores such a list of sublists, each with the relevant occurrences and admissible number of choices. The number of DOs of the pattern p in the sequence S, $noc(p, S)$, is then the sum of the numerals at the node $T[p]$, e.g., there are 3 DOs of ab and 4 DOs of bc, as indicated by the sum of the (implicit) numerals in each sublist. Correctness of this claim is established by the following fact, characterizing sequences consisting of a fixed number of DOs of a pattern.

Fact 3.3. *Let $p = p_1...p_z$ be a pattern with all symbols distinct, and let S be a shuffle of m copies of these symbols, i.e. $S \in \bigotimes\langle p_1^m, ..., p_z^m\rangle$. Then $S \in \bigotimes\langle p^m\rangle$ iff $\mathsf{P}(S, p)$, where the latter is defined as $\mathsf{P}(S, p) \Leftrightarrow \forall_{1\leq k\leq |S|}\ \forall_{1\leq i<z} : noc(p_i, S[1 : k]) \geq noc(p_{i+1}, S[1 : k])$.*

[5]The choice need not be limited at the closing bracket, since it can be always made further to the right. E.g., having $x : [145]$ and $y : [1 : 23][2 : 678]$ in a pattern xy, would allow choice of the last three y's ($y : 6, 7, 8$) since any occurrence of the preceding symbol x that matches occurrence 2 or 3 of y will also match any later occurrence of y. On the other hand, the closing bracket says that below the last position there is no more than the indicated number of DOs of the pattern. For example, the choice of $y : 2, 3$ does not allow to build two occurrences of xy, since there is only one preceding x.

In words, an $S \in \bigotimes\langle p_1^m, ..., p_z^m\rangle$ consists of m DOs of $p = p_1...p_z$ iff for every $1 \leq i < z$ and $1 \leq n \leq m$, the n-th occurrence of the i-th symbol of p precedes the n-th occurrence of the $(i+1)$-th symbol of p.

Example 3.4. *For the following two sequences $S, R \in \bigotimes\langle a^4, b^4\rangle$, we verify the condition by counting, for each symbol, the number of its occurrences up to each position in the sequence:*

$S:$	a	a	b	a	b	b	a	b
$a:$	1	2	2	3	3	3	4	4
$b:$	0	0	1	1	2	3	3	4

$R:$	a	a	b	b	b	a	a	b
$a:$	1	2	2	2	2	3	4	4
$b:$	0	0	1	2	3	3	3	4

In S, the number of b's never exceeds the number of a's, so Fact 3.3 allows us to conclude that $S \in \bigotimes\langle ab^4\rangle$. In R, at position 5 we have 3 b's and 2 a's, so $R \notin \bigotimes\langle ab^4\rangle$. Indeed, $noc(ab, R) = 3$.

For an arbitrary sequence Q, Fact 3.3 means that the maximal number of DOs of a pattern p is determined by a subsequence $S \sqsubseteq Q$, satisfying the property $\mathsf{P}(S, p)$. By restricting Q to the subsequence Q' using only symbols in $\Sigma(p)$, the condition $\mathsf{P}(Q', p)$ may be still violated due to unequal numbers of occurrences of various symbols, i.e., because $Q' \notin \bigotimes\langle p_1^m, ..., p_z^m\rangle$. It suffices now to "exclude" from such sequence Q' the irrelevant symbol occurrences, namely, those $Q'_k = p_{i+1}$ which violate the condition $noc(p_i, Q'[1:k]) \geq noc(p_{i+1}, Q'[1:k])$. The first occurrence of p_{i+1} violating this condition signals that one of its occurrences so far will not enter the disjoint occurrences of p. In the sequence R from Example 3.4, encountering the third b means that one of the three b's does not enter a DO of pattern ab, since there are only two matching a's before. This "exclusion" is achieved in the trie by the numerals in the sublists, specifying only the number of relevant occurrences contributing to the maximal number of DOs of each pattern.

The importance of Fact 3.3 consists in allowing to determine the numbers of DOs of various patterns without detailing their actual occurrences. It establishes the correctness of the following Algorithm 3.5/3.6 which, in a single traversal of the input sequence (line 2), constructs the trie T and determines the number of DOs for all patterns. Algorithm 3.5 calls Algorithm 3.6 which adds each symbol S_i to the trie, traversing recursively the trie and updating the nodes storing the symbol S_i.

Algorithm 3.5 BuildTrie(S:Seq)

Output: A trie T with pattern occurrences in S where $\forall_{p \in T} : nrOc[p] = noc(p, S)$, by Fact 3.3

1: $T = new\ Trie$
2: **for each** $1 \leq i \leq |S|$ **do**
3: AddToTrie$(S_i, i, T.root)$ ..... // Algorithm 3.6

Algorithm 3.6 below assumes the following attributes at every node P of the trie:

- $P.symb$ – the symbol stored at the node.
- $P.pairs$ – the list of $[N : L]$ pairs, with a numeral N and list of positions L, each component accessed by its name.
- $P.nrOcc$ – the sum of the N-components, namely, for $m = |P.pairs| : P.nrOcc = \sum_{i=1}^{m} P.pairs_i.N$ ($= m$ when all $N = 1$).
- $P.children$ – the set of children nodes.

The algorithm is complicated by the need to record the information about the occurrences of symbols and to manipulate various list elements. But its essential operation consists simply in extending every pattern p, which does not already contain the symbol a, to pa and in adding the current symbol occurrence $a = S_i$ to the list of every pattern pa.

Starting from the root of the trie (representing the empty pattern), for each node P the algorithm inspects all P's children C. While a is not a symbol in C, it descends recursively the tree, line 15. Otherwise, if the test $C.symb = a$ at line 2 succeeds, there are several cases:

- If the parent node P is the root of the trie (line 3), then its children mark the starts of patterns and, having no preceding symbols, their occurrences need not be divided into sublists. Therefore, the new occurrence i is simply added to the list of occurrences of C (line 5) and the number of occurrences is increased as well (line 4).

Algorithm 3.6 AddToTrie(a:Symbol, i:int, P:NodeOfTrie)

```
1: for each C ∈ P.children do
2:   if C.symb = a then
3:     if P is the root of the trie then
4:       C.pairs_1.N = C.pairs_1.N + 1
5:       C.pairs_1.L.append(i)
6:     else if P.nrOcc > C.nrOcc then
7:       if P.pairs_-1.L_-1 < C.pairs_-1.L_1 then
8:         C.pairs_-1.N := C.pairs_-1.N + 1
9:         C.pairs_-1.L.append(i)
10:      else
11:        C.pairs.append([1 : i])
12:    else
13:      C.pairs_-1.L.append(i)
14:  else
15:    AddToTrie(a,i,C)
16: if ∀_{C∈P.children} : C.symb ≠ a then
17:   P.children := P.children ∪ {new node (a, [1 : i], ∅)}
```

- Otherwise, if P is not the root node and the symbol at P has more occurrences than C (the test at line 6 succeeds), then the current symbol a can enter a new DO of the pattern terminating at C. The question now is whether this new occurrence should enter an existing sublist in C, or whether it should start a new sublist.
  - The two possibilities here depend on the test at line 7 which compares the last position in the last L of P ($P.pairs_{-1}.L_{-1}$) with the first position in the last L of C ($C.pairs_{-1}.L_1$). If the latter is greater, then this means that all occurrences in $C.pairs_{-1}.L$ come after the occurrences in $P.pairs_{-1}.L$ and therefore the current symbol occurrence i can enter the sublist $C.pairs_{-1}.L$. In this case, the N counter is increased (line 8) and the new position i is added to the corresponding L component (line 9).
  - Otherwise, if $P.pairs_{-1}.L_{-1}$ is greater than $C.pairs_{-1}.L_1$, then a new sublist must be created and this is done by appending $[1 : i]$ to the list of pairs in C (line 11).
- If P is not the root node and the symbol at P has no more occurrences than C (the test at line 6 fails), then the new occurrence of a is recorded, by appending i to the last pair of C (line 13) but keeping the number N of that pair unchanged (this represents an additional choice of $C.symb$ but not an additional occurrence of the overall pattern ending in C).
- If none of the children of P contains the current symbol a, then a new node is added to $P.children$ (line 17). This node contains the symbol a, the pair $[1 : i]$, and no children.

Complexity

The number of patterns occurring in S can be exponential in its length. If T is a sequence with all symbols distinct, then $S = TT$ has $2^{|S|/2}$ distinct patterns. Besides the fact that such a sequence has a solution with only 1 pattern T, it represents an exceptional case which can be handled by simple preprocessing. (E.g., count the number of distinct symbols and, if it approaches $|S|/2$, alert the user about the unusual instance.)

The average, expected instance of the problem consists of a relatively long sequence over a relatively small alphabet; i.e., for a given sequence S and alphabet Σ with $|\Sigma| = E$, we assume in general $E \ll |S|$. Now, the number of distinct patterns with non-repeating symbols is large as a function of E, $nP(E) = \sum_{i=1}^{i=E} \frac{E!}{i!}$. But it is independent from the length of the sequence, while the number of patterns actually occurring in the sequence S, $nP(S)$, can be expected to be much smaller than this worst case.

Under these assumptions, the algorithm is linear in the length of the sequence $|S|$, inspecting at most all (currently) occurring patterns at each sequence position, i.e.,

$$BuildTrie(S) = \mathcal{O}(|S| \cdot nP(S)). \tag{3.7}$$

4. The candidate solutions

The main Algorithm 2.7 iterates over the number k of patterns, starting with $k = 1$ and increasing k while no solution is found. In each iteration, it generates and verifies candidate solutions with k patterns. As the number of patterns in the trie can quickly reach thousands or even millions, trying all combinations of k patterns becomes unfeasible. This section describes how candidate solutions are generated and how their number can be limited.

