
Production Systems and Information Engineering
Volume 10 (2), pp. 99–113
doi: 10.32968/psaie.2022.2.7.

ANALYSIS OF THE MAXIMAL PATTERN MINING METHOD AND ITS VARIANTS

Dávid Gégény
University of Miskolc, Hungary
Institute of Mathematics
matgd@uni-miskolc.hu

Sándor Radeleczki
University of Miskolc, Hungary
Institute of Mathematics
matradi@uni-miskolc.hu

Abstract. In this paper, within the framework of process mining we examine the Maximal Pattern Mining method introduced by Liesaputra et al. in [1]. This method constructs a transition graph, i.e. a labelled directed graph for traces with similar structure. The idea behind the algorithm is to analyze the traces in the event log, identify loops, parallel events and optionality between them, in order to determine the maximal patterns. In [1], the authors provide a pseudo code for the skeleton of their algorithm and discuss some parts, but other parts are not detailed. Here, we briefly discuss the steps of the algorithm and elaborate the steps that are not explained in [1]. We introduce some new subroutines to handle the loops, parallel and optional sequences.

Keywords: event log, trace, pattern, workflow graph, deterministic finite automaton

1. Introduction

Automated process mining refers to the analysis and control of internal processes within a business or customer service via automated systems. This topic is closely related to algorithms using artificial intelligence. There are several approaches, including graph based solutions, such as the *Maximal Pattern Mining* (MPM) method introduced in [1], which is the main topic of our paper. Another approach uses finite subsequential transducers to recognize patterns, see e.g. [2], which provides the basis for the *OSTIA* (onward subsequential transducer inference algorithm) method. These methods construct a transition graph for traces with similar structure, which is a labelled directed graph

called *transaction pattern*. The α-algorithm [3] and its variants, i.e. the $\alpha++$-algorithm [4] also use this approach. The MPM algorithm [1] investigated in our paper is based on the α-algorithm, as well. There are also methods based on context-free grammars, where typical process sequences are retrieved from a tree structure, see e.g. [5], [6]. There are approaches based on recurrent neural networks, text recognition and text generation, see e.g. [7].

In practice, a customer care system should provide answers (solutions) to incoming customer requests. These requests generally contain the type of the request, some identifier of the person making the request, a timestamp and other significant data. The response is a sequence of events, a so-called *trace*, The *event log* is a collection of traces containing such responses. Formally, a trace t is a finite sequence of events $t = (z_1, ..., z_m)$ and an event log T is a non-empty set of traces $T = \{t_1, t_2, ..., t_n\}$. A trace t is treated as a sequence of coherent events ordered by their timestamp. If only the type of events are considered in a trace, then the trace itself can be treated as a word over a finite alphabet, e.g. $t = (a, b, b, c, e, d)$. In this case, the event log can be considered (a subset of) a language over this alphabet.

The task is to analyze the traces and distinguish real traces from *noise* that can be a result of faulty or incomplete recording of events, and to generate traces automatically that could be real customer service events. The first process mining approaches were derived from data mining and pattern recognition methods, see e.g. [4] and [8]. Cook and Wolf [9] proposed a data based statistical process mining method. Agrawal et al. [10] suggested a graph based solution to represent workflow processes. Mannila and Meek [11] proposed a method based on describing global partial orders on events making up parts of the event sequence. A significant step forward was the introduction of the α-algorithm by van der Aalst et al. in [3], later improved by Alves de Medeiros et al. in [12] named the $\alpha+$-algorithm. Later, Wen et al. [4] further developed it into the $\alpha++$-algorithm. This method is robust enough not to be influenced by small changes in the input, however, it cannot handle loops, alternative routes and parallel events well. A heuristic variant [13] and a variant using fuzzy classification [14] were also developed.

