Trace Retrieval as a Tool for Operational Support in Medical Process Management

Alessio Bottrighi (1), Luca Canensi (2), Giorgio Leonardi (1), Stefania Montani (1), Paolo Terenziani (1)

(1) DISIT, Computer Science Institute, Università del Piemonte Orientale, Alessandria, Italy
(2) Department of Computer Science, Università di Torino, Italy

Abstract. Operational support assists users while process instances are being executed, by making predictions about the instance completion, or recommending suitable actions, resources or routing decisions, on the basis of the already completed process traces. Operational support can be particularly useful is the case of medical processes, where a given process instance execution may differ from the indications of the existing reference clinical guideline. In this paper, we propose a Case Based Reasoning approach to medical process management operational support. The framework enables the user to retrieve past traces by submitting queries representing complex patterns exhibited by the current process instance. Information extracted from the retrieved traces can guide the medical expert in managing the current instance in real time. The tool relies on a tree structure, allowing fast retrieval from the available event log. Thanks to its characteristics and methodological solutions, the tool implements operational support tasks in a flexible, efficient and user friendly way.

1 Introduction

Operational support is a process management activity meant to assist users while process instances are being executed, by making predictions about the instance completion, or recommending suitable actions, resources or routing decisions, on the basis of the already completed instances [1]. Operational support can be particularly useful in the case of medical processes, where a given process instance execution may (significantly) differ from the indications of the existing reference clinical guideline. Indeed, specific patient characteristics (e.g., co-morbidities, allergies, etc.), or local resource constraints, may lead to deviations from the default behavior, which need to be managed in real time. Prediction and recommendation heavily rely on experiential knowledge, stored in the so-called "event log" in the form of past process traces. Case Based Reasoning (CBR) [2], and specifically the retrieval step in the CBR cycle, thus appears to be a very valuable methodology for implementing these operational support tasks. The percentage of retrieved traces that, e.g., were completed on time, can then be used to calculate the probability that the current instance will complete on time too. A similar approach can be adopted to estimate costs, or predict problems. Moreover, the best actions to execute next can also be extracted from the retrieved traces.

In this paper we propose a case-based retrieval framework, where cases are traces of process execution, aimed at enabling prediction and recommendation in medical process operational support. In our framework, queries can be composed of several simple patterns (i.e., single actions, or direct sequences of actions), separated by delays (i.e., interleaved actions we do not care about). Delays can also be imprecise (i.e., the number of interleaved actions can be given as a range). To the best of our knowledge, an operational support facility like this is not available in the tools described in the literature. Our framework relies on a tree structure, called the *trace tree*, allowing fast retrieval, thus avoiding a flat search for all the traces in the log that match the input pattern. The trace tree is a sort of "model" of the traces, that we learn using a process mining technique we recently implemented [3], and built in such way that it can be used as an index[1].

The paper is organized as follows. In section 2 we illustrate our retrieval approach. In section 3 we discuss related work. In section 4 we present our concluding remarks and future work directions.

2 Trace retrieval

In our framework, trace retrieval relies on a tree structure, called the **trace tree**, in order to avoid a flat search for all the traces in the log that match the input query. In the following, we will first describe the trace tree semantics, and then introduce our query language and, finally, our retrieval procedure.

Trace tree semantics. In the trace tree, nodes represent actions, and arcs represent a precedence relation between them. More precisely, each node is represented as a pair $< P, T >$.

P denotes a (possibly unary) set of actions; actions in the same node are in AND relation, or, more properly, may occur in any order with respect to each other. Note that, in such a way, each path from the starting node of the tree to a given node N denotes a set of possible process patterns (called *support patterns* of N henceforth), obtained by following the order represented by the arcs in the path to visit the trace tree, and ordering in each possible way the actions in each node (for instance, the path $\{A, B\} \rightarrow \{C\}$ represents the support patterns "ABC" and "BAC").

T represents a set of pointers to all and only those traces in the log whose prefixes exactly match one of the patterns in P (called *support traces* henceforth). Specifically, prefixes correspond to the entire traces if the node at hand is a leaf. In the case of a node representing a set of actions to be executed in any order, T is

[1] While the motivations for defining such a novel mining algorithm, and its advantages with respect to existing process mining literature contributions (e.g., ProM [4]), are extensively discussed elsewhere [3], in this work we focus on its usage to support case retrieval.

more precisely composed of several sets of support traces, each one corresponding to a possible action permutation.