The idea is that in any candidate solution M, the number of occurrences of each individual symbol must equal the number of occurrences of the same symbol in the whole sequence S, i.e. $\text{Symb}(M) = \text{Symb}(S)$. For example, the sequence $S = ababbbccbc$ can only admit candidate solutions with $\text{Symb}(M) = \text{Symb}(S) = \langle a^2, b^5, c^3 \rangle$; one such candidate is $M = \langle ab^2, bc^3 \rangle$. In general, a sequence S with an alphabet $\Sigma = \{a_1, a_2, ..., a_E\}$ can only admit a candidate solution $M = \langle p_1^{n_1}, ..., p_k^{n_k} \rangle$ (where $p_1, .., p_k$ represent different patterns) which satisfies the following system of equations, with one equation for each symbol in the alphabet,

$$\left\{ \begin{array}{ccccccc} \alpha_{1,1} \cdot n_1 & + & ... & + & \alpha_{1,k} \cdot n_k & = & noc(a_1, S) \\ \vdots & & & & \vdots & & \vdots \\ \alpha_{E,1} \cdot n_1 & + & ... & + & \alpha_{E,k} \cdot n_k & = & noc(a_E, S) \end{array} \right. \tag{4.1}$$

where $\alpha_{i,j} = 1$ if pattern p_j contains symbol a_i, and zero otherwise; n_j is the multiplicity of pattern p_j in M, and $noc(a_i, S)$ is the number of occurrences of symbol a_i in S. In this system, all $\alpha_{i,j}$ and n_j are unknowns, for $1 \leq i \leq E$ and $1 \leq j \leq k$, and only $noc(a_i, S)$ are known directly from (the trie of) S.

Example 4.2. *For the sequence $S = ababbbccbc$ with the alphabet $\Sigma = \{a, b, c\}$, the system of equations for $k = 2$ patterns is shown to the left, with a possible instantiation and solution shown to the right:*

$$\left\{ \begin{array}{ccccc} \alpha_{1,1} \cdot n_1 & + & \alpha_{1,2} \cdot n_2 & = & 2 \\ \alpha_{2,1} \cdot n_1 & + & \alpha_{2,2} \cdot n_2 & = & 5 \\ \alpha_{3,1} \cdot n_1 & + & \alpha_{3,2} \cdot n_2 & = & 3 \end{array} \right. \qquad \left\{ \begin{array}{ccccc} 1 \cdot 2 & + & 0 \cdot 3 & = & 2 \\ 1 \cdot 2 & + & 1 \cdot 3 & = & 5 \\ 0 \cdot 2 & + & 1 \cdot 3 & = & 3 \end{array} \right.$$

The solution means that p_1 contains the symbols $\{a, b\}$, p_2 contains $\{b, c\}$, $n_1 = 2$ and $n_2 = 3$. The candidates are therefore: $M_1 = \langle ab^2, bc^3 \rangle$, $M_2 = \langle ab^2, cb^3 \rangle$, $M_3 = \langle ba^2, bc^3 \rangle$ and $M_4 = \langle ba^2, cb^3 \rangle$.[6]

Determining the possible solutions of the equation system (4.1) is complicated by the fact that not only the vector $[n_1, ..., n_k]$ but also the coefficients $\alpha_{i,j}$ are unknown. These coefficients are boolean, admitting only 0 or 1, so there are 2^{kE} ways of setting them.[7] Furthermore, even after instantiating the coefficients $\alpha_{i,j}$, the system may take different forms depending on whether $k < E$, $k > E$ or $k = E$. Seen as a system of linear equations over integers, when $k < E$ it has, typically, no solutions (being overdetermined, with more equations than unknowns), and when $k > E$ it is underdetermined, admitting infinitely many solutions. The third case, $k = E$, is the only one that would make it amenable to usual linear algebra packages, based on inverting square matrices.

Enumerating all the 2^{kE} systems of equations and solving each of them for $[n_1, ..., n_k]$ is hardly desirable. In the currently implemented prototype, we produce systematically all possible solutions to the first equation,

[6] A second solution to the system, namely $\underline{p}_1 = \{b, c\}$, $\underline{p}_2 = \{a, b\}$, $n_1 = 3$ and $n_2 = 2$, generates the same candidates.

[7] Some combinations of $\alpha_{i,j}$ can be ruled out immediately. For example, there can be no zero rows (i.e. $\forall_i : \sum_j \alpha_{i,j} > 0$) as this would mean not covering a symbol, and each column must contain at least two 1's ($\forall_j : \sum_i \alpha_{i,j} >= 2$) since any pattern should have at least two symbols.

then try these solutions in the second equation, then in the third, and so on. Initially, equations are also sorted according to the increasing values of $noc(s_i, S)$. This value is proportional to the number of possible solutions to the equation, so we start with the least possible number. The set of possible solutions diminishes[8] as we go through the system, keeping only the solutions that satisfy all of the previous equations. One ends up with the complete set of solutions, which becomes empty in case of an inconsistent system.

A solution to the equation system (4.1) defines the set of symbols that each pattern p_j may contain, denoted $\underline{p}_j = \{s_i \mid \alpha_{i,j} = 1\}$. Every permutation of the symbols in $\underline{p}_j$ gives a possible pattern, and a pattern p generated from $\underline{p}_j$ will enter a candidate solution with multiplicity n_j. One must check whether this is actually possible, as there could be some permutations of $\underline{p}_j$ which do not have n_j occurrences in S. To check this, we resort to the trie, and accept the pattern p over symbols $\underline{p}_j$ into a candidate only if $noc(p, S) \geq n_j$.

A sketch of this procedure is given in Algorithm 4.3. Having solved the equation system (line 1), the algorithm generates the permutations for each $\underline{p}_j$ (lines 4–6) and, if a given permutation p has n_j occurrences in S (line 7) then p is included in the set of patterns Y_j. Having constructed k sets of patterns, $Y_1, ..., Y_k$, for each of the solutions of the equation system, the algorithm combines them into candidate solutions for S (line 9). The condition $\forall_{1 \leq i,j \leq k} : i \neq j \Rightarrow p_i \neq p_j$ ensures that all patterns in a candidate are different, even if they come from identical sets $\underline{p}_i = \underline{p}_j$. (If identical patterns were allowed then such candidate would have fewer than k different patterns; but if such solution exists, then it has already been found in an earlier iteration of Algorithm 2.7 for a smaller value of k.)

Algorithm 4.3 Candidates(S:Seq, k:int)

Input: Sequence S (its trie T) and the number of patterns to consider
Output: the set $Cand$ of multisets M of k patterns, with $\text{Symb}(M) = \text{Symb}(S)$
1: $W :=$ solutions to the equations (4.1), each in the form $\langle \underline{p}_1^{n_1}, ..., \underline{p}_k^{n_k} \rangle$
2: $Cand := \varnothing$
3: **for each** $\langle \underline{p}_1^{n_1}, ..., \underline{p}_k^{n_k} \rangle \in W$ **do**
4: **for each** $j := 1...k$ **do**
5: $Y_j = \varnothing$
6: **for each** permutation p of $\underline{p}_j$ **do**
7: **if** $noc(p, S) \geq n_j$ **then**
8: $Y_j := Y_j \cup \{p\}$
9: $Cand := Cand \cup \{\langle p_1^{n_1}, ..., p_k^{n_k} \rangle \mid \forall_{1 \leq j \leq k} : p_j \in Y_j \ \wedge \ \forall_{1 \leq i,j \leq k} : i \neq j \Rightarrow p_i \neq p_j\}$
10: **return** $Cand$

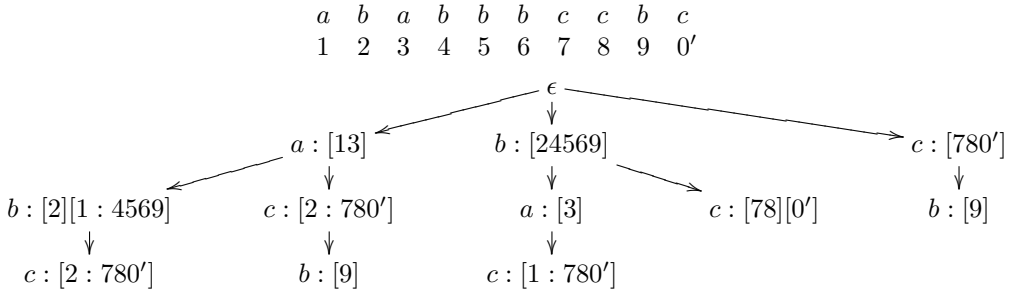
Example 4.4. *The system of equations for $S = ababbccbc$ and $k = 2$ patterns, admits the solution:*

$$\langle \underline{p}_1^2, \underline{p}_2^3 \rangle \text{ with } \underline{p}_1 = \{a, b\} \text{ and } \underline{p}_2 = \{b, c\}$$

The permutations of $\underline{p}_1$ are $\{ab, ba\}$ and the permutations of $\underline{p}_2$ are $\{bc, cb\}$. One would therefore consider the candidates: $M_1 = \langle ab^2, bc^3 \rangle$, $M_2 = \langle ab^2, cb^3 \rangle$, $M_3 = \langle ba^2, bc^3 \rangle$ and $M_4 = \langle ba^2, cb^3 \rangle$. However, looking at the trie of S, depicted below, one finds that the pattern ba has at most one occurrence, which rules out M_3 and M_4. Also, the pattern cb has at most one occurrence in S, while M_2 requires cb^3. The only valid candidate

[8] Except when some $\alpha_{i,j}$ becomes 1 when it was 0 in all previous equations; in this case, a new n_j enters the system and there is an increase in the number of possible solutions.

is thus M_1, which results from Algorithm 4.3 generating $Y_1 = \{ab\}$, $Y_2 = \{bc\}$, and $Cand = \{\langle ab^2, bc^3\rangle\}$.