Finite transducer automata with output have successfully been applied to represent workflow processes even in the '90s. One of the most well-known algorithms is the OSTIA [2] by Oncina, García and Vidal. Another method using finite automata and Petri nets was introduced by van der Aalst et al. [15]. The internal business event sequences can also be described as the words of a context free grammar (CFG) [5] [6]. Genetic algorithms can also be used for this task, for example the DT Genetic Miner [16], but their time complexity is not favorable (see the analysis in [1]). In the last decade, the analysis and

generation of business processes were carried out by automatic text recognition and generation tools using recurrent neural networks, see e.g. [7], [17], [18] and [19].

Here, we analyze some variants of the maximal pattern mining method, describing the implementation possibilities of the parts that were not discussed in detail in the original paper. The structure of the paper is as follows. In Section 2, we introduce the main idea behind maximal pattern mining based on [1]. In Section 3, we detail and analyze some implementation possibilities for specific steps of the algorithm. In Section 4 we try to evaluate the obtained results of our work, presenting some conclusions.

2. Overview of Maximal Pattern Mining

The *Maximal Pattern Mining* (MPM) method was introduced by Liesaputra et al. in [1]. The idea behind the algorithm is to analyze the traces in the event log, identify loops, determine which patterns are maximal, and identify parallel events and optionality between them. The result of the algorithm is a set of maximal patterns that are, in essence, regular expressions over the alphabet of events fitting the traces. A finite deterministic automaton can be assigned to these regular expressions. In [1], the authors provide a pseudo code for the skeleton of the algorithm and discuss some parts, but other parts are not detailed. Here, we briefly discuss the steps of the algorithm and elaborate the steps that are not included in [1].

Let $T = \{t_1, t_2, ..., t_n\}$ be an event log as the input of the MPM algorithm. First, we identify loops in traces. Loops could contain a single event type, or they can contain sequences of event types. Since smaller loops can be a part of larger loops, this step is performed in multiple iterations. For example, the trace $t = (a, b, b, b, c, b, b, c, d, e)$ would first be transformed into the pattern $p_1 = (a, LOOP(b), c, LOOP(b), c, d, e)$, then into $p_2 = (a, LOOP(LOOP(b), c), d, e)$. The number of traces fitting a given pattern is called the *support* of the pattern.

The next step in the algorithm is identifying frequent events and frequent patterns based on the event log. This happens by providing *threshold* parameters. If the number (or percentage) of the traces where an event type occurs does not reach the predefined threshold value, we treat that event type and the corresponding traces as noise and exclude them from the further steps of the algorithm. A similar filtering can be performed on infrequent patterns, where the support value of the pattern must reach a certain threshold value (or must reach a threshold percentage of the number of traces). The original description in [1] builds up the pattern from frequent events until the support of the pattern does not go under the threshold value. Since the patterns can

be treated as regular expressions, simply counting the matching traces also works.

Some events in the traces could be treated as parallel events, i.e. their order does not matter. Therefore, we need to identify these parallel events in the patterns to get a more accurate representation of the business process. For example, the two patterns $p_1 = (a, b, c, d, e)$ and $p_2 = (a, b, d, c, e)$ are nearly identical, the only difference being the order of event types c and d. Since this similarity indicates that they represent the same process, c and d can be treated as parallel events. Thus, p_1 and p_2 can be combined into a single pattern $p = (a, b, AND(c, d), e)$. This step is performed after identifying loops, because loops can also be included in parallel elements. Another important consideration is that we might want to perform another check for loops, as the parallel events can introduce new repetitions into the pattern. For example, the patterns $p_1 = (a, LOOP(b), c, d, LOOP(b), d, c, e)$ and $p_2 = (a, LOOP(b), d, c, LOOP(b), c, d, e)$ could be combined into $p = (a, LOOP(b), AND(c, d), LOOP(b), AND(c, d), e)$. Then, a new loop arises and the pattern can be transformed into $p' = (a, LOOP(LOOP(b), AND(c, d)), e)$. However, identifying that p_1 and p_2 have parallel events at two separate points at once can make the implementation more complex. Hence, it may suffice to treat them as separate patterns.

