Mining Sequences for Patterns with Non-Repeating Symbols

Michal Walicki and Diogo R. Ferreira

Abstract— **Finding the *case id* in unlabeled event logs is arguably one of the hardest challenges in process mining research. While this problem can be addressed with greedy approaches, these usually converge to sub-optimal solutions. In this paper, 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. We show that for practical purposes it is possible to reduce the search space to maximal disjoint occurrences of these patterns. Experimental results suggest that, whenever this approach finds a solution, it usually finds a minimal one.**

I. Introduction

The goal of *process mining* [1], [2] is to rediscover the process model from the runtime behavior of process instances recorded in an event log. An event log contains a sequence of entries in the form $\langle case\,id, task\,id\rangle$ where *case id* identifies the process instance and *task id* specifies the task that has been performed. The event log can therefore be regarded as a labeled sequence of events, where the events from different process instances become interleaved but can be immediately separated by resorting to their case ids.

While such event logs can be easily obtained from workflow and case-handling systems, in other applications and systems that are not fully process-aware it may not be possible to retrieve event logs in that form. If the case id attribute is absent, the event log becomes an unlabeled sequence of events where it is unknown whether any two events come from the same process instance or not. However, process instances essentially repeat the sequential patterns of the business process, so in principle it should be possible to identify these patterns in the unlabeled sequence.

In previous work [3] we have shown that event logs (with case id) contain patterns that can be clustered into different groups according to their sequential behavior. Then in [4] the authors have introduced an Expectation-Maximization technique to find the case id in unlabeled event logs. The approach is based on a greedy algorithm that converges to a local maximum of the likelihood function. To improve the chances of the algorithm finding the global maximum, a special initialization procedure is used.

In this paper, we are concerned with traversing the complete search space in order to find the global optimum solution as a set of patterns that describe an unlabeled sequence. We are also concerned with the concept of *minimum description length* (MDL) [5], so we define the optimal solution as the minimal set of patterns that is able to cover the entire sequence. The principle of MDL has already been used in process mining to evaluate the quality of the mined models [6]. Here we use it as a guiding principle while looking for optimal solutions across the entire search space. To make the approach more feasible in practice, we consider the hypothesis of using only the *maximum number of disjoint occurrences* (MDOs) of each pattern. By means of experimental results, we show that this usually suffices to find a minimal solution.

Section II defines the problem, and Section III presents some facts to facilitate the treatment of MDOs. Section IV introduces the *trie*, a compact representation of pattern occurrences which is built during the initial processing of the sequence. Section V explains how to generate the MDOs for each pattern, and Section VI presents an adaptation of Knuth's algorithm X [7] to search for solutions. Section VII reports on a set of experimental results that support the hypothesis that using MDOs is usually sufficient to find a minimal solution. Section VIII describes the limiting cases where this assumption does not yield a satisfactory solution, and Section IX presents a simple practical scenario to illustrate the use of the proposed approach.

II. Problem definition

A pattern p is a sequence of length at least 2, and it occurs in another sequence S if S contains p as a *subsequence*. Two occurrences of pattern p are disjoint if they are disjoint subsequences of sequence S. We will refer to disjoint occurrences simply as DOs. For instance, the sequence $abbbaabbcc$ contains DOs such as: 1) $2 * abc + abbb$, 2) $2 * babc + ab$ or 3) $3 * ab + 2 * bc$. The problem we are considering is:

partition of a sequence into a minimal number of patterns. (2.1)

The central and distinctive point is that we do not know what patterns we are looking for. They have to be mined from the sequence as we proceed.

Various variants can be considered. Single symbols never count as patterns, but can be allowed or disallowed in the final solution. Similarly, one can allow or disallow subsequences occurring only once. The *exact variant* assumes the input sequence to be covered by repeating patterns, i.e., would exclude solutions 1) and 2) above.

Allowing, on the other hand, such "fringes" highly complicates the problem. Considerations of this variant in the present paper are motivated by the intended application to process mining. When only an arbitrary, even if large,

Michal Walicki*, Institute of Informatics, University of Bergen, HiB, 5020 Bergen, Norway (phone: +47 55 58-41-78; email: michal@ii.uib.no)

Diogo R. Ferreira, IST – Technical University of Lisbon, Avenida Prof. Dr. Cavaco Silva, 2744-016 Porto Salvo, Portugal (phone: +351 21 423 3552; email: diogo.ferreira@ist.utl.pt)

*This work was performed while the first author was visiting the Technical University of Lisbon.

segment/window of the whole sequence is analysed, one must allow for fringes which do not match any pattern within the window. Thus, neither single symbols (3 b's and a in 1) nor subsequences occurring only once (ab in 2) are patterns, but they can enter the solution as fringes, left after removal of patterns. The user can specify their admissible size.

Admitting arbitrary patterns confronts one with the problem of their sheer number, the exponential number $2^{|S|}$ of subsequences for a sequence S. This makes it computationally infeasible, even before we start checking their occurrences. As we face intractability at several stages, we leave this case aside and consider **only patterns with no repeated symbols**.

Restricting thus the patterns seems to make the problem fixed-parameter tractable in the size $m = |\Sigma|$ of the alphabet which, however, does not help much in practice. Already the number of distinct patterns is of order $\sum_{i=1}^{i=m} \frac{m!}{i!}$, which quickly becomes prohibitive as m grows. The possible number of DOs causes further problems. Given a sequence S with $n = |S|$, a pattern of length $k \leq m$ 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 $k = \frac{n}{2}$.) In short, the case of patterns with non-repeating symbols is sufficiently difficult to warrant a closer attention in this paper.