Complexity

The length of each pattern, i.e., the size of each set $\underline{p}_i$, is limited by the size E of the alphabet, and the loop at line 6 of Algorithm 4.3 may iterate $E!$ times, giving the upper bound $k \cdot E!$ on the number of iterations of the loop at line 4. The complexity of this algorithm is dominated by the number $|W|$ of solutions to the system (4.1).

Being a linear system, and assuming it to have more than one solution, it will, in general, have infinitely many solutions. But we look only for ones with non-negative elements and limit these to the relevant subspace determined by the sequence and its length. Consider the number of solutions to a single equation for a single symbol, i.e., let $m = noc(a_i, S)$, to the equation $n_1 + ... + n_k = m$, where $m < |S|$. The upper bound on the number of its solutions is given by $\binom{m-1}{k-1}$, namely, the number of ordered additive k-partitions of m. The system has E such equations, but their solutions must be mutually compatible, i.e., yield the same n_j values. The number of solutions is thus limited by the equation having fewest of them. In short, overestimating $\binom{m-1}{k-1} = \mathcal{O}(m^{k-1})$, and taking $m = |S|$, we obtain the seriously overestimated bound $\mathcal{O}(|S|^{k-1})$ on the number of the solutions to the whole system (4.1). But even this overestimate shows the complexity polynomial in the length of the sequence.

We record also the potential number 2^{kE} of possible instances of the matrix α, each giving another system of equations to be solved. Solving such a small system of linear equations is of negligible complexity, so we ignore it in the expression for the whole algorithm which becomes thus:

$$Candidates(S, k) = \mathcal{O}(2^{kE} \cdot k \cdot E! \cdot |S|^{k-1}). \tag{4.5}$$

This may give exponentiation in $|S|$ when E or k approach it. But as we have observed before, these are the cases unlikely to occur in practice. Still, even if practically polynomial in $|S|$, the number of candidates may quickly become unmanageable, as k increases for longer sequences. Our experiments and the case study presented ahead in Section 7 confirm this worry, indicating the need to further limit the number of potential candidates. Still, this number is not the highest factor contributing to the overall, worst case complexity. It is the verification of particularly malicious ones, as we will see in the following section.

5. Verifying the candidates

Given a candidate solution – a multiset $M = \langle p_1^{n_1}, ..., p_z^{n_z}\rangle$ of patterns – we apply a variation of algorithm (2.6) to verify if the sequence can be covered by M. The operation $Ch(M, a)$ consumes the symbol a from the patterns in M, producing a set of children multisets, each corresponding to the removal of a from a different pattern in M. This is the opposite procedure to that of algorithm (2.6) which appended each symbol to the end of patterns in X; here we remove each symbol from the beginning of patterns in M. This operation is iterated starting with M, which represents the candidate to be tested, and applying it to the successive sets of multisets that are generated from M as we proceed through the sequence $S = S_1S_2...S_{|S|}$. The procedure is described in Algorithm 5.1 and illustrated in Example 5.3.

The operation $Ch(M, S_i)$ may produce the empty set if there is no pattern in M beginning with S_i (and no single-symbol pattern S_i in M). If this happens, verification fails and the candidate is not a

Algorithm 5.1 *Verify*(S:Seq, M:multiset)

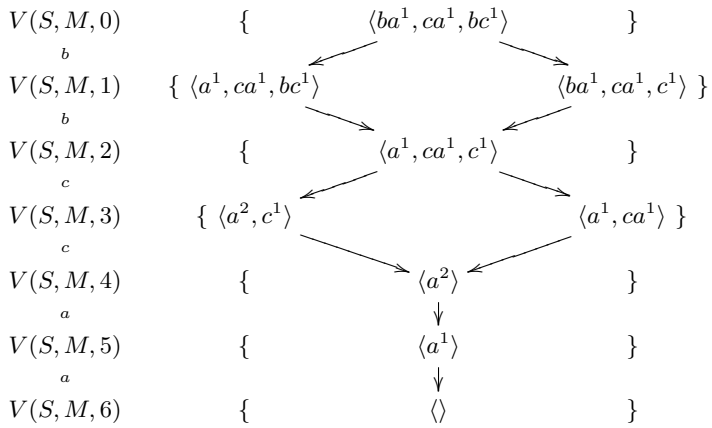
$$
\begin{array}{rcl}
V(S,M,0) & = & \{M\} \\
V(S,M,i) & = & \bigcup_{X \in V(S,M,i-1)} Ch(X,S_i), \text{ where} \\
Ch(X,a) & = & \{(X \ominus \langle ax^1 \rangle) \oplus \langle x^1 \rangle \mid ax \in X \wedge x \neq \epsilon\} \cup \{X \ominus \langle a^1 \rangle \mid a \in X\} \\
\mathit{Verify}(S,M) & = & V(S,M,|S|)
\end{array}
$$

solution. Verification is successful, i.e. $S \in \bigotimes M$, if we reach the end of S with the empty multiset, i.e., if $\langle\rangle \in V(S,M,|S|)$. This means that it was possible to consume the candidate M completely and exactly, matching its symbols against all successive symbols in S. The correctness of this claim is expressed by the following fact.

Fact 5.2. $S \in \bigotimes M \iff \langle\rangle \in V(S,M,|S|)$.

The proof does not need to assume that patterns in the candidate M have no repeating symbols. In fact, Algorithm 5.1, *Verify*, works for arbitrary patterns, and can be used for verification of candidates other than those generated by Algorithm 4.3.

Example 5.3. *For* $M = \langle ba^1, ca^1, bc^1 \rangle$ *and the sequence* $S = bbccaa$ *we obtain :*



On completing the sequence, we check if the resulting collection of multisets contains $\langle\rangle$*, which marks success. In the present case, it is the only element of the result. The same sequence tested for the candidate* $M' = \langle bca^2 \rangle$ *would lead to the steps:*

$$\{\langle bca^2 \rangle\} \xrightarrow{b} \{\langle bca^1, ca^1 \rangle\} \xrightarrow{b} \{\langle ca^2 \rangle\} \xrightarrow{c} \{\langle a^1, ca^1 \rangle\} \xrightarrow{c} \{\langle a^2 \rangle\} \xrightarrow{a} \{\langle a^1 \rangle\} \xrightarrow{a} \{\langle\rangle\}$$

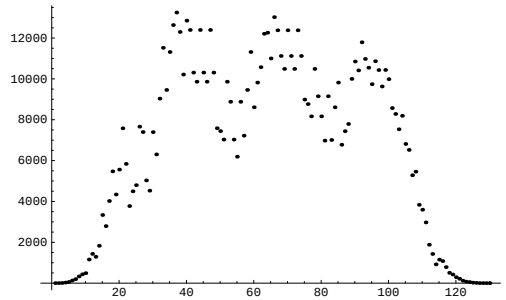
while for $M'' = \langle bac^2 \rangle$*:* $\{\langle bac^2 \rangle\} \xrightarrow{b} \{\langle bac^1, ac^1 \rangle\} \xrightarrow{b} \{\langle ac^2 \rangle\} \xrightarrow{c} \varnothing$

The paths from $V(S,M,0)$ to a solution node $\langle\rangle$ mark the DOs in S of the patterns from M. For S and M in the above example, the first b can be used either for ba (the left branch) or for bc (the right branch). There is no choice at the following b, which gives the same resulting multiset, but then the first c can be consumed from c^1 (the right branch) or from ca^1. The algorithm does not keep track of all DOs. For instance, at $V(S,M,2)$, the two possibilities from the previous step are merged into one, since it is inessential to know which b was used for ba and which for bc. This optimization helps reducing the complexity of the present algorithm, which is the most significant contribution to the overall complexity of the main Algorithm 2.7, as explained in the rest of this section.

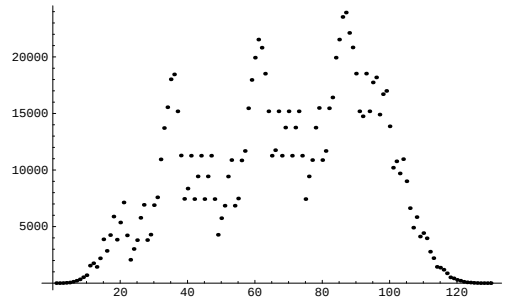
5.1. Complexity

Algorithm 5.1 traverses S only once but its worst case complexity can be dominated by the number of multisets at each stage. There are, namely, quite frequent situations where the same combination of patterns covers the sequence with different multiplicities.

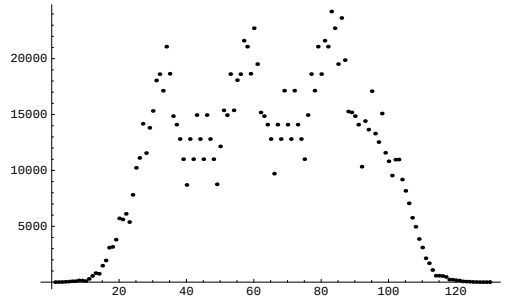
Example 5.4. *The values of $|V(S, M, i)|$ for the sequence $S = 5 \cdot T$ with $T = babcbccbccaaacbabcabcaabbc$ of length $|S| = 130$, tested for the specified candidates are given in the figure below. The candidates in a) and b) have the same set of patterns. The candidates in b) and c), with 6 patterns, reach some 25000 multisets. (Note that distinct graphs use different scales.)*



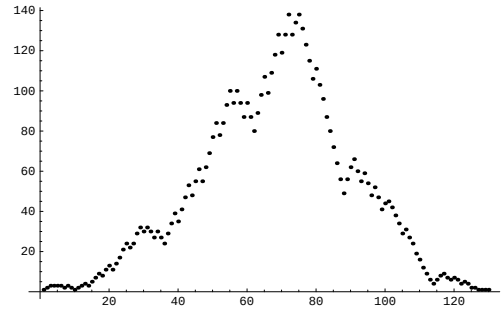
a) $\langle ac^2, ba^2, bc^5, cb^2, bca^3, cab^{33}\rangle$



b) $\langle ac^2, ba^2, bc^2, cb^5, bca^{32}, cab^4\rangle$



c) $\langle acb^5, ac^2, ab^2, bc^5, cb^2, bca^{31}\rangle$



d) $\langle ac^{20}, ab^{20}, bc^{25}\rangle$

The growth, reaching relatively high values in b) and c), is due to the fact that at each position i, symbol S_i can often be consumed from several elements in each of the current multisets. Given symbol S_i, each multiset $X \in V(S, M, i-1)$ will generate several offspring via $(X \ominus \langle ax^1\rangle) \oplus \langle x^1\rangle$. While there are sufficient multiplicities, both x and ax will coexist in the same multiset, and the number of multisets will keep increasing rapidly. It is only when some elements vanish (as a result of their multiplicity reaching zero) that the number of possible ways to consume the next symbols diminishes.