In the next step, we determine which patterns are maximal. We say that a pattern p_1 is the *subpattern* of p_2 if for all traces t in the event log T, p_1 covers t implies p_2 covers t. We say that p is a *maximal pattern* in the constructed set of patterns if and only if it is not a subpattern of any other patterns. Since the patterns are essentially regular expressions, a finite automaton can be constructed to check which set of traces they cover. From there, determining maximal patterns is as simple as determining maximal sets.

Next, we determine optionality in the patterns. This is somewhat similar to parallel events as we look for matches at the beginning and end of the patterns (in some cases, only the beginning or only the end matches). The sequences between the matching prefix and suffix are combined into a XOR element. These matches are examined pairwise between the maximal patterns, making sure that the patterns with the longest matching parts are combined first. Here, we can also define a lower bound for the length of the matching parts, so separate processes will have different patterns assigned. If no threshold is given, all the maximal patterns will be combined into one single pattern. If it is also known what the request was for a specific trace response, it can also be used to determine whether or not we allow combining relevant patterns with optionality. Finally, the transition graph (finite automaton) is constructed

based on the maximal patterns. If no threshold is given for the matching length during detecting optionality, then the result will be a single transition graph.

3. Loops, parallel events, optionality

As mentioned earlier, in [1], some of the steps are not discussed in detail and pseudo-code is not given for them. The concrete implementation of these steps may depend on specific requirements. In this section, we provide a description for a possible implementation. The three parts to be discussed are detecting loops, identifying parallel events and solving for optionality.

3.1. Loops

The first step is going through all the traces and identifying loops. In the pseudo-code found in [1], this step is referred to as the "Solve Loop" subroutine, however, no detailed description is given. The input of the subroutine is a trace and its output is a pattern where loops are noted. In the following, we start indexing at 0 and we will refer to segments of a trace (or pattern) by giving the start index and end index. For example, consider the trace $t = (a, b, b, c, b, c, d, e)$, where $t[1..5] = (b, b, c, b, c)$. Our proposed algorithm handles loops as sets of intervals given by their start and end indices. We denote the resulting pattern as p and the resulting set of loops as L. For the previously mentioned example trace, the pattern would be $p = (a, b, c, d, e)$ and the set of loops would be $L = \{(1, 1), (1, 2)\}$. Though these are closed integer intervals, here we use regular brackets, so it does not get mixed up with the index notation.

Our proposed algorithm is iterative, incrementing the length of the examined loops, i.e. first we look for loops of length 1, then length 2, etc. Since no loop can be longer than half the length of the pattern itself, this continues until we reach that limit. This algorithm uses a *"sliding window"* that moves from left to right on the pattern (or trace). Wherever the window is, we check whether or not the content of the window repeats afterwards. If so, the repeating part is added (only once) to the result pattern and the start and end indices of the window are added to the set of loops.

Now, let us consider the example trace $t = (a, b, c, b, b, c, d, e)$. After the first iteration, the resulting pattern would be $p_1 = (a, b, c, b, c, d, e)$ and the set of loops is $L = \{(3, 3)\}$. In the next iteration, we find that the elements b and c in the pattern are in a loop with length 2. Therefore the pattern would become $p_2 = (a, b, c, d, e)$ with the looping interval $(1, 2)$ added. However, the existing $(3, 3)$ interval would be incorrect, because element b repeats there, not element d. We can see that in some cases, the loops contained in a repeating

part ("subloops") need to be translated into the current window in order to correctly represent repetitions. Let $I_w = (w_s, w_e)$ denote the interval of the current window and $I_c = (c_s, c_e)$ denote the repeating found afterwards. Let $I_l = (l_s, l_e)$ be a loop from the previous iteration such that $I_l \subseteq I_c$. Then the interval I_l will be translated into I_w using the following calculation, giving I_r as the result interval:

$$I_r = (w_s + l_s - c_s, w_e + l_e - c_e). \tag{3.1}$$

Note that this interval must *not overlap* with already existing intervals in the window. The intervals $I_1, I_2 \in L$ are *not overlapping (not colliding)* if the following condition holds:

$$I_1 \subseteq I_2 \text{ or } I_2 \subseteq I_1 \text{ or } I_1 \cap I_2 = \emptyset. \tag{3.2}$$

As an example, consider the following result after the second iteration: $p = (a, b, c, d, b, c, d, e)$ and $L = \{(1, 2), (5, 6)\}$. In the next iteration, we consider loops of length 3. At window $(1, 3)$, we find that the sequence (b, c, d) repeats. However, translating the existing $(5, 6)$ loop into the window, we get $(2, 3)$, which overlaps with $(1, 2)$, a subloop of the current window. This means that (b, c, d) cannot be considered a new loop, and the window should slide to the right as if no loops were found. The pseudo-code of detecting loops is given in Algorithm 1.