III. DOs OF PATTERNS

One of the basic tools to address problem 2.1 is counting the maximal number of DOs of a given pattern. As the following fact shows, this number can be determined in linear time, without listing the actual occurrences of the pattern. In this formulation, $S[1:k]$ denotes the substring of S from position 1 to k, and $occ(p_i, S)$ the number of occurrences of symbol p_i in sequence S. $Q \sqsubseteq S$ denotes that Q is a subsequence of S, and $occ(p, Q)$ is the maximal number of occurrences of pattern p in Q.

Fact 3.1: For a pattern p with non-repeating symbols, let P be the following sequence predicate:

$$\mathsf{P}(S,p) \Longleftrightarrow \forall_{1\leq k\leq |S|}\ \forall_{1\leq i<|p|}: \\ occ(p_i, S[1:k]) \geq occ(p_{i+1}, S[1:k])$$

Then the number of occurrences of p in S is $occ(p, S) = \max\{occ(p, Q) \mid Q \sqsubseteq S \wedge \mathsf{P}(Q, p)\}$. In other words, the maximal number of DOs of a pattern p in S is the maximal number of its occurrences in subsequences Q of S satisfying the predicate $\mathsf{P}(Q, p)$.

Example 3.2: For $S = cabbacbc$ and $p = abc$, a subsequence satisfying the predicate is, for instance, the underlined $R = \underline{ca}bba\underline{cbc} \in P(S, p)$. But it has only 1 occurence of p while $Q = \underline{cab}b\underline{acbc}$ has two, giving $occ(p, S) = 2$. The initial c violates the predicate by having no preceding b, while one of the two b's at S_3S_4 violates it since there is only a single preceding a.

PROOF: Let $n = occ(p, S)$ and $m = \max\{occ(p, Q) \mid Q \sqsubseteq S \wedge \mathsf{P}(Q, p)\}$. Obviously, $n \geq m$, since each Q is a subsequence of S, so all patterns occurring in Q occur also in S. To show that $n = m$, we consider a subsequence Q' consisting of n DOs of p. Surely this subsequence $Q' \sqsubseteq S$ must exist since S has n occurrences of p. Now, if $\mathsf{P}(Q', p)$ holds for this sequence, then $Q' \in \{Q \mid Q \sqsubseteq S \wedge \mathsf{P}(Q, p)\}$ and therefore $m = \max\{occ(p, Q) \mid Q \sqsubseteq S \wedge \mathsf{P}(Q, p)\} = occ(p, Q') = n$. So what is left to prove is that $\mathsf{P}(Q', p)$ indeed holds for Q'.

Suppose that, contrary to our belief, $\mathsf{P}(Q', p)$ does not hold. Then let k be the first index in Q' where $\mathsf{P}(Q', p)$ is violated, i.e. where $occ(p_i, Q'[1:k]) + 1 = occ(p_{i+1}, Q'[1:k]) = x + 1$, for some $1 \leq i < |p|$. Then $Q'_k = p_{i+1}$ and there are x occurrences of p_i and p_{i+1} to the left of Q'_k. But this means that one of these $x + 1$ occurrences of p_{i+1} does not belong to any DO of p in Q', since x preceding occurrences of p_i allow to form at most x such DOs using the occurrences of p_{i+1} in $Q'[1:k]$. As there are only $n - (x + 1)$ occurrences of p_{i+1} to the right of Q'_k, there are at most $n - 1$ disjoint occurrences of p in Q', contrary to the assumption that there are n such. Thus $\mathsf{P}(Q', p)$ holds for Q'. □

Fact 3.1 is telling us that, for the purpose of counting the number of DOs of a given pattern p in a sequence S, one can ignore the occurrences of symbol p_{i+1} that go beyond the number of occurrences of symbol p_i up to the same position k in the sequence. The following example illustrates this.

Example 3.3: For each position in the following sequence S, the table contains the number of occurrences of each symbol up to that position:

$S =$	a	b	b	b	a	a	b	b	c	c
a	1	1	1	1	2	3	3	3	3	3
b	0	1	2	3	3	3	4	5	5	5
c	0	0	0	0	0	0	0	0	1	1

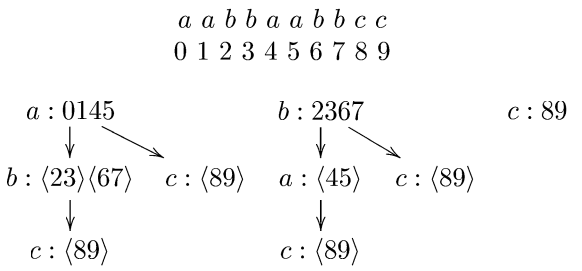
The first occurrence of b, having a preceding occurrence of a, can enter a DO of ab. So can the second b. But it cannot form a new *DO of ab, since there are not enough preceding a's. Each time, only one of the initial 3 b's can be used in the formation of 3 DOs of ab in the sequence.*

The overall approach to address problem 2.1 involves two tasks. The first one, given in Algorithm 4.3, is to build a trie with all the patterns and their occurrences in the given input sequence. This is based on Fact 3.1, and amounts to traversing the sequence once, counting the preceding occurrences of distinct symbols as in Example 3.3. The second task, given in Algorithm 6.1, is to determine the possible partitions of the input sequence into the repeating patterns stored in the trie. Section IV describes the first part while Sections V and VI describe the second part.