In fact, these two cases are still far from the worst one. Verifying the patterns from c) with more evenly distributed multiplicities, namely, $\langle acb^5, ac^{10}, ab^{10}, bc^{13}, cb^2, bca^{15}\rangle$ leads to the following development (which was stopped around position 40 of the sequence): 1, 2, 6, 11, 26, 46, 80, 87, 154, 139, 91, 289, 615, 1074, 964, 1887, 3165, 5070, 5058, 7964, 12032, 11407, 16913, 23181, 33544, 45115, 44924, 59422, 78273, 98725, 98447, 123072, 118746, 107956, 138858, 123927, 104481, 142050, 184820, 230777, 196858, 265663, 323495, 415502... In general, for the same combination of patterns, an even distribution of multiplicities gives more possible multisets than uneven one, as explained below.

These high numbers, however, represent an exceptional situation, occurring when the involved patterns contain (almost) all permutations (of a given length) of the alphabet. In such cases, the user may be warned about the potential problem. Observe though that one will then, typically, obtain a solution with fewer patterns, before trying such a large candidate. The sequence S from Example 5.4 has several solutions with only 3 patterns and case d) shows a verification of one of them. Its peak is around 150 times lower than in the cases b) and c).

The following analysis does not aim at any exact, worst case upper bound on the number of distinct multisets, but at showing the polynomial (in the length of the sequence) complexity of the algorithm and identifying the factors in the exponent. Writing now S for the length of the sequence should not cause any confusion, while simplifying notation.

At each sequence position, the number of multisets is limited by the actual candidate $M = \langle p_1^{n_1}, ..., p_k^{n_k} \rangle$. Each pattern p_i can give rise to $|p_i|$ distinct suffixes and, at each point (after the initial stage of approaching the global maximum), n_i is distributed between these (until it starts diminish due to the disappearance of the used patterns from the multiset). The worst case is thus limited by the number of ordered additive partitions of n_i into p_i components (writing now p_i for $|p_i|$), which equals $\binom{n_i-1}{p_i-1}$. All such distinct multisets for a given i can be, in principle, combined with all multisets for another $j \neq i$, giving the hardly possible upper bound of $\prod_{i=1}^{k} \binom{n_i-1}{p_i-1}$ on the maximal number of multisets in any $V(S, M, x)$. The maximum of this product is bounded from above by the case when all multiplied numbers are equal, i.e., $\binom{n-1}{p-1}^k$. Given a fixed k and the fact that $S = n \cdot p \cdot k$, we estimate $\binom{(S/pk)-1}{p-1}^k$. We calculate $\binom{(S/pk)-1}{p-1} = \frac{((S/pk)-1)!}{(p-1)!((S/pk)-1-(p-1))!} = \frac{((S/pk)-1)\cdot((S/pk)-2)\cdot\ldots\cdot((S/pk)-(p-1))}{(p-1)!} < \frac{(S/pk)^{p-1}}{(p-1)!} = \frac{S^{p-1}}{(pk)^{p-1}\cdot(p-1)!}$ and substitute into the exponentiation by k, obtaining

$$\mathit{Verify}(S, M) \quad \leq \quad \mathit{Verify}(S, \langle \underbrace{p^{S/pk}, \ldots, p^{S/pk}}_{k} \rangle) \quad \leq \quad \frac{S^{pk-k}}{((pk)^{p-1} \cdot (p-1)!)^k} \quad = \quad \mathcal{O}(S^{pk-k}). \tag{5.5}$$

Although the exponent is worryingly high, we rest satisfied with the complexity polynomial in the length of the sequence and with the fact that the obtained expression, ignoring the relatively high denominator, is severely overestimated. The representative cases from Example 5.4 show that the maximal number of multisets, reached only a few times during computation, is far below this overestimated value.

5.2. The overall complexity

To estimate the overall complexity of the main Algorithm 2.7, we note that the exponent in (5.5) depends on the number of patterns and on their length. Assuming, as we did before and as is the case in all practical applications, an alphabet of a fixed and limited size $E \ll |S|$, we obtain also the upper bound on the lengths of patterns, $p \leq E$, and on their maximal number in a possible solution, $k \leq nP(E) = \sum_{i=1}^{i=E} \frac{E!}{i!}$. (Again, the potentially high value of $nP(E)$ is not to be expected in actual sequences, with the number of patterns in an actual solution much lower than the number of patterns occurring in the sequence which, in turn, is much lower then the number of patterns in the whole alphabet, $K \ll nP(S) \ll nP(E)$.)

Substituting the limit E for p, we obtain the average multiplicity $n = S/kE$ of each of k patterns. The overall complexity, ignoring the initial building of the trie, is dominated by the maximum of $\mathit{Candidates}(S, k) \times \mathit{Verify}(S, \langle E^{S/kE} \rangle^k)$, as k increases $1 \leq k \leq K$, reaching the number K of patterns in an actual solution. Taking $\mathit{Candidates}(S, k) = \mathcal{O}(k \cdot E! \cdot 2^{kE} \cdot S^{k-1})$ from (4.5) and applying (5.5) for *Verify*, we obtain the upper bound for $k = K$:

$$\mathit{MinCover}(S) \quad \leq \quad \sum_{k=1}^{K} k \cdot E! \cdot 2^{kE} \cdot S^{k-1} \cdot S^{Ek-k} \quad = \quad \mathcal{O}(K \cdot E! \cdot (2S)^{KE}) \tag{5.6}$$

As emphasized repeatedly and as indicated by the examples, the exponent KE is unrealistically high and never occurs in practice. Also, since K is limited by a function of E, it shows the critical importance of the size E of the alphabet. As this size remains relatively small and constant, we can expect the developed algorithm to scale reasonably well on long sequences.[9]

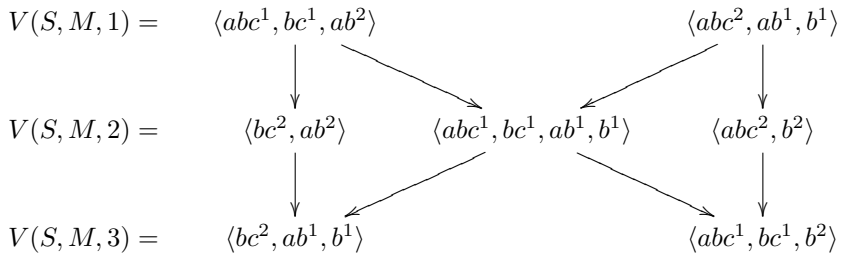
[9] In theory, S could enter the exponent, if the number of patterns, each of size (approaching) 2, approaches $\frac{S}{4}$. An example would be a sequence *aabbaaccaaddaaeeaaff...*, where half of the symbols are *a*'s and each must be combined with some other letter to form a pattern. This would mean having an alphabet of the order $E = \frac{S}{4}$, which is not realistic in pratical applications.

6. Improvements and adaptations

The following subsections explain further improvements in the implementation of the main algorithm, and the adjustments needed to solve the related Problem 2.2 as well as to address other types of behavior that may occur in practice.

6.1. Forward or backward testing

The following simple test enhances the performance by reducing the number of multisets generated during the verification of candidates. For example, the first steps in the verification of the candidate $M = \langle abc^2, ab^2\rangle$ for the sequence $S = aaaabbbbcc$ would generate:



However, we can run the same algorithm backward, with all patterns reversed, i.e., for $S' = ccbbbbaaaa$ and $M' = \langle cba^2, ba^2\rangle$, which starts $M' \xrightarrow{c} \langle cba^1, ba^3\rangle \xrightarrow{c} \langle ba^4\rangle$... and continues with only one multiset through the whole sequence. The difference is that while patterns in M have common prefixes, which lead to splits of the multisets into multiple children, the patterns in M' have no common prefixes and do not generate such splits. Before verifying the candidate, the algorithm checks whether to do it forward or backward, by estimating the number of overlapping prefixes and suffixes and choosing the alternative with the lowest number.

6.2. Checking the same set of patterns only once

As we have seen, there may be several candidates with the same set of patterns but with different multiplicities and, in general, it is faster to check candidates with uneven rather than balanced distribution of multiplicities. We generate the candidates with Algorithm 4.3 and we check them with Algorithm 5.1. The latter is called after we have collected all possible candidates, and after we sort those candidates so that the ones with uneven distribution of multiplicities will be checked first. Once a solution for a given pattern combination has been found, this combination is not further investigated, and one only checks other pattern combinations. For example, the sequence $S = abbabbaabbaa$ was built from $\bigotimes\langle ab^3, ba^3\rangle$. However, as soon as the candidate $\langle ab^4, ba^2\rangle$ is verified and is shown to be a solution, other candidates with the same combination of patterns (such as $\langle ab^3, ba^3\rangle$ and $\langle ab^2, ba^4\rangle$) will not be verified.