3.2. Parallel events

As a next step the algorithm detects events that can be executed in parallel. Here it is important to draw our attention to one of the peculiarities of the traces: the concept of a series of events or a word on an alphabet gives only an approximate definition of the notion of a trace, because within a trace (that is created as an imprint of event types in an administrative workflow), the order of some events may be irrelevant, i.e. they can be executed in any order within a given process. For instance, in case of the traces (a, b, b, b, c, d, e) and (a, b, b, d, c, e) the order of the events events c and d can be arbitrary, therefore, both traces can be subordinated to a single pattern, namely to $(a, LOOP(b), \{c, d\}, e)$. The set notation $\{c, d\}$ indicates that the order of c and d can be chosen arbitrary - another frequently used notation is $(a, LOOP(b), AND(c, d), e)$. The finite automaton corresponding to this pattern can be found on Figure 1.

If there are several events (e.g. b, c, d) to decide whether they are parallel, it is sufficient to check that the order of any two events can be interchanged. Thus, the scheme $(a, \{b, c, d\}, e)$ can only be legitimately created if the traces $(a, b, c, d, e), (a, c, b, d, e), (a, d, c, b, e), (a, b, d, c, e)$ (or their versions with some

Algorithm 1 Algorithm solving for loops in a trace or pattern

```
SolveLoop(input) :
p ← input
loops ← ∅
for len ∈ {1, ..., |p|} do
    newp ← ∅, newloops ← ∅, i ← 0
    while i < |p| do
        w ← p[i .. i + len − 1]
        cloop ← NULL
        firstIter ← true
        ni ← i + len
        while TRUE do
            next ← p[ni .. ni + len − 1]
            checkInt ← (ni, ni + len − 1)
            tloops ← ∅
            if cloop = NULL then
                candidate ← (|newp|, |newp| + len − 1)
            else
                candidate ← cloop
            for loop ∈ {x ∈ loops | x ⊆ checkInt} do
                tloop ← translate loop from checkInt into cloop
                if tloop does not overlap with any subloops of cloop then
                    tloops ← tloops ∪ {tloop}
            if w = next AND there were no collisions then
                if firstIter then
                    newp.add(w)
                    cloop ← candidate
                    newloops ← newloops ∪ {candidate}
                    newloops ← newloops ∪ {x ∈ loops | x ⊆ candidate}
                newloops ← newloops ∪ tloops
                ni ← ni + len
                firstIter ← FALSE
            else
                if firstIter then
                    newp.add(w[0])
                    i ← i + 1
                else
                    i ← ni
                EXIT while
    p ← newp
    loops ← newloops
return p, loops
```