IV. BUILDING THE TRIE

In this part, the main data structure is a trie storing the relevant symbol occurrences for each pattern. Each node is labeled with a symbol, and a path from the root identifies a unique pattern. In addition, each node contains a list of (relevant) occurrences of its symbol. An edge $(a : A) \rightarrow (b : B)$ means that for every occurrence of b in the list B there is a preceding occurrence of a in the list A.

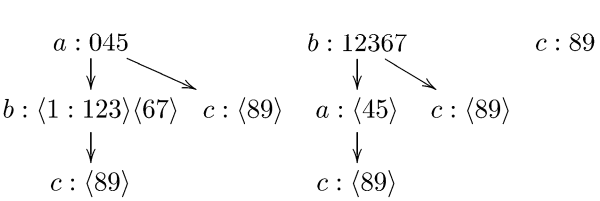
Example 4.1: $S = aabbaabbcc$ gives the following trie:



The length of the list (at the non-root nodes) tells the number of DOs of the pattern terminating there. E.g., there are two DOs of abc, signaled by 8,9 at node abc, and 4 DOs of ab, represented by 2,3,6,7 at ab. The angle brackets at node ab are used to denote that the two DOs of abc can be built either by choosing the b's at positions 2,3 or the b's at positions 6,7. In any case we are left with two remaining DOs of ab. The first branch of the trie is enough to conclude that the sequence admits the solution $2 * abc + 2 * ab$, as these DOs cover all sequence positions.

A sequence may contain some symbols not belonging to any pattern and choices may be possible when dividing the sequence into disjoint pattern occurrences. This information is also stored in the trie. Each node $T[p]$, containing the last symbol $p_{|p|}$ of the pattern p, stores its relevant occurrences – namely, the ones with fewer occurrences than of the preceding $p_{|p|-1}$. It is a list of pairs $\langle N_i : L_i \rangle$, each $N_i \leq |L_i|$ telling the number of elements to be chosen from the list L_i, e.g., $\langle 1 : 234 \rangle \langle 2 : 67 \rangle$ means there are 3 possible occurrences of the current p_j, with $N_1 = 1$ occurrence chosen from S_2, S_3, S_4 and $N_2 = 2$ occurrences from S_6, S_7. In this last case there is a single choice so, in the notation, we drop the number, writing only the list $\langle 67 \rangle$.

Example 4.2: Only one of the three initial b's can be chosen for the pattern ab or abc in the sequence below. This is marked as $\langle 1 : 123 \rangle$ in the first component list at node ab.



In the actual implementation of this data structure one must have access, for each node $T[p]$ and $1 \leq j \leq |p|$, to the j-th symbol of pattern p, $T[p][j]$, with the list of pairs $\langle N : L \rangle$, as explained above. Also, $nrOcc[p, j]$ is the sum of the N-components, i.e., for $m = |T[p][j]|$: $nrOcc[p, j] = \sum_{i=1}^{m} (T[p][j])[i].N$ ($= m$ when all $N = 1$), and $nrOcc[p] = nrOcc[p, -1]$ (where negative indexes refer to sequence positions counted from the end, e.g., $S[-1]$ denotes the last element of S.) We treat here single symbols as length-1 patterns.

Algorithm 4.3 builds the trie T from an input sequence S. The trie T is constructed while traversing the sequence once

Algorithm 4.3 BuildTrie(S:Seq)

Output: A trie with pattern occurrences in S
$\forall p \in T : nrOcc[p] =$ no. MDOs of p

```
1:  T = new Trie
2:  for each 1 ≤ i ≤ len(S) do
3:    Patts := patterns(T)
4:    if S[i] ∉ Patts then
5:      T := T + S[i]; T[S[i]][0] := ⟨1 : i⟩
6:    endif
7:    for each p ∈ Patts do
8:      if S[i] ∉ p and (p; S[i]) ∉ Patts then
9:        T := T + (p; S[i])
10:       T[p; S[i]][|p|] = ⟨1 : i⟩
11:     elseif S[i] = p then
12:       T[p][0].append(⟨1 : i⟩)
13:     elseif S[i] = p[−1] and (nrOcc[p, −2] > nrOcc[p, −1])
        then
14:       if T[p][−1][−1].L[0] > T[p][−2][−1].L[−1] then
15:         T[p][−1][−1].N := T[p][−1][−1].N + 1
16:         T[p][−1][−1].L.append(i)
17:       else
18:         T[p][−1].append(⟨1 : i⟩)
19:       endif
20:     elseif S[i] = p[−1] then
21:       T[p][−1][−1].L.append(i)
22:     endif
23:   endfor
24: endfor
25: return T
```

(line 2). Line 3 retrieves the patterns from the trie (paths from the root). For each symbol $S[i]$ several cases must be considered:

- if the symbol is not in the trie (line 4) then it is added as a new pattern, initializing also its main list (line 5);
- if it is possible to create a new pattern by extending an existing pattern p with the current symbol $S[i]$ (line 8), then the new pattern $(p; S[i])$ is added to the trie (line 9) and the corresponding list is initialized (line 10);
- if there is a length-1 pattern in the trie that is equal to $S[i]$ (line 11) then we add i as a new occurrence of this pattern (line 12);
- if $S[i]$ is the last symbol of (at least length-2) pattern p and there are enough occurrences of the previous symbol (line 13), then we add i either as a new occurrence in an existing component list (lines 15–16) or as the start of a new component list (line 18); this depends on whether i can be placed in an existing component list or not (line 14);
- finally, if $S[i]$ is the last symbol of (at least length-2) pattern but there are not enough occurrences of the previous symbol (line 20) then i is recorded as an alternative choice in an existing component list.