6.3. Specifying patterns, Problem 2.2, SP'

This version, where the user specifies also a subset of patterns to be used in the solution, and which is NP-hard, is solved by restricting appropriately the generated candidates. Computationally, it represents therefore a limitation of our main Algorithm 2.7. If $Pred(p)$ is the predicate required of each pattern p involved in the solution, we ensure that only such patterns are collected into the candidates generated by Algorithm 4.3, inserting there an additional test:

6: **for each** permutation p of $\underline{p}_i$ **do**
7: **if** $noc(p, S) \geq n_i$ **and** $Pred(p)$ **then**
8: ...

6.4. Handling repeating symbols

Having the set of all solutions to Problem 2.3, with all patterns containing no repeated symbols, we can with relative ease generate also solutions where patterns admit repeating symbols. For any solution $M = \langle p_1^{n_1}, ..., p_k^{n_k} \rangle$ we simply consider combinations of patterns p_i depending on their multiplicities. For instance, since $\forall_{r \in \bigotimes \langle p,q \rangle} : \bigotimes \langle r^n \rangle \subseteq \bigotimes \langle p^n, q^n \rangle$ so, if two patterns have the same multiplicity $n = mlt(p, M) = mlt(q, M)$, it will be natural to consider possible solutions which replace in M the pair $\langle ..., p^n, q^n, ... \rangle$ by $\langle ..., r^n, ... \rangle$ for all $r \in \bigotimes \langle p, q \rangle$. In general, when $mlt(p, M) = m < n = mlt(q, M)$, one may consider solutions with $\langle ..., r^m, q^{n-m} ... \rangle$ instead of $\langle ... p^n, q^m ... \rangle$, where $r \in \bigotimes \langle p, q \rangle$. This process can be iterated. Doing it in all possible ways would require a significant amount of computation, but one can expect that only a few combinations will make sense for the actual problem instance, so that the formation and the number of trials is better left to the user.

Considering such alternatives amounts here to running Algorithm 5.1 on the initial sequence with these modified candidates. This works fine because the algorithm does not assume that patterns have no repeating symbols and works equally well for all kinds of patterns.

6.5. Handling loops

Even if patterns have no repeating symbols, the generating process may have loops or other forms of repetitive behavior. This can be handled by our approach without adaptation, with the only provision that it will identify the body of the loop as a separate pattern. For example, if the process is specified as $a \to b \to c \to d$ with an additional looping arc from d to b, then the algorithm may find pattern $abcd$ as well as bcd with a multiplicity related to the number of times the loop was executed. The presence of both a pattern p and a substring of p in a solution is an indication of such behavior.

6.6. Handling parallelism

Another type of behavior that frequently occurs in process models is parallelism [31]. For example, in a process with two parallel branches ab and cd, their concurrent execution may lead to (sub-)sequences such as $abcd$, $acbd$, $cabd$, $acdb$, $cadb$, etc. However, activities have often time constraints or other ordering restrictions, meaning that only a small fraction of all possible interleavings is actually observed in practice. In any case, the interleaving of parallel branches fits well into the nature of our problem and can be handled without adaptation, provided that, in a process with parallel branches, these will be captured as separate patterns. As a consequence, the presence of parallel behavior may increase the number of patterns required to find a minimal solution.

6.7. Coping with noise

In practical applications it is often the case that the event log generated from a running process contains some amount of *noise* [32]. This may be due to some fault in the recording mechanism (e.g. events not recorded, recorded out of order, etc.) or some problem in the process itself (e.g. an unexpected error that generates an unforeseen event). In our problem we have an additional reason to worry about noise, which is the fact that the given sequence may represent only a fragment of the whole event data. For example, when retrieving the sequence of events between two dates, the truncation at both ends may leave some incomplete instances, which correspond to incomplete pattern occurrences.

It is possible to cope with this issue both within and outside the scope of our problem. Within our algorithm, we can (at least partly) address this problem by relaxing the definition of pattern to include also patterns that have only one symbol or only one occurrence in the sequence. For example, the sequence $S = abcabecabc$ has no solution due to the presence of symbol e. However, if we allow for single-symbol and single-occurrence patterns, the sequence has 15 solutions, one of which is $\langle abc^3, e^1 \rangle$ and others include $\langle bca^2, aebc^1 \rangle$, $\langle abc^2, bcae^1 \rangle$, etc. The problem of such relaxation is indeed the extra number of candidates and solutions that suddenly appear.

An alternative way to cope with noise is by means of preprocessing. In the previous example, just by checking the frequency of each event, one would immediately realize that event e has a single occurrence and therefore could be attributed to noise. Such filtering is commonly used to prepare the event log before

the application of process mining techniques [33]. This can take care of isolated events, but not of single occurrences of certain fragments of patterns. E.g. the sequence $S' = abcababcabc$, where e has been replaced by ab, admits no solution, and the frequency of events reveals nothing suspicious. It is only when we relax the definition of pattern that we find the 3 solutions: $\langle abc^3, ab^1 \rangle$, $\langle abc^3, ba^1 \rangle$ and $\langle ab^4, c^3 \rangle$. Of course, if the fragment ab occurrs more than once in S', then it can be captured as a regular pattern.

The decision of whether to dismiss something as noise should also be made carefully. For example, seeing $\langle abc^{50}, ab^2 \rangle$ come up as a solution, the user might be tempted to dismiss ab^2 as noise. However, as we will see in the following case study, where something similar occurs, this can actually be given a meaningful interpretation in terms of the generating process.

7. Case study

The IT Governance unit at INOV[10] is a relatively small but growing R&D group which provides services and solutions for IT management, with a special focus on the implementation of ITIL solutions[11]. Currently, the group has several ongoing projects in companies from different industry sectors. These projects develop mostly on the basis of *service requests* from customers. A service request may represent a problem fix or the development of new functionality in the solution provided to a customer. In order to manage service requests, the group has created a supporting system for internal use, which keeps track of each request from the moment it is created to its successful fulfillment.

Figure 2 presents the business process associated with the handling of service requests. The process begins when a new service request is created. As the user records the request in the system, it may be left in a number of states. A "draft" state means that the user will resume filling in the request details at a later time. A "needs quotation" state means that the development team will have to provide a quotation (i.e. the estimated cost) for the service request. A "needs approval" state requires the unit coordinator to explicitly approve the request. The "submitted" state is used when the request needs no quotation nor approval.

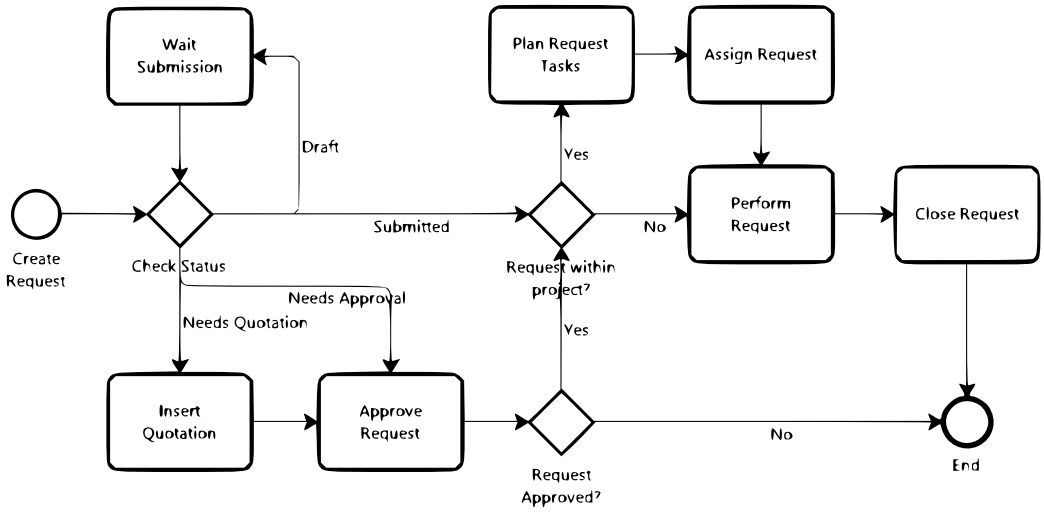


Figure 2: The process of handling service requests as implemented in the supporting system.

Service requests may be within or outside the scope of an existing project. If the request is within the scope of a project, then it enters that project as a set of additional tasks. These tasks will be assigned to a team member working on that project. If the request is outside the scope of a project, then it will be

[10]INOV is a technology transfer institute based in Lisbon (`http://www.inov.pt/`)
[11]ITIL = Information Technology Infrastructure Library (`http://www.itil-officialsite.com/`)

carried out as soon as someone is available to take care of it. After the request has been performed, another team member will check the results to ensure that the request was properly addressed. The request is then closed, and at this point the system records an end event.

This process is implemented in a Web-based software application where each activity consists in the user filling out a form and submitting it back to the system. The system then evaluates the request data in order to determine what the next activity will be. This means that all decision points are performed automatically by the system, while the remaining activities are performed manually by the user(s). The user also performs the initial activity which triggers the process, but it is the system that automatically records the end event. In addition to the end event, the system records all activities that take place, including the decision points which are performed automatically. There are a total of 12 different events, as illustrated in Table 2.

Event	Symbol	Event	Symbol	Event	Symbol
Create Request	a	*Approve Request*	–	*Assign Request*	e
Check Status	b	*Request Approved?*	–	*Perform Request*	f
Wait Submission	i	*Request within project?*	c	*Close Request*	g
Insert Quotation	–	*Plan Request Tasks*	d	*End*	h

Table 2: Symbols used to represent the events in the process of handling service requests.

7.1. Experiment 1

For the purpose of the case study, we had access to the events recorded by the system during a period of seven weeks between March 22, 2010 and May 11, 2010. In this experiment, the input dataset contained the events for service requests initiated and completed between those dates, i.e. only complete process instances have been considered. A total of 364 events have been extracted from the system, and each different event was relabeled with a single letter according to the order in which it appeared in the event log. For example, the first 5 events extracted from the system were:

Create Request $\longrightarrow$ *Check Status* $\longrightarrow$ *Request within project?* $\longrightarrow$ *Create Request* $\longrightarrow$ *Check Status* $\longrightarrow$ ...

and hence *Create Request* was relabeled to a, *Check Status* to b, *Request within project?* to c, etc.