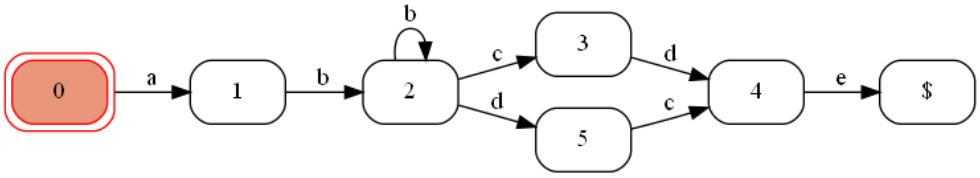

Figure 1. Example parallel events c and d

loops) all occur in the event log. This is enough because any permutation can be induced by the transposition of its elements. (The good news, however, is that in the case of a sequence with k elements we only need to find $\binom{k}{2} = \frac{k(k-1)}{2}$ of the $k!$ possible permutations.) It is also possible that the order of an event and of a "short" sequence of events can be interchanged. For example, in case of the traces (a, b, c, f, d, e) and (a, b, c, e, f, d) the pattern $(a, b, c, \{(f, d), e\})$ covers both. So we can also talk about the "parallelization" of sequences (groups) of events, although their length should be kept below a reasonable threshold. (Because in an administrative process, the order of events within longer series are usually not incidental!) In Liesaputra et al. paper [1], the exploration of parallel event groups is performed by an algorithm called Solve Concurrency. We modified this algorithm allowing parallel sequence with a length not more than four - in this way we were able to avoid also multi-composed parallel groups. This modification was also reducing the number of the errors occurring during the tests: the sample created with a trace generator gave us quite good running results with a Re value higher than 0.95.

The basic procedure is as follows. The input of the procedure is two samples of patterns, between which the algorithm identifies parallelism. If there is no possibility to indicate parallelism between them, the algorithm returns both samples. If there is a parallel sequence between the two patterns, it returns a single merged pattern as the obtained result. The pseudo-code is shown in Algorithm 2.

Algorithm 2 Algorithm solving for concurrency between two patterns

SolveConcurrency$(p, q, threshold)$:
if $|p| \neq |q|$ **then**
    return p, q
$start \leftarrow -1$
$end \leftarrow |p|$
for $i \in \{0, 1, ..., |p| - 1\}$ **do**
    if $p[i] \neq q[i]$ **then**
        $start \leftarrow i - 1$
        EXIT for
for $i \in \{|p| - 1, |p| - 2, ..., 0\}$ **do**
    if $p[i] \neq q[i]$ **then**
        $end \leftarrow i + 1$
        EXIT for
if $start > end$ **then return** p, q
for $i \in \{start + 1, ..., end - 1\}$ **do**
    $cp \leftarrow 0$
    $cq \leftarrow 0$
    for $j \in \{start + 1, ..., end - 1\}$ **do**
        if $p[i] = p[j]$ **then**
            $cp \leftarrow cp + 1$
        if $p[i] = q[j]$ **then**
            $cq \leftarrow cq + 1$
    if $cp \neq cq$ **then**
        return p, q
if $end - start - 1 < threshold$ **then**
    return $(p[0], ..., p[start], AND(p[start{+}1], ..., p[end{-}1]),$
    $p[end], ..., p[|p|{-}1])$
else
    return p, q

3.3. Optional sequences of events

Here we would like to recall the fact that a customer service system is built to respond to certain incoming request. If $p_1 = us_1v$, $p_2 = us_2v$ and $p_3 = us_3v$ are traces having the same prefix u and the same suffix v, which represent all possible responses to the same request r, then then this means that the sub-sequences s_1, s_2 and s_3 are interchangeable and the Administrator can choose any of them to obtain a correct solution for the request. In this case we say that these sub-sequences are "optional" and the above three traces can be merged in a scheme $p := (u, XOR(s_1, s_2, s_3), v)$ that covers all three of them. For instance, the traces $(a, i, b, c, d), (a, i, f, d), (a, i, g, d)$ corresponding to a request r_0 express optional possibilities and they can be merged in the pattern $p = (a, i, XOR(bc, f, g), d)$. Here we made two changes to the original "Solve Optionality" method known from [1].

1. We allowed the so-called ε-movement in case of the obtained finite automata.
2. Since several optional sections are possible within a pattern, we gave a lower bound to how similar the prefix and suffix parts combined must be - it must exceed one third of the average trace length. Another restriction is that the length of the optional sections cannot be longer than 4 (can not contain more than four events).

The justification for these changes is the following.