V. Generating the DOs

The trie constructed by Algorithm 4.3 allows to identify quickly the number of DOs and maximum number of DOs (MDOs) for each pattern. The mere inspection of the trie allows to conclude, for instance, in Example 4.1, that the sequence contains 2 DOs of abc and two more DOs of ab,

since in the trie we have $nrOcc[ab] = 4$ and $nrOcc[abc] = 2$. Summing up the number of DOs of all symbols allows in this case to conclude a complete partition of the sequence. In general, there may be several partitions and various criteria of comparison may be chosen. We therefore present the algorithm listing all possibilities for covering the original sequence with a minimal number of patterns. Since it requires inspection of all DOs of all patterns, it is hardly feasible. We describe some preliminary ways of limiting the number of alternatives to be considered by the algorithm. We also allow the user to restrict the alternatives to only the maximum number of DOs (MDOs). We conjecture that this greedy variant will usually yield a satisfactory solution.

From the trie, we construct the list of all DOs of all patterns and form the matrix PS (*patterns* $\times$ *sequence*). E.g., looking for DOs of ab in the sequence from Example 4.2, we start at the node ab which requires to make one choice among 1,2,3 and take 6,7. For every such choice, we proceed to the parent node, from which we make choices of occurrences preceding those in the child node (here, one earlier than the chosen 1, 2 or 3, i.e., 0, and two earlier than 6,7, i.e. 4,5). The selected symbol occurrences form 1's in one row of PS, for the pattern ab.

Example 5.1: The trie from Example 4.2 will give the matrix PS with the MDOs of the relevant patterns. An excerpt is presented in Table I. There are 15 ways of choosing 2 DOs of abc, 3 ways of choosing 2 DOs of bac, 3 ways of choosing 3 DOs of ab, etc. In total, there are 37 ways of choosing maximal DOs of various patterns.

TABLE I
SOME DOS OF PATTERNS IN EXAMPLE 5.1

$S =$	0 a	1 b	2 b	3 b	4 a	5 a	6 b	7 b	8 c	9 c	*no. rows*
abc	0	0	0	0	1	1	1	1	1	1	15
	1	0	0	0	1	0	1	1	1	1	
	1	0	0	0	0	1	1	1	1	1	
	1	1	0	0	0	1	0	1	1	1	
	1	1	0	0	0	1	1	0	1	1	
	1	0	1	0	0	1	0	1	1	1	
	...	...	...	...	...	...	...	...	...	...	
bac	0	1	1	0	1	1	0	0	1	1	3
	0	1	0	1	1	1	0	0	1	1	
	0	0	1	1	1	1	0	0	1	1	
ab	1	1	0	0	1	1	1	1	0	0	3
	1	0	1	0	1	1	1	1	0	0	
	1	0	0	1	1	1	1	1	0	0	
ac	0	0	0	0	1	1	0	0	1	1	3
	1	0	0	0	0	1	0	0	1	1	
	1	0	0	0	1	0	0	0	1	1	
ba	0	1	1	0	1	1	0	0	0	0	3
	...	...	...	...	...	...	...	...	...	...	
bc	...	...	...	...	...	...	...	...	...	...	10

In the above example, each row marks one *maximal* number of DOs of its pattern. This greediness does not guarantee a solution: a covering of a sequence by a minimal set of patterns may contain less than the maximal number of DOs of some pattern, as illustrated in the following example.

Example 5.2: The sequence $S = abcbacbacabc$ is a collation of abc;bac;bac;abc. The solutions with 2 patterns, having at least 2 symbols and 2 repetitions, are: 1) $2 * abc + 2 * bac$, 2) $2 * abc + 2 * cba$, 3) $2 * abc + 2 * bca$, 4) $2 * acb + 2 * bac$. However, the pattern abc used in solutions 1), 2) and 3) has 3 occurrences in S, so it is not possible to find these solutions using the MDOs of abc. In fact, it is not possible to find any solution using the MDOs of abc since, if we choose the 3 occurrences of abc, we are left with a fringe (a single occurrence of bca). In the same way, the pattern acb in solution 4) has 3 occurrences in S, so it is not possible to find solution 4) via MDOs. Disallowing fringes (i.e. patterns of length 1 or with a single occurrence), this sequence has no solution by MDOs.

In the above example, a solution can only be found if one considers a number of occurrences that is less than the MDOs. One must therefore consider *subsets of DOs* to be able to find some solutions or, in some cases, to be able to find any solution at all. The other extreme to taking only rows for MDOs is to construct the PS matrix with all subsets of DOs of each pattern, as illustrated in the following example.

Example 5.3: In the PS matrix of Table I there are 3 ways of choosing 3 MDOs of ab. In order to consider subsets of DOs one would have to include the rows for 12 ways of choosing just 2 DOs of ab, as shown in Table II. For other patterns there are no subsets of DOs to include since their number of occurrences, 2, is already minimal. However, if fringes were allowed we would have to include the rows for all possible ways of choosing a single occurrence of each pattern, and also the rows for all possible occurrences of single-symbol patterns; this would amount to a total of 154 rows.