Two things became immediately apparent from merely looking at the sequence of events. One is the overlap between process instances, as the events above suggest (a new instance is being initiated at the fourth event, while the first has not reached the end event yet). Second is that not all of the events defined in the process actually appear in the event log. As seen in Table 2, the three activities *Insert Quotation*, *Approve Request* and *Request Approved?* have not been relabeled with a symbol since they did not occur at all in the event log.

The 364-symbol sequence obtained was the following:

$$abcabcabcabcabcdedeffdeabcabcfghdeabcfdedefghfdeghghdeffghghabcdefghghghabc\\defghabcabcdeabcfdefdefdefghghghabcdefghabcdeabibcabibcfabcdefabcdeabcghfghdefd\\efghghabcabcabcabcdefdedeffdefghdefghghghabcabcghabcabcdefghabcdefdedefabcdefa\\bcdedghgheffffghghabcdeghabcghghabcdedefghfabcdeabcabcfghfdefdefghghghabcdefgh\\abcabcghabcdedefghdefghabcdefghfghghabcdefghabcdefgh$$

Running Algorithm 2.7 on this sequence produced the following minimal solutions,

$$\begin{array}{l}\langle abcdefgh^{45}, ib^2\rangle\\ \langle abcdefgh^{45}, bi^2\rangle\end{array} \tag{7.1}$$

which can be interpreted as follows:

- Both solutions represent the fact that there are only two variants of behavior in this event log.[12] One of these, common to both solutions, is the sequence $abcdefgh$ which corresponds, in Figure 2, to the

[12]We refer to variants in the sense of usage scenarios, as in [34]. Note that the usual meaning of process variants involves a reference model and a set of changes to that reference model, as in e.g. [35].

flow: *Create Request* $\longrightarrow$ *Check Status* $\longrightarrow$ *Request within project?* $\longrightarrow$ *Plan Request Tasks* $\longrightarrow$ *Assign Request* $\longrightarrow$ *Perform Request* $\longrightarrow$ *Close Request* $\longrightarrow$ *End*.

- The second variant is either *ib* or *bi*. Clearly, none of these can represent a complete pattern since any complete process instance would start with an a and end with h. The pattern that these fragments belong to is $abibcdefgh$, with *ib* or *bi* standing out as $abibcdefgh$ or $abibcdefgh$, respectively. But this pattern repeats symbol b and is therefore not captured directly. The correct way to interpret these solutions would be $\langle abcdefgh^{43}, abibcdefgh^2\rangle$. This is confirmed to be a solution by Algorithm 5.1, which runs successfully on this candidate and without ever exceeding 2 multisets.

- From the above, one concludes that there are 43 instances of $abcdefgh$ and 2 instances of $abibcdefgh$. The same conclusion can be reached without prior knowledge about the process by noting that any sequence in the form $xyzyzy...zy$, with n repetitions of zy, will be captured both as $\langle xy^1, zy^n\rangle$ and $\langle xy^1, yz^n\rangle$ depending on whether xy picks up the first or the last y in the sequence. Recognizing the same situation with *ib* and *bi* in the solutions above, one could admit the hypothesis of having either $\langle abcdefgh^{43}, abibcdefgh^2\rangle$ or $\langle abcdefgh^{44}, abibibcdefgh^1\rangle$ as solutions of the sequence. However, *Verify* does not confirm the latter, while it does the former.

- In terms of the business process, this means that there were 45 service requests fulfilled during the seven weeks captured in the event log. All process instances followed basically the same path in the process, with only two of them having the intermediate activity *Wait Submission*. In particular, this means that in the period of this study there were no service requests with a need for quotation or approval. Moreover, all service requests were found to be within the scope of some existing project, since there were no direct transitions from *Request within project?* to *Perform Request*.

We used an implementation of Algorithm 2.7 in Python which, running on a 2.2GHz machine, recovered the original patterns in about 3 minutes.[13] Most of this time was spent building the trie (Algorithms 3.5 and 3.6), which ended up having approximately 1.000.000 patterns. There was only one solution to the equation system (4.1): $\langle\{abcdefgh\}^{45}, \{bi\}^2\rangle$. From this solution, and using the information stored in the trie, Algorithm 4.3 produced only the two candidates (7.1). This was because the set $\{abcdefgh\}$, despite having 8 symbols, had only one permutation with 45 occurrences in S. The candidates were verified by Algorithm 5.1 and both turned out to be a solution.[14]

7.2. Experiment 2

For the second experiment, we obtained a new dataset including not only the events for the completed process instances but also the events for the incomplete ones. The same relabeling from Table 2 was used and again the three activities *Insert Quotation*, *Approve Request* and *Request Approved?* have not been observed. The following 543-symbol sequence S was obtained:

$$
\begin{aligned}
&abababababcababababababababababababcababcababababcabcabcabcabcabcababababcdeded\\
&efffabcdeabcabcfghdeabcfdedefghfdeghghdeffghghabcdefghghghabcdefghabibibcabcabc\\
&deabcfdefdefdefghghghabcdefghabcabcdeabibcabibcfabcabcdefabcdabeibcabcghfghdefde\\
&fghghabcabcabcabcdefdedeffdefghdefghghghabcabcghabcabcabcabcdefghabcdefdedef\\
&abcdefabcdedghghefdeffghghabcdeghabcghghabcabcdedefghfabcabcdeabcabcfghfdefdef\\
&ghghghabcdefghabcabcabcghabcdedefghdefdeghabcdefghfghabcabcghabcdeabcfabcabcabc\\
&abcdefdefabcdeabcdeffdefdefdefdeabcdeababcdefghabcdefabcabcdeabcdefgh
\end{aligned}
$$

Running Algorithm 2.7 on this sequence showed that there was no solution of the equation system (4.1) with a number of patterns $k = 2$, $k = 3$ or $k = 4$. Proceeding to $k = 5$ patterns, Algorithm 4.3 found

[13] The implementation is available at: `http://web.tagus.ist.utl.pt/~diogo.ferreira/seqpart/`

[14] For comparison, the approach described in [22] (which, as explained in Section 1.2, does not aim at listing all minimal solutions but just finding one solution) produced the single solution $\langle abcdefgh^{43}, abi^2, bcdefgh^2\rangle$ in about 1s. This solution is verified by Algorithm 5.1, but it is not minimal.

423 solutions of the equation system (4.1) and a sample is shown in the first column of Table 3. In the second column, the number of permutations for each of these solutions of the equation system is given, with the factors corresponding, in Algorithm 4.3, to $|Y_j|$ for $1 \leq j \leq 5$, where Y_j contains only the patterns (permutations) with a sufficient number of DOs in S.

Equation system solutions	Permutations		
$\langle\{abcfgh\}^{40}, \{abcde\}^{34}, \{abde\}^{17}, \{abdef\}^{8}, \{bfghi\}^{5}\rangle$	$178 \times 120 \times 24 \times 120 \times 120$	=	7382016000
$\langle\{abcfgh\}^{40}, \{abde\}^{30}, \{abcde\}^{21}, \{abcdef\}^{8}, \{bcfghi\}^{5}\rangle$	$178 \times 24 \times 120 \times 720 \times 706$	=	260585164800
$\langle\{abcfgh\}^{40}, \{abde\}^{38}, \{abcde\}^{21}, \{cf\}^{8}, \{bcfghi\}^{5}\rangle$	$178 \times 24 \times 120 \times 2 \times 706$	=	723847680
$\langle\{abcfgh\}^{40}, \{abcde\}^{29}, \{abde\}^{22}, \{abdef\}^{8}, \{bcfghi\}^{5}\rangle$	$178 \times 120 \times 24 \times 120 \times 706$	=	43430860800
$\langle\{abcfgh\}^{40}, \{abcde\}^{26}, \{abde\}^{25}, \{abcdef\}^{8}, \{bfghi\}^{5}\rangle$	$178 \times 120 \times 24 \times 720 \times 120$	=	44292096000
$\langle\{abcfgh\}^{40}, \{abde\}^{33}, \{abcde\}^{26}, \{cf\}^{8}, \{bfghi\}^{5}\rangle$	$178 \times 24 \times 120 \times 2 \times 120$	=	123033600
$\langle\{abcde\}^{56}, \{abfgh\}^{40}, \{cf\}^{13}, \{bcghi\}^{5}, \{abde\}^{3}\rangle$	$17 \times 39 \times 2 \times 120 \times 24$	=	3818880
$\langle\{abcfgh\}^{40}, \{abcde\}^{34}, \{abdef\}^{13}, \{abde\}^{12}, \{bghi\}^{5}\rangle$	$178 \times 120 \times 120 \times 24 \times 24$	=	1476403200
$\langle\{abcfgh\}^{40}, \{abde\}^{30}, \{abcde\}^{16}, \{abcdef\}^{13}, \{bcghi\}^{5}\rangle$	$178 \times 24 \times 120 \times 720 \times 120$	=	44292096000
$\langle\{abde\}^{43}, \{abcfgh\}^{40}, \{abcde\}^{16}, \{cf\}^{13}, \{bcghi\}^{5}\rangle$	$24 \times 178 \times 120 \times 2 \times 120$	=	123033600

Table 3: The first 10 of 423 solutions to the equation system (4.1) for the 543-symbol sequence, with $k = 5$ patterns.

Clearly, the number of permutations becomes prohibitive, as well as the number of candidates generated from these permutations.[15] Verifying such a number of candidates becomes impractical, even if the verification of each candidate is relatively fast. Also, simply storing those candidates in memory becomes a problem. Therefore, it is impossible to run Algorithm 2.7 (using our equipment) on the sequence above with $k = 5$.