1. makes possible to merge the traces $(b, c, b, b, b, b, c, b, b, c, b, b, b, c, b, c, d, e, f)$ and $(a, b, c, b, c, b, c, c, d, c, d, e, f)$ under the common pattern

$(XOR(\varepsilon b, ab), LOOP(LOOP(b), c), XOR(\varepsilon, cdc), d, e, f)$. In addition, the recombination of "branches" at the end of optional (or parallel) event sequences can be handled by adding an ε-movement to the end of each branch into a single state representing the start of the next sequence. This also solves the problem of an optional (or parallel) sequence ending in a loop, where generating an automaton without ε -movement would needlessly complicate the implementation. Furthermore, it is known that every finite automaton with ε-movement is equivalent to a finite automaton without ε-movement.

2. is motivated by the fact that in everyday practice, event sequences that differ too much (e.g. more than the above mentioned) belong in general to different requests.

Algorithm 3. shows the pseudo-code for handling optional sequences of events. We tested the modified algorithm with a training set created with a trace generator by using the Recall (Re) and Precision (P) metrics to evaluate

the effectiveness of the tested subroutines. We obtained results with Re and P values higher than 0.95.

Algorithm 3 Algorithm solving for optionality in patterns

SolveOptionality$(plist, threshold)$:
if $|plist| = 1$ **then**
  return $plist$
$starts \leftarrow [\,][\,]$
$ends \leftarrow [\,][\,]$
for $i \in \{0, 1, ..., |p| - 2\}$ **do**
  for $j \in \{i + 1, i + 2, ..., |p| - 1\}$ **do**
    $maxindex = min(|plist[i]|, |plist[j]|)$
    for $k \in \{0, ..., maxindex - 1\}$ **do**
      if $plist[i][k] \neq plist[j][k]$ **then**
        $starts[i][j] \leftarrow k$

        EXIT for
    for $k \in \{maxindex - 1, ..., 0\}$ **do**
      if $plist[i][k] \neq plist[j][k]$ **then**
        $ends[i][j] \leftarrow k$

        EXIT for
$u, v = argmax\{starts[u][v] + ends[u][v] \mid u, v \in \{0, ..., |plist|\}, u < v\}$
if $starts[u][v] + ends[u][v] \leq threshold$ **then**
  return plist
else
  $s \leftarrow starts[u][v]$
  $e \leftarrow ends[u][v]$
  $result \leftarrow$ take first s elements of $plist[u]$
  $x \leftarrow plist[u]$
  Remove first s elements from x
  Remove last e elements from x
  $y \leftarrow plist[v]$
  Remove first s elements from y
  Remove last e elements from y
  Append $XOR(x, y)$ to $result$
  Append last e elements of $plist[u]$ to $result$
  Remove $plist[u]$ and $plist[v]$ from $plist$
  Add $result$ to $plist$
  return $SolveOptionality(plist, threshold)$

4. Evaluation of the obtained results, conclusions

We implemented our variant of the Maximal Pattern Mining algorithm in *python* [20] using the following libraries:

- *numpy* [21] – used for making mathematical calculations easier;
- *itertools* [22] – used to generate permutations in order to check parallel events;
- *transitions* (also known as *pytransitions*) [23] – used to handle finite state machines and build automata;
- *graphviz* [24] – used to visualize the transition graphs for automata.

For our tests, we created example event logs manually and used regular expression based trace generation as well. We found that our implementation handled loops reliably, could identify parallel events and could also handle optional events with or without any restriction on the similarity of patterns to be combined.

In the future, we intend to perform a more thorough benchmark on the algorithm, where the event logs also include what type of request induced a given response trace from the system. This way we can test if the algorithm can be used to generate sample responses for a given request. We would also like to test the algorithm on a large amount of real-world data, ensuring applicability in real-world situations. Furthermore, the algorithm should also be able to classify an unknown trace into request categories based on the sequence of event types. Another promising related future research is to enhance the algorithm with neural network based learning as well.