Since all traces start with a dummy common action #, the root node contains the action #, paired to the pointers to all log traces.

Query language. In a tool implementing this framework, the user can issue a query, composed of one or more simple patterns to be searched for. In turn, simple patterns are defined as one or more actions in direct sequence. Multiple simple patterns can be combined in a complex pattern, by separating them by delays. A delay is a sequence of actions interleaved between two simple patterns; the semantics is that we do not care about these actions, so they will not be specified in the query. Instead, only their number will be provided, possibly in an imprecise way (i.e., we allow the user to express the number of interleaved actions as a range).

Formally, a query is represented in the following format:

$$\langle (min_1, max_1) SP_1 (min_2, max_2) SP_2 ... (min_k, max_k) SP_k (min_{k+1}, max_{k+1}) \rangle$$

where:

- SP_j is a simple pattern (i.e. a sequence of letters, representing the actions we are looking for; these actions have to be in direct sequence);
- (min_j, max_j) is the delay between two items (i.e., two simple patterns, or a simple pattern and the trace starting/ending point), expressed as a range in the number of interleaved actions.

As an example, the query
$\langle (0,0) B (0,1) E (2,2) Z (0,1) \rangle$
looks for action B, which has to start at the very beginning of the trace (just after the start action # - all traces start with a dummy common action # in our approach). This first simple pattern B must be followed (with zero or a single interleaved action in between) by action E. E must be followed by two actions, which we do not care about; after them, Z is required. Z can be the final action, or can be followed by one additional action we do not care about.

For instance, in the stroke management domain, where we will test our approach, actions B, E and Z could correspond to “Arrival at the emergency department”, “Neurological examination”, and “Chest X-ray” respectively. Looking for “Arrival at the emergency department” at the very beginning of the trace allows the exclusion of all those patients that were first stabilized at home or in the ambulance. The query then aims for searching for those situations where “Neurological examination” is executed early, and before “Chest X-ray”; in fact, this specific ordering is not mandatory, because “Chest X-ray” is a procedure common to many different disease management processes, and may be executed at different times, also depending on the availability of the X-ray machine. Similarly, in some cases “Neurological examination” might be delayed, if the neurologist is not available. The two actions between “Neurological examination” and “Chest X-ray” would typically correspond to “CTA” and “ECG”, always obtained to every patient in the case of a suspected stroke (but not explicitly queried in the example).

It is worth noting that a query written as above corresponds to a whole set of queries, each one obtained by choosing a specific delay value and specific actions in each of the (min_j, max_j) intervals. Every query in this set can be made partially explicit as a string, containing as many dummy symbols $*$ as needed, to cover the corresponding delay length (where the dummy symbol is chosen because we are not interested in the specific interleaved actions). For example, the query above would correspond to the following four partially explicit queries, whose length ranges from 6 to 8 actions (including #), where the dummy symbol $*$ has been properly inserted, according to the delay values information: $\#BE**Z$; $\#BE**Z*$; $\#B*E**Z$; $\#B*E**Z*$

Since each $*$ could be substituted by any of the N types of actions recorded in the log and/or existing in the application domain, the example query corresponds to $N^2 + 2 * N^3 + N^4$ totally explicit queries.

The problem is obviously combinatorial, with respect to the possible delay ranges and action types. We thus believe that extensional approaches (in which only explicit queries can be issued) would not be feasible in many domains. Our query language, allowing for compact "intensional" queries, is therefore a significant move in the direction of implementing an efficient and user-friendly operational support tool.

Trace retrieval. In order to retrieve the log traces that match a query, we have implemented a multi-step procedure, articulated as follows: (1) automaton generation; (2) tree search; (3) filtering.

To generate the automaton, in turn, we implement the following procedure:

1. transform the query into a regular expression;
2. apply the Berry and Sethi [5] algorithm, to build a non-deterministic automaton that recognizes the regular expression above;
3. unfold the non-deterministic automaton;
4. transform the unfolded non-deterministic automaton into a deterministic automaton [6].