TABLE II
DOS OF PATTERN ab IN EXAMPLE 5.3.

$S =$	0 a	1 b	2 b	3 b	4 a	5 a	6 b	7 b	8 c	9 c	*no. rows*
ab	1	1	0	0	1	0	1	0	0	0	12
	1	1	0	0	1	0	0	1	0	0	
	1	1	0	0	0	1	1	0	0	0	
	1	1	0	0	0	1	0	1	0	0	
	1	0	1	0	1	0	1	0	0	0	
	1	0	1	0	1	0	0	1	0	0	
	1	0	1	0	0	1	1	0	0	0	
	1	0	1	0	0	1	0	1	0	0	
	1	0	0	1	1	0	1	0	0	0	
	1	0	0	1	1	0	0	1	0	0	
	1	0	0	1	0	1	1	0	0	0	
	1	0	0	1	0	1	0	1	0	0	

In general, the number of rows may grow exponentially in the length of the sequence, and in some cases it may be necessary to include all rows to be able to find some particular solutions. However, as we will discuss ahead, using MDOs is usually sufficient to find a minimal solution. Our implementation allows the user to choose whether to take only MDOs or any specified number of DOs for each pattern.

VI. Solution as a set coverage problem

The matrix PS can be seen as an instance of the exact set cover problem [8]. The set to be covered is S and the DOs of patterns, the rows, represent the subsets available for partitioning S.

There are a few obvious deviations from the standard formulation of the problem. We are looking for a minimal number of patterns, and use a global integer min to keep track of this minimal value. Most importantly, we should be looking not for an *exact cover*, but for the best possible cover with some fringes allowed. This deviation could be handled by admitting additional patterns (e.g., single-symbol occurrences or non-repeating subsequences). However, these increase significantly the running time, since they must allow for covering any remaining part of the sequence. We therefore allow the user to specify the admissible size lim of the fringes and terminate the search as soon as the size of the remaining subsequence gets below this value. $lim = 0$ gives the exact cover, which is what we used in our experiments.

These deviations and other extras are incorporated into the following adaptation of Knuth's algorithm X [7] which has been implemented with "dancing links". Algorithm 6.1 takes a matrix $PS = rows \times cols$ as input and returns the solutions to the problem (2.1). The parameter $csol$ is the current solution under construction, while sol is the list of solutions generated so far and stored as the best ones. (Both are initially empty.) The main deviations from Knuth's formulation of the algorithm concern the termination condition at line 1 and associated evaluation of the solutions. If the current solution is about to surpass the min number of patterns, we terminate the recursion and backtrack, line 6.

Algorithm 6.1 Adapted algorithm X
$Part(PS, sol : Seq[Seq], csol : Seq, lim : int)$

```
 1: if |cols| ≤ lim then
 2:   if |csol| = min then  sol := sol + |csol|
 3:   elseif |csol| < min then
 4:     sol := [csol]; min := |csol| endif
 5: endif
 6: if |csol| ≥ min then return endif
 7: choose c ∈ cols with minimal number of entries
 8: for each r ∈ rows with PS[r, c] = 1 do
 9:   if pt(r) ∈ csol then
10:     skip r endif
11:   for each j ∈ cols with PS[r, j] = 1 do
12:     for each i ∈ rows with PS[i, j] = 1 do
13:       remove row i from PS
14:     endfor
15:     remove column j from PS
16:   endfor
17:   Part(PS, sol, csol + r, min, lim)
18: endfor
```

Each row r has an associated pattern $pt(r)$ and at most one row for a given pattern is considered at a time. Thus, when the row r being considered (line 8) has a pattern that is already present in the current solution (line 9), the row is skipped, i.e. it is not selected for inclusion in the current solution. The reason for this is the following:

- if PS contains only the rows for the MDOs of each pattern, there is no need to consider the same pattern again because two rows having the same pattern will not be disjoint;
- if PS contains all subsets of DOs for each pattern, there is no need to consider two rows with the same pattern since, even if they are disjoint, there will be a third row in the matrix which combines those two rows.

Lines 11–16 are just a standard implementation of Knuth's algorithm X: each column (i.e. sequence position) j that is covered by the currently selected row r is removed (line 15); also, each row i that covers the same positions as r is removed (line 13). These two steps remove the covered symbols and the incompatible DOs of other patterns. In line 17 the algorithm proceeds by trying to cover the remaining sequence positions with the available rows after these removals have taken place.

VII. Test runs

The hypothesis that using only the MDOs will provide a satisfactory result in most cases was tested on a series of randomly generated sequences. Table III presents the results, which were obtained in the two scenarios described above, namely, when PS contains all subsets of DOs, or only the MDOs for each pattern. In both cases, DOs were computed from the trie and each test was run using the following procedure:

1) generate an input sequence from a given set of patterns;
2) generate the trie for the sequence;
3) generate the PS matrix with all subsets of DOs;
4) generate the PS matrix with only the MDOs;
5) run Algorithm 6.1 on the results of steps 3 and 4, collect the solutions with minimal number of patterns, and record the execution time.
6) check the solutions against the patterns and the number of patterns that were used to generate the input sequence.