Figure 2. shows the generated finite automaton for the event log

$$T = \{(a,b,c,b,c,d,c,d,e,f),(a,b,c,b,c,d,c,d,f,e),$$
$$(a,b,b,c,b,b,b,c,d,e,f),(a,b,b,c,d,b,c,d,d,e,f)\}.$$

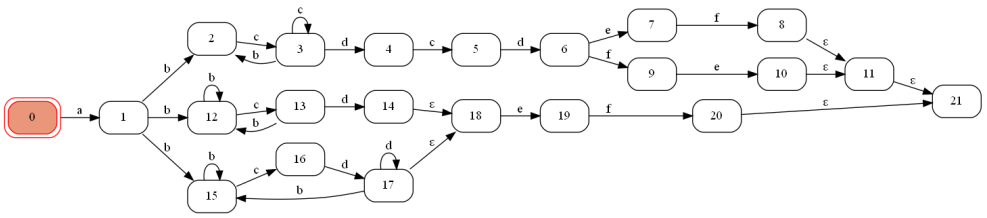

Figure 2. Generated finite automaton example

Note that the first trace has a loop with elements b and c, but no loop with c and d. Therefore, the corresponding pattern after detecting loops is

$(a, LOOP(b, c), d, c, d, e, f)$. The sliding window in the algorithm moves from left to right, meaning that the repeating b and c elements are recognized first. Then, the c and d elements cannot be looping, because the c was used in constructing $LOOP(b, c)$. If the trace had another c element, i.e. if it was $(a, b, c, b, c, d, c, d, c, e, f)$, then another loop would be recognized and the pattern would be $(a, LOOP(b, c), LOOP(d, c), e, f)$. The same is true for the second trace.

Also note that after detecting loops, the algorithm has the patterns $(a, LOOP(b, c), d, c, d, e, f)$, $(a, LOOP(b, c), d, c, d, f, e)$, $(a, LOOP(LOOP(b), c), d, e, f)$ and $(a, LOOP(LOOP(b), c, LOOP(d)), e, f)$. Then, e and f are identified as concurrent elements in the first two patterns, hence they are combined into $(a, LOOP(b, c), d, c, d, AND(e, f))$. In the automaton, this yields two branches recombining with ε-movements at the end (from state 6 to state 11 on the figure). The other two patterns have optional sequences at the beginning (from state 1 to state 18 on the figure), and the sequence e and f are common between the two. Finally, in order to get a single maximal pattern, the patterns we got are combined into one single pattern with XOR, recombining them at the end with ε-movements (from 11 and 20 to 21). The number of states and transitions of the final automaton could be reduced by adding an additional step to the algorithm that eliminates some ε-movements. For example, on Figure 2, states 11 and 21 could be removed, and states 8, 10 and 20 could be combined into a single final state.

Acknowledgement. The authors would like to thank László Kovács for his valuable hints.
The described article was carried out as part of the 2020-1.1.2-PIACI-KFI-2020-00165 "ERPA - Development of Robotic Process Automation solution for heavily overloaded customer services" project implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2020–1.1.2-PIACI KFI funding scheme.

References

[1] Liesaputra, V., Yongchareon, S., and Chaisiri, S.: Efficient process model discovery using maximal pattern mining. In H. R. Motahari-Nezhad, J. Recker, and M. Weidlich (eds.), *Business Process Management*, Springer International Publishing, Cham, ISBN 978-3-319-23063-4, 2015, pp. 441–456, URL https://doi.org/10.1007/978-3-319-23063-4_29.

[2] Oncina, J., Garcia, P., and Vidal, E.: Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis*

and Machine Intelligence, **15**(5), (1993), 448–458, URL https://doi.org/10.1109/34.211465.

[3] van der Aalst, W., Weijters, T., and Maruster, L.: Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, **16**(9), (2004), 1128–1142, URL https://doi.org/10.1109/TKDE.2004.47.

[4] Wen, L., Van Der Aalst, W. M., Wang, J., and Sun, J.: Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, **15**(2), (2007), 145–180, URL https://doi.org/10.1007/s10618-007-0065-y.

[5] Burattin, A. and Sperduti, A.: PLG: A framework for the generation of business process models and their execution logs. In M. zur Muehlen and J. Su (eds.), *Business Process Management Workshops*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-20511-8, 2011, pp. 214–219, URL https://doi.org/10.1007/978-3-642-20511-8_20.

[6] Hoschele, M. and Zeller, A.: Mining input grammars with autogram. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, pp. 31–34, URL https://doi.org/10.1109/ICSE-C.2017.14.