Still, Algorithm 5.1 allows us to verify candidate solutions that could be possibly obtained by other means. In the present case, we looked into the system database for event attributes that could help us identify whether two events belong to the same service request. From this investigation we obtained the following candidate with $k = 8$,

$$\langle ab^{25}, abc^{13}, abcde^{6}, abcdef^{8}, abcdefgh^{43}, abibcdefgh^{2}, abibc^{1}, abibibc^{1}\rangle \tag{7.2}$$

which verifies to be a solution by Algorithm 5.1. Among these 8 patterns, one can recognize some fringes ($abibc^1$ and $abibibc^1$) and potential loops (bi or ib). By removing the bi pattern from $abibc$ and $abibibc$, these turn into instances of abc. Removing also bi from $abibcdefgh$, this yields additional instances of $abcdefgh$. We might therefore consider the following candidate with $k = 6$,

$$\langle ab^{25}, abc^{15}, abcde^{6}, abcdef^{8}, abcdefgh^{45}, bi^{5}\rangle \tag{7.3}$$

and a similar one with ib^5 instead of bi^5. Both these candidates verify to be a solution by Algorithm 5.1. These solutions are fully consistent with those in (7.1) and support the earlier interpretation of the process.[16]

The results obtained in these two experiments confirm the conclusion (5.6) of the complexity analysis, suggesting that Algorithm 2.7 will perform satisfactorily even on long sequences, provided that they can be covered exactly by a small number of patterns. As this number increases, the number of candidates may become despairingly high, as in Table 3. Finding ways to exclude as many candidates as possible before starting verification is the main challenge for future work.

8. Conclusion

In this work we have addressed the difficult problem of interpreting a sequence of symbols as the interleaving of a multiset of patterns. In particular, the goal is to find the minimal set of patterns that can

[15] The number of candidates generated from each solution to the equation system will be less than the number of permutations shown in Table 3 once the condition $\forall_{1 \leq i,j \leq k} : i \neq j \Rightarrow p_i \neq p_j$ is imposed, but still it will be close to that number.

[16] For comparison, the approach described in [22] produces the solution $\langle a^{99}, bcdefgh^{45}, b^{25}, bc^{15}, bcdef^{8}, bcde^{6}, bi^{5}\rangle$. Again, this solution is not minimal, and contains two single-symbol elements a and b.

cover the sequence. In a process mining scenario where the case ids do not appear in the event log, finding the minimal set of patterns will provide a plausible explanation for how that event log has been generated from the execution of several process instances. Similar problems have been addressed in sequential pattern mining but not without some sort of constraints. Here we addressed the problem in its full generality, the only restriction being that each pattern must not contain repeating symbols, otherwise the sheer number of possible patterns would preclude any kind of complete search.

The proposed solution has several stages:

- First, we build a trie to determine the complete set of patterns occurring in the input sequence – and also their number of occurrences by listing the choices for each symbol but not the choices for the overall pattern.
- Second, we generate a list of candidate solutions by solving a system of equations so that each candidate preserves the number of occurrences of each symbol in the input sequence. The information stored in the trie is used here to retrieve those patterns which may enter a candidate solution.
- Third, each candidate solution is verified in order to check that it actually covers, and covers exactly, the given sequence.

The result is Algorithm 2.7, which is highly optimized and can handle sequences of hundreds of symbols with relatively small alphabets ($|\Sigma| < 10$) and a small number of patterns ($k < 5$). The algorithm is able to retrieve all minimal solutions from the entire search space. This represents a significant progress for a problem that seemed at first completely beyond modern computational capabilities. Although the problem is expected to be intractable, our solution provides hope for practical applications, especially, with further improvements in the efficiency of the implementation.

A prototype has been implemented and applied in a case study with real data coming from a business process implemented in a software application. Two experiments were conducted. In the first experiment, there was an event log without case ids containing over 350 events, but only with complete process instances. Given this sequence, the algorithm discovered the minimal solutions in a matter of minutes. In a second experiment we tried to handle all events, about 550, including those from the incomplete process instances as well. In this case, the sequence had no solutions with 2, 3, or 4 patterns, and this fact was found very quickly. However, when looking for solutions with 5 patterns, the number of candidates became exceedingly high. Finding ways to exclude as many candidates as possible is the main challenge for future work.

Overall, we have developed a set of techniques which can be of independent interest and which, in the present application context, allow to analyze non-trivial, real-life data. They offer a promising start and confidence that, in the future, it will be possible to recover the case ids from large unlabeled event logs in a reasonable amount of time. This will still present some challenges to the user, as there may be many solutions to choose from. But already the present implementation is a useful tool for analyzing the behavior of business processes, especially those that are not managed by process-aware information systems.

9. Appendix (proofs)

NP-hardness of Problem 2.2, SP', in its variant asking only for the minimal number of patterns needed for a solution, is shown by reducing to it the following NP-complete edge-coloring problem [26]. (This means that its more complex variants, in particular, one asking also for the list of patterns and their multiplicites, are also NP-hard.)

An edge-coloring of a graph $G = (V, E)$ assigns colors to edges so that adjacent edges (sharing a node) obtain different colors. The chromatic index of a graph, $\chi'(G)$, is the smallest number of colors needed for such a coloring. Letting $\Delta(G)$ denote the maximum degree in G, for any graph $G : \Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$, [36]. If $\chi'(G) = \Delta(G)$ the graph is Class 1, and otherwise it is Class 2. Almost all graphs are Class 1, but deciding if a given graph is Class 1 or 2 is NP-complete, as shown in [37].

Fact 9.1. *Problem 2.2, asking only for a minimal number of patterns solving a sequence, is NP-hard.*

PROOF: We reduce the problem of determining the chromatic index of a graph to this variant of SP'. Given a graph $G = (V, E)$ with $E \subseteq V \times V$, we order V totally and let

– $F = \{ab \mid (a, b) \in E, a < b\}$ be a set of building blocks representing the edges

– $P = \{p \in F^+ \mid p$ has no repeated symbols, its F substrings are ordered lexicographically$\}$.

– S be the concatenation of all elements of F ordered lexicographically, and $S(G) = SS$

A pattern p is thus a sequence of one or more edges. When it occurs in a solution, this means that its edges obtain the same color. The sequence $S(G) = SS$ is used to ensure that each pattern has at least 2 occurrences, one in each copy of S. The number of patterns in a solution to SP' for $S(G)$ and P gives $\chi'(G)$, where each pattern corresponds to a different color. □

Fact 3.3. *Let $p = p_1...p_z$ be a pattern with all symbols distinct, and let S be a shuffle of m copies of these symbols, i.e. $S \in \bigotimes\langle p_1^m, ...p_z^m\rangle$. Then $S \in \bigotimes\langle p^m\rangle$ iff $\mathsf{P}(S, p)$, where the latter is defined as $\mathsf{P}(S, p) \Leftrightarrow \forall_{1\le k\le|S|}\ \forall_{1\le i<z} : noc(p_i, S[1 : k]) \ge noc(p_{i+1}, S[1 : k])$.*

PROOF:⇐) Certainly $noc(p, S) \le m$ so, assuming $\mathsf{P}(S, p)$, we show that there are indeed m DOs of p in S, i.e., $S \in \bigotimes\langle p^m\rangle$. Order the m occurrences of p_z, $occ(p_z, S) = \{S_{a_1}, ..., S_{a_m}\}$, so that $a_1 < a_2 < ... < a_m$. $\mathsf{P}(S, p)$ implies that, (*) for each $1 \le i \le m : noc(p_{z-1}, S[1 : a_i]) \ge noc(p_z, S[1 : a_i]) = i$. Ordering analogously the m occurrences of p_{z-1}, $occ(p_{z-1}, S) = \{S_{b_1}, ..., S_{b_m}\}$, so that $b_1 < b_2 < ... < b_m$, (*) implies for every $1 \le i \le m : b_i < a_i$. Hence, each S_{a_i} can be preceded by respective S_{b_i}, forming m DOs of $p_{z-1}p_z$. Repeating now this argument with $b_1...b_m$ instead of $a_1...a_m$ and p_{z-1} instead of p_z, entails m DOs of $p_{z-2}p_{z-1}p_z$. Proceeding down to p_1, we obtain m DOs of p. They are mutually distinct because occurrences of each p_i are mutually distinct and, for any $i \ne j$, p_i and p_j are distinct symbols, so no S-occurrence of any symbol can appear at two such distinct p-indexes.

⇒) If $S \in \bigotimes\langle p^m\rangle$ then suppose $1 \le k \le |S|$ to be the first index in S where $\mathsf{P}(S, p)$ is violated, i.e., where $noc(p_i, S[1 : k]) + 1 = noc(p_{i+1}, S[1 : k]) = x + 1$, for some $1 \le i < z$. Then $S_k = p_{i+1}$ and there are x occurrences of p_i and p_{i+1} to the left of S_k. But this means that one of these $x+1$ occurrences of p_{i+1} does not belong to any disjoint occurrence of p, since x preceding occurrences of p_i allow to form at most x such disjoint occurrences using the currently considered occurrences of p_{i+1}. Since there are only $m - (x + 1)$ occurrences of p_{i+1} in the rest of the sequence $S[k + 1 : |S|]$, so there are at most $m - 1$ disjoint occurrences of p in S, contrary to the assumption that there are m such. Thus $\mathsf{P}(S, p)$ must hold. □

The following fact expresses correctness of Algorithm 5.1. For a candidate multiset M and a multiset M' obtained in some of $V(S, M, i)$, let $M \setminus\setminus M'$ denote the set of possible multisets which could have been consumed on the way from M to M'. E.g., starting with $M = \langle ba^1, bc^1, ca^2\rangle$ and arriving at $M' = \langle a^1, c^1\rangle$, we have $M \setminus\setminus M' = \{\langle b^2, ca^2\rangle, \langle ba^1, b^1, ca^1, c^1\rangle\}$. The first point is auxiliary for establishing the second one, as stated in the text.