Table III shows the results of running the algorithm, implemented in Python 2.6, on a standard desktop PC with a dual-core 2.13GHz processor. It shows the execution times for the search over each matrix, over all DOs and over MDOs. Not surprisingly, while for short sequences (of length 8) the difference in search time is negligible, as soon as the sequence length reaches 22, the search over all DOs is already taking an hour to complete, while the search over MDOs is taking about one minute.

Table III shows the number of minimal solutions that were found from each matrix. From the matrix with all subsets of DOs it was possible to discover not only the original set of patterns but also additional solutions with the same number of patterns. For example, one of the sequences generated with length 10 – $aabbadbbad$ (with ab:2 bad:2) – also admitted another solution ($2 * abd + 2 * ba$). In the same example, from the matrix with MDOs, the algorithm discovered only the latter.

TABLE III
TEST RUNS

$\|S\|$	Generating Patterns	All DOs	Time (s)	MDOs	Time (s)
8	ab:2 bc:2	1	0.003	1	0.003
	abcd:2	1	0.002	1	0.002
10	ab:2 bad:2	2	0.047	1	0.021
	abcde:2	1	0.010	1	0.010
12	ab:2 bc:2 ac:2	2	0.362	1	0.025
	abc:2 cbd:2	1	0.047	1	0.024
	abcdef:2	1	0.049	1	0.047
14	ab:2 bc:3 cd:2	2	1.177	1	0.221
	abc:2 bdef:2	3	0.094	3	0.097
	abcdefg:2	1	0.210	1	0.231
16	ab:2 bc:3 cd:3	3	13.76	0	0.962
	abcd:3 cb:2	1	29.23	0	1.184
	ab:4 cd:4	1	6.84	1	0.392
18	ab:3 bc:4 cd:2	2	146.4	1	3.095
	abc:3 cbd:3	2	102.5	1	4.274
	abcd:3 de:3	4	19.6	4	3.150
20	ab:2 bc:3 cd:3 de:2	10	580.8	1	35.80
	abc:4 cdef:2	1	176.5	1	39.94
	abcde:4	1	713.8	1	12.03
22	ab:2 bc:3 cd:3 de:3	6	3601	2	49.39
	abc:3 cde:3 bd:2	7	3786	2	72.03
	abcd:4 de:3	1	5689	1	53.16

In most cases, using MDOs is sufficient to discover at least one minimal solution. However, there were two cases of length 16 where this did not happen; these are marked by the two zeros in Table III. As we saw in Example 5.2 the search over MDOs may fail to provide any solution, even though solutions do exist. Such failure is due to the fact that two patterns, entering a solution, combine to generate new occurrences of the same or other patterns, in addition to the occurrences that were used to generate the original sequence. In Example 5.2 the two occurrences of bac form a third occurrence of abc; thus, when searching for a solution by MDOs, this additional occurrence of abc collides with the occurrences of bac. In the two cases of length 16 reported in Table III, the search over MDOs did not find any solution at all. In general, this may happen if the patterns of a sequence share common symbols and are interleaved in such a way that their MDOs collide (i.e. there is no disjoint set of MDOs covering the entire sequence).

When the search over MDOs yields a non-emtpy set of solutions, there still remains the issue of whether any of those solutions is a minimal solution. Obviously, any non-empty set of solutions contains at least one solution which is minimal *in that* set. But the issue is whether, in fact, there is a *globally* minimal solution in that set, i.e., minimal among all possible solutions, including those found by using all subsets of DOs. In our test runs we have observed that whenever the search over MDOs is able to retrieve a non-empty set of solutions, this set contains at least one globally minimal solution. However, this cannot be expected to occur in every situation, as explained in the next section.

VIII. LIMITATIONS OF USING MDOS

The test runs reported in the previous section support the idea that sometimes there is no solution by MDOs but when there is, it is often a minimal one. The absence of solutions by MDOs has to do with the restrictions to patterns of at least length 2 and with at least 2 occurrences. When such fringes are allowed, there is *always* a solution by MDOs. In the worst case, it contains single symbols or single-occurrence patterns, as illustrated in the following example.

Example 8.1: The sequence abcdcdbcabbcbccd has been generated from ab:2 bc:3 cd:3. Searching by MDOs finds no solution. However, by allowing patterns of length 1 it is possible to find the solution $2*a+5*b+6*c+3*d$.

The set of generating patterns in Example 8.1 has an interesting feature: the occurrences of ab and cd can combine to generate additional occurrences of bc. This means that the MDOs for bc will have more than 3 occurrences. The additional occurrences of bc will collide with the MDOs for ab and cd, since they are constructed from occurrences of these patterns. Therefore, the solution by MDOs cannot contain the pattern bc, and the algorithm must select other patterns. For some of the input sequences generated from ab:2 bc:3 cd:3, the result is that the solutions found by MDOs have more than 3 patterns. This is illustrated in the following example.