[7] Sutskever, I., Martens, J., and Hinton, G.: Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, Omnipress, Madison, WI, USA, ISBN 9781450306195, 2011, pp. 1017—1024, URL https://doi.org/10.5555/3104482.3104610.

[8] van der Aalst, W.: Process mining: Overview and opportunities. *ACM Transactions on Management Information Systems*, **3**(2), (2012), 1–17, URL https://doi.org/10.1145/2229156.2229157.

[9] Cook, J. E. and Wolf, A. L.: Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, **7**(3), (1998), 215—249, URL https://doi.org/10.1145/287000.287001.

[10] Agrawal, R., Gunopulos, D., and Leymann, F.: Mining process models from workflow logs. In *Advances in Database Technology – EDBT'98*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-69709-1, 1998, pp. 467–483, URL https://doi.org/10.1007/BFb0101003.

[11] Mannila, H. and Meek, C.: Global partial orders from sequential data. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, Association for Computing Machinery, New York, NY, USA, ISBN 1581132336, 2000, pp. 161—168, URL https://doi.org/10.1145/347090.347122.

[12] de Medeiros, A. K. A., van Dongen, B. F., van der Aalst, W. M. P., and Weijters, A. J. M. M.: Process mining: Extending the α-algorithm to mine short loops. In *Eindhoven University of Technology, Eindhoven*, 2004.

[13] Weijters, A., Aalst, van der, W., and Alves De Medeiros, A.: *Process mining with the HeuristicsMiner algorithm*. A publicatie : working papers, Technische Universiteit Eindhoven, 2006, ISBN 978-90-386-0813-6.

[14] Günther, C. W. and van der Aalst, W. M. P.: Fuzzy mining – adaptive process simplification based on multi-perspective metrics. In G. Alonso, P. Dadam, and M. Rosemann (eds.), *Business Process Management*, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-75183-0, 2007, pp. 328–343, URL https://doi.org/10.1007/978-3-540-75183-0_24.

[15] van der Aalst, W. M. P., Rubin, V., Verbeek, H. M. W., van Dongen, B. F., Kindler, E., and Günther, C. W.: Process mining: a two-step approach to balance between underfitting and overfitting. *Software and Systems Modeling*, **9**(1), (2008), 87, URL https://doi.org/10.1007/s10270-008-0106-z.

[16] de Medeiros, A. K. A., Weijters, A. J. M., and van der Aalst, W. M. P.: Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery*, **14**(2), (2007), 245–304, URL https://doi.org/10.1007/s10618-006-0061-7.

[17] Graves, A.: Generating sequences with recurrent neural networks, arXiv, pp. 43. 2014.

[18] Kuo, C. and Chien, J.-T.: Markov recurrent neural networks. In N. Pustelnik, Z.-H. Tan, Z. Ma, and J. Larsen (eds.), *2018 IEEE International Workshop on Machine Learning for Signal Processing, MLSP 2018 - Proceedings*, IEEE International Workshop on Machine Learning for Signal Processing, MLSP, IEEE Computer Society, United States, 2018, pp. 1–6, URL https://doi.org/10.1109/MLSP.2018.8517074.

[19] Hanga, K. M., Kovalchuk, Y., and Gaber, M. M.: A graph-based approach to interpreting recurrent neural networks in process mining. *IEEE Access*, **8**, (2020), 172923–172938, URL https://doi.org/10.1109/ACCESS.2020.3025999.

[20] Welcome to python.org. https://www.python.org/. Accessed: August 1, 2022.

[21] Numpy. https://numpy.org. Accessed: August 1, 2022.

[22] itertools – functions creating iterators for efficient looping. https://docs.python.org/3/library/itertools.html. Accessed: August 1, 2022.

[23] Github – pytransitions/transitions: A lightweight, object-oriented finite state machine implementation in python with many extensions. https://github.com/pytransitions/transitions. Accessed: August 1, 2022.

[24] Graphviz. https://graphviz.org. Accessed: August 1, 2022.