Fact 5.2. *The following statements hold:*

1. $\forall_{0\le i\le|S|}$:
   (a) $\forall_{M'\in V(S,M,i), D\in M \setminus\setminus M'} : S[1 : i] \in \bigotimes D$
   (b) $S \in \bigotimes M \Rightarrow \exists_{M'\in V(S,M,i)} : S[i + 1 : |S|] \in \bigotimes M'$
2. $S \in \bigotimes M \Leftrightarrow \langle\rangle \in V(S, M, |S|)$

PROOF: 1a holds trivially when $V(S, M, i) = \varnothing$, so we assume that it is not empty. For $i = 0$ there is only one $\{M\} = V(S, M, 0)$ and $M \setminus\setminus M = \varnothing$, so the claim is vacuously true. Proceeding by induction on i, let $M' \in V(S, M, i)$ be arbitrary, i.e. $M' = (X \ominus \langle ax^1\rangle) \oplus \langle x^1\rangle$ or $M' = X \ominus \langle a\rangle$, for some $X \in V(S, M, i - 1)$. Any $D \in M \setminus\setminus M'$ is then equal to $Y \ominus y^1 \oplus ya^1$, or to $Y \oplus ya^1$, for some $Y \in M \setminus\setminus X$. Hence, since 1a holds for Y by IH and $a = S_i$, so it holds also for D.

1b holds trivially for $i = 0$ and we prove the claim by induction for $i \ge 0$. By IH, let $X \in V(S, M, i)$ be such that $S[i + 1 : |S|] \in \bigotimes X$. Then any shuffle of X covering $S[i + 1 : |S|]$ begins with $a = S_{i+1}$ and continues

with the rest of X from which a has been removed from the appropriate pattern. But such an $X \ominus ax^1 \oplus x^1$, or $X \ominus a$, is then among $Ch(X, a)$ and witnesses to 1b at $i + 1$.

2. If $S \in \bigotimes M$ then, on completion, $S[|S| + 1 : |S|] = \epsilon$ and, by 1b, $\epsilon \in M'$ for some $M' \in V(S, M, |S|)$. But since all patterns in all multisets are non-empty, this requires $M' = \langle\rangle$, since only then $\epsilon \in \bigotimes\langle\rangle$. Conversely, if $\langle\rangle \in V(S, M, |S|)$, it must appear in the last step, as the result of removing the last member of a multiset, i.e., as the child of $M' = \langle a^1\rangle$, for $a = S_{|S|}$, obtained by $\langle\rangle = M' \ominus \langle a^1\rangle$ in $Ch(M', a)$. By 1a, any $D \in M \setminus\setminus M'$ covers $S[1 : |S| - 1]$ which therefore implies that $S \in \bigotimes M$. □

Acknowledgment

We would like to thank Carlos Mendes at the IT Governance unit of INOV for providing us the data for the case study as well as insights regarding the process of handling service requests and the corresponding supporting system.

References

[1] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, A. J. M. M. Weijters, Workflow mining: A survey of issues and approaches, Data & Knowledge Engineering 47 (2) (2003) 237–267.

[2] A. Tiwari, C. Turner, B. Majeed, A review of business process mining: state-of-the-art and future trends, Business Process Management Journal 14 (1) (2008) 5–22.

[3] W. van der Aalst, T. Weijters, L. Maruster, Workflow mining: Discovering process models from event logs, IEEE Transactions on Knowledge and Data Engineering 16 (2004) 1128–1142.

[4] A. Rozinat, A. A. de Medeiros, C. Günther, A. Weijters, W. van der Aalst, Towards an evaluation framework for process mining algorithms, BETA Working Paper Series WP 224, Eindhoven University of Technology (2007).

[5] P. Laird, Identifying and using patterns in sequential data, in: ALT '93: Proceedings of the 4th International Workshop on Algorithmic Learning Theory, Springer-Verlag, 1993, pp. 1–18.

[6] R. A. Morris, W. D. Shoaff, L. Khatib, An algebraic formulation of temporal knowledge for reasoning about recurring events, in: Proceedings of the International Workshop on Temporal Representation and Reasoning (TIME), 1994, pp. 29–34.

[7] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the Eleventh International Conference on Data Engineering (ICDE), 1995, pp. 3–14.

[8] R. Srikant, R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, in: Proceedings of the 5th International Conference on Extending Database Technology (EDBT), Vol. 1057 of Lecture Notes in Computer Science, Springer, 1996, pp. 3–17.

[9] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, M. Hsu, Prefixspan: Mining sequential patterns by prefix-projected growth, in: Proceedings of the 17th International Conference on Data Engineering (ICDE), 2001, pp. 215–224.

[10] C. Antunes, A. L. Oliveira, Sequential pattern mining algorithms: Trade-offs between speed and memory, in: Second International Workshop on Mining Graphs, Trees and Sequences (MGTS 2004 @ ECML/PKDD), 2004.

[11] T.-R. Li, Y. Xu, D. Ruan, W. ming Pan, Sequential pattern mining *, in: Intelligent Data Mining: Techniques and Applications, Vol. 5 of Studies in Computational Intelligence, Springer, 2005, pp. 103–122.

[12] B. Ding, D. Lo, J. Han, S.-C. Khoo, Efficient mining of closed repetitive gapped subsequences from a sequence database, in: Proceedings of the 25th International Conference on Data Engineering (ICDE), IEEE, 2009, pp. 1024–1035.

[13] B. Olstad, F. Manne, Efficient partitioning of sequences, IEEE Transactions on Computers 44 (11) (1995) 1322–1326.

[14] J. Pei, J. Han, W. Wang, Constraint-based sequential pattern mining: the pattern-growth methods, Journal of Intelligent Information Systems 28 (2) (2007) 133–160.

[15] H. Mannila, H. Toivonen, A. I. Verkamo, Discovery of frequent episodes in event sequences, Data Mining and Knowledge Discovery 1 (3) (1997) 259–289.

[16] W.-C. Peng, Z.-X. Liao, Mining sequential patterns across multiple sequence databases, Data & Knowledge Engineering 68 (10) (2009) 1014–1033.

[17] M. N. Garofalakis, R. Rastogi, K. Shim, Spirit: Sequential pattern mining with regular expression constraints, in: Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann Publishers Inc., 1999, pp. 223–234.

[18] D. R. Ferreira, M. Zacarias, M. Malheiros, P. Ferreira, Approaching process mining with sequence clustering: Experiments and findings, in: Proceedings of the 5th International Conference on Business Process Management (BPM 2007), Vol. 4714 of Lecture Notes in Computer Science, Springer, 2007, pp. 360–374.

[19] C. Wang, A. Kao, J. Choi, R. Tjoelker, Advances of Computational Intelligence in Industrial Systems, Springer, 2008, Ch. Discovering Time-Constrained Patterns from Long Sequences, pp. 99–116.

[20] M. Lothaire, Combinatorics on Words, Cambridge University Press, 1983.
[21] M. Ito, Shuffle decomposition of regular languages, Journal of Universal Computer Science 8 (2) (2002) 257–259.
[22] D. R. Ferreira, D. Gillblad, Discovering process models from unlabelled event logs, in: Proceedings of the 7th International Conference on Business Process Management (BPM 2009), Vol. 5701 of Lecture Notes in Computer Science, Springer, 2009, pp. 143–158.
[23] J. Rissanen, Modeling by shortest data description, Automatica 14 (5) (1978) 465–471.
[24] T. Calders, C. W. Günther, M. Pechenizkiy, A. Rozinat, Using minimum description length for process mining, in: SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing, ACM, New York, NY, USA, 2009, pp. 1451–1455.
[25] M. Walicki, D. R. Ferreira, Mining sequences for patterns with non-repeating symbols, in: IEEE Congress on Evolutionary Computation 2010 (IEEE World Congress on Computational Intelligence), Barcelona, Spain, 2010, pp. 3269–3276.
[26] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman, 1979.
[27] A. Durand, M. Hermann, On the counting complexity of propositional circumscription, Information Processing Letters 106 (4) (2008) 164–170.
[28] D. S. Johnson, M. Yannakakis, C. H. Papadimitriou, On generating all maximal independent sets, Information Processing Letters 27 (3) (1988) 119–123.
[29] M. W. Krentel, The complexity of optimization problems, in: STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing, ACM, New York, NY, USA, 1986, pp. 69–76.
[30] V. T'Kindt, K. Bouibede-Hocine, C. Esswein, Counting and enumeration complexity with application to multicriteria scheduling, Annals of Operations Research 153 (1) (2007) 215–234.
[31] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros, Workflow patterns, Distributed and Parallel Databases 14 (14) (2003) 5–51.
[32] W. van der Aalst, A. Weijters, Process mining: a research agenda, Computers in Industry 53 (3) (2004) 231–244.
[33] M. Bozkaya, J. Gabriels, J. van der Werf, Process diagnostics: A method based on process mining, in: International Conference on Information, Process, and Knowledge Management (eKNOW '09), 2009, pp. 22–27.
[34] G. Greco, A. Guzzo, L. Pontieri, Mining taxonomies of process models, Data & Knowledge Engineering 67 (1) (2008) 74–102.
[35] C. Li, M. Reichert, A. Wombacher, Discovering reference models by mining process variants using a heuristic approach, in: Proceedings of the 7th International Conference on Business Process Management (BPM 2009), Vol. 5701 of Lecture Notes in Computer Science, Springer, 2009, pp. 344–362.
[36] S. Fiorini, R. J. Wilson, Edge-colourings of Graphs, Vol. 16 of Research Notes In Mathematics, Pitman, 1977.
[37] I. Holyer, The NP-completeness of edge-coloring, SIAM Journal on Computing 10 (4) (1981) 718–720.