Example 8.2: For the sequence abcdcdbcabbcbccd generated from ab:2 bc:3 cd:3, Table IV shows the MDOs for the generating patterns. (There are other patterns, but only the generating ones are shown.) As can be seen, there are 5 occurrences of bc, while to generate the sequence only 3 occurrences have been used. If the algorithm selects pattern bc, it will be unable to select ab or cd since the MDOs for these patterns collide with those of bc. Searching MDOs, the only solution that can be found is $2*a+5*b+6*c+3*d$ *with 4 patterns, by allowing fringes. Searching over all subsets of DOs (with fringes) it is possible to find the following solutions with 3 patterns:*

$2*ab+3*bc+3*cd$
$2*abcd+3*bc+1*cd$
$2*abcd+3*bc+1*dc$
$2*abcd+3*cb+1*dc$

TABLE IV
THE GENERATING PATTERNS, EXAMPLE 8.2.

a	b	c	d	c	d	b	c	a	b	b	c	b	c	c	d
a	b							a	b						
a	b							a		b					
a	b							a				b			
a						b		a	b						
a						b		a		b					
a						b		a				b			
a								a	b	b					
a								a	b			b			
a								a		b		b			
	b	c				b	c		b	b	c	b	c	c	
	b			c		b	c		b	b	c	b	c	c	
		c	d	c	d		c								d
		c	d	c	d						c				d
		c	d	c	d								c		d
		c	d	c	d									c	d

$2 * acd + 4 * bc + 1 * db$
$2 * ad + 5 * bc + 1 * cd$
$2 * ad + 5 * bc + 1 * dc$
While searching over MDOs retrieved a 4-pattern solution, there are several possible 3-pattern solutions.

These examples show that using MDOs is not guaranteed to produce a globally minimal solution. To illustrate this, we have used examples where we have allowed fringes to occur, but the same phenomenon can be observed in longer sequences without using fringes. However, in practical applications where *the generating patterns do not combine to yield additional occurrences of other patterns*, the MDOs for the true patterns will not collide, and therefore it will be possible to find a minimal solution by MDOs.

IX. A SIMPLE BUSINESS PROCESS

The process in Figure 1 illustrates a simple purchasing scenario for a chain of retail stores. Whenever a store is out of stock of some item, it will submit a request to the back office for processing. Upon receiving the request, the back office checks products, quantity and other details such as previous requests from the same store. If the request meets certain criteria, it is sent to the warehouse. The warehouse checks the inventory for the requested products and, if some product or quantity is unavailable, places an order to a supplier. In principle the order will be delivered with all items but if this does not happen, the warehouse will have to check with the supplier when the remaining items will be received. The products are then shipped to the store.

If the process is performed as described, it is capable of generating patterns such as ab, acd, $acefh$ and also $acefgfh$, $acefgfgfh$, $acefgfgfgfh$, etc. First we restrict our analysis until step e. For sequences comprising instances of ab, acd and ace, the true solution can be found with relative ease by MDOs. Also, additional solutions may be found. For example, for the sequence $aabceaaaaacdcabcdeaccbde$ (ab:3, acd:3, ace:3) the solution $3 * ab + 3 * acd + 3 * ace$ is found, as well as $6{*}ac{+}3{*}ad{+}3{*}be$ and $6{*}ac{+}3{*}ae{+}3{*}bd$. These additional solutions show the tendency of MDOs to capture subsequences (such as ac) that are common to several patterns (e.g. acd and ace) and which have more occurrences than any of those patterns alone.

Now, if we consider steps f, g and h, it may be that the loop does not occur (and the pattern $acefh$ is produced) or, if incomplete orders arrive, the process generates patterns in the form $acefgfg \ldots fgh$ with any number of fg's between ace and h (although the likelihood of having more than 3 or 4 repetitions of fg can be assumed to be rather low in this scenario). In this case the process generates patterns with repeating symbols. However, for sequences comprised of occurrences of $acefgfh$, $acefgfgfh$, etc., searching by MDOs will return solutions containing patterns $acefh$ and gf, or $acefh$ and fg, depending on whether $acefh$ picks up the f closest to e or the f closest to h. This means that *even though our approach does not handle repeating symbols, it will effectively capture the execution of loops as independent occurrences of the steps within those loops.*

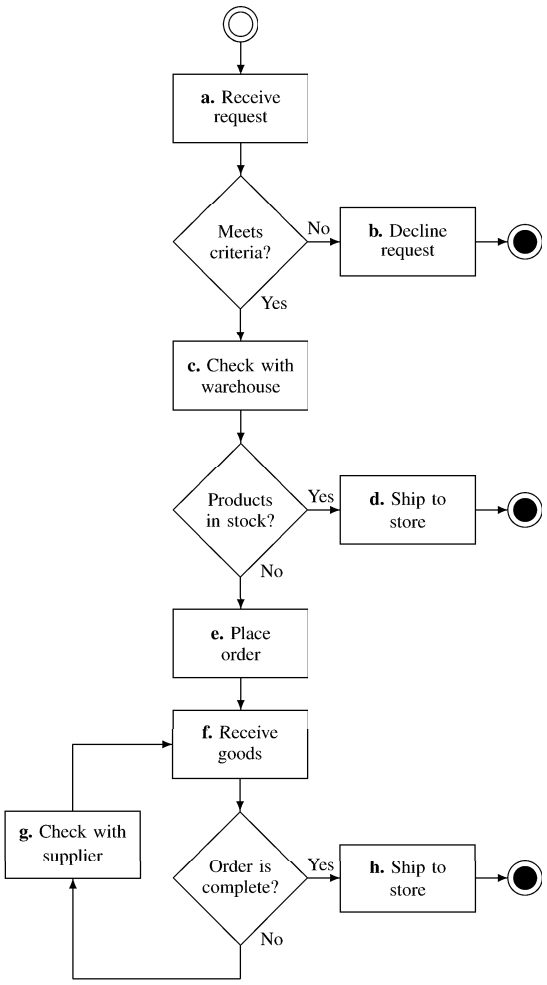


Fig. 1. A purchasing scenario

On the other hand, there are situations in which it is not possible to recover the original patterns by MDOs. The process in Figure 1 contains two steps – d and h – that are performed in different contexts, but are very similar in nature. These two steps could be represented as a single step without affecting the process logic, by removing step h and connecting its incoming arc directly to step d. The process would then generate patterns such as ab, acd, $acefd$, $acefgfd$, etc. In this case, however, these patterns cannot be recovered using MDOs, since the pattern acd is a subsequence of patterns $acefd$ and $acefgfd$ (and $acefd$ is a subsequence of $acefgfd$, and so on). The effect is that, in a sequence comprising occurrences of acd and $acefd$ for example, the pattern acd will have more occurrences than originally expected. The search by MDOs will consider all disjoint occurrences of acd and hence, if this pattern is included in the solution, the algorithm will be unable to capture $acefd$ and other patterns that contain acd as a subsequence. The algorithm may, however, be able to find solutions containing other patterns.

One last issue to consider is the assumption of exact coverage. In practical applications it may not be possible to

extract a sequence that comprises complete occurrences of the patterns to be discovered. For example, if the sequence $aabceaaaaacdcabcdeaccbde$ (ab:3, acd:3, ace:3) is missing the first and last symbol ($S' = abceaaaaacdcabcdeaccbd$), then the algorithm will find the solution $6{*}ac{+}3{*}bd{+}2{*}ea$, but not the original patterns. Trying to cover the sequence S' with $2 * ab + 3 * acd + 2 * ace$ results in one stray b and one ac, which by definition are not patterns. If one goes further and removes 2 symbols at both ends ($S'' = bceaaaaacdcabcdeaccb$) then there is no solution by MDOs. It could be that such truncation of the sequence leaves only complete occurrences of patterns of the original patterns, in which case searching by MDOs will find the desired solution; but this can only occur due to a fortunate coincidence. In general, such truncations will result in fringes.

Although our implementation allows a fringe to be specified (in terms of number of symbols), the use of a fringe with MDOs may not produce the expected result, as the following example demonstrates:

Example 9.1: Sequence $S' = abceaaaaacdcabcdeaccbd$ has been obtained from $S = aabceaaaaacdcabcdeaccbde$ (ab:3, acd:3, ace:3) by truncating the first and the last symbol of S. In principle, one would like to find the desired solution $2 * ab + 3 * acd + 2 * ace + b + ac$ where b and ac are the fringes created by the removal of the first a and the last e from S. Searching over MDOs and allowing a fringe of 3 symbols, one finds several solutions such as:

$3 * ab + 3 * acd + 2 * ce + 2 * a + c$
$2 * ab + 3 * acd + 2 * ace + a + b + c$
$2 * abc + 3 * acd + 2 * ea + a + b + c$
...

However, no solution can contain ac as a fringe, since this pattern has 6 disjoint occurrences in S'. Hence, the desired solution cannot be found by MDOs.

To cope with possible truncations of the input sequence, one may have to search using all subsets of DOs, in addition to specifying a certain fringe. This, however, becomes computationally expensive. A quick look at the solutions found in Example 9.1 shows that the search by MDOs is able to find two out of three original patterns. Moreover, the solutions have the same number of patterns as the desired one (3, since the fringes do not count; actually, searching by MDOs produces the pure 3-pattern solution $6{*}ac{+}3{*}bd{+}2{*}ea$, but this is not being counted as well since it does not contain any of the original patterns). These and other experiments support the claim that searching by MDOs is usually sufficient to find a satisfactory solution, at a fraction of the cost of searching all subsets of DOs.

X. Conclusion

In this paper we have addressed the problem of finding a minimal set of patterns to describe the sequence of events recorded in an unlabeled event log. While traversing the complete search space is necessary to find all solutions, our experiments suggest that, in most cases, it suffices to search only the maximal number of disjoint occurrences (MDOs) for each pattern. This usually yields a minimal solution.

We have described a data structure (the trie) that can be built in linear time to store the occurrences of each pattern. We have shown how to retrieve the MDOs from the trie. Also, we have adapted Knuth's algorithm X to traverse the search space. Finally, we have identified typical cases when searching by MDOs is not guaranteed to yield a solution.

These contributions provide confidence that, in the future, it will be possible to recover the *case ids* from unlabeled event logs without being restricted to sub-optimal solutions and without having to traverse the complete search space in order to find a globally optimal one.

References

[1] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters, "Workflow mining: A survey of issues and approaches," *Data and Knowledge Engineering*, vol. 47, no. 2, pp. 237–267, 2003.

[2] A. Tiwari, C. Turner, and B. Majeed, "A review of business process mining: state-of-the-art and future trends," *Business Process Management Journal*, vol. 14, no. 1, pp. 5–22, 2008.

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

[4] D. R. Ferreira and D. Gillblad, "Discovering process models from unlabelled event logs," in *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 2009. Proceedings*, ser. Lecture Notes in Computer Science, U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, Eds., vol. 5701. Springer, 2009, pp. 143–158.

[5] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, pp. 465–471, 1978.

[6] T. Calders, C. W. Günther, M. Pechenizkiy, and A. Rozinat, "Using minimum description length for process mining," in *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 1451–1455.

[7] D. E. Knuth, "Dancing links," in *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Palgrave Macmillan, 2000, pp. 187–214.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.