Steps (1) and (4) are trivial. As regards step (1) note that our query language is just a variation of regular expressions, useful to express delays and "do not care" (i.e., dummy) symbols in a compact way. The cost of step (1) is linear in the number of delays used in the query. Steps (2) and (3) use classical algorithms in the area of formal languages. The cost of step (2) is linear in the number of symbols in the query expressed as a regular expression (i.e., the output of step (1) [5]), and the cost of step (3) is the product between the number of dummy symbols in the query and the cardinality of the action symbols available in the log. Step (4) substitutes each arc labeled by the dummy symbol in the automaton with a set of arcs, one for each action in the event log. Although in the worst case step (4) is exponential with respect to the number of states in the automaton (i.e., the output of step (2)), note that the worst case is rare in practice [7].

Once the deterministic automaton has been obtained, it would be possible to exploit it in a classical way, by providing all event log traces in input to it, to verify which of them match the query. However, some of these traces may be identical, or share common prefixes of various length, so that the straightforward approach would lead to repeated analyses of the common parts. In order to optimize efficiency, we have therefore proposed a novel approach, that provides the trace tree as an input to the automaton. Each path in the trace tree may index several identical support traces, that will be considered only once, thus speeding up retrieval with respect to a flat search into the event log. Moreover, in the tree common prefixes of different traces are represented just once, as common branches close to the root (different postfixes can then stem from the common branches, to reach the various leaves). These common parts will be executed on the automaton only once, without requiring repeated, identical checks.

It is worth noting that providing a tree as an input to the automaton represents a significantly novel contribution, since in the formal languages literature the input to be executed on the automaton is typically a string. The work in [8] represents an exception, but the tree it exploits (a Patricia tree) has very different semantics with respect to ours.

In detail, our approach operates as follows: the algorithm *Search_Process* (see algorithm 1) takes in input the trace tree T and the deterministic automaton A, and provides as an output a set of pairs, composed of a trace tree leaf node and a corresponding string. Notably, there could be several pairs having the same leaf node. Each of the strings is an explicit instantiation of the query represented by the automaton, verified by (some of) the support traces in the leaf node. The output support traces are then provided as an input to the filtering step (see below).

Basically, *Search_Process* executes a breadth first visit of the trace tree; it exploits the variable *searching*, defined as a set of triples, composed of a trace tree node, an automaton state, and the string that has been recognized on the automaton so far. Initially (line 4), *searching* contains the root (with the dummy action #), paired to the initial state of the query automaton and to the empty string. The visit procedure (lines 7-35) extracts one triple at a time from the set *searching*. If the *node* in the triple contains a set of actions to be executed in any order (line 9), we simulate all the permutations on the automaton, and save the states we reach and the corresponding recognized strings into *new_states* set (line 12). If the node contains one single action, we simply simulate it on the automaton, and save the state we reach and the corresponding string into *new_states* set (line 17). In both cases, the string saved in *new_states* is the one in the input triple properly updated with the newly recognized symbols.

After the simulation, if the *node* at hand is a leaf (line 20), then for each item in *new_states* we check whether the state component is a final state (lines 22-24); if this is the case, *node* and the string paired to the final state are saved in the output variable *result* (line 23). Otherwise, if *node* is not a leaf, we pair its children to all the items in *new_states*, and save these objects into *searching* (lines 27-33). The visit terminates when *searching* is empty, i.e., all tree levels have been visited. The visit procedure is linear in the number of the trace tree nodes.

Referring to our example query, providing the trace tree in figure 1 as an input to the algorithm *Search_Process*, after examining the root (which is triv-

ALGORITHM 1: Pseudo-code of the procedure *Search_Process*.

```
Search_Process(T, A)
Output: set of $< node, string >$
result ← {}
searching ←$< root(T), 0, empty >$
repeat
    tmp ← {}
    foreach $< node, state, string > \in searching$ do
        new_states ← {}
        if node is an any-order-node then
            foreach $Perm \in$ permutation(node) do
                foreach $act \in Perm$ do
                    new_states ← new_states ∪ simulate(A, act,state,string)
                end
            end
        end
        else
            new_states ← simulate(A, action(node),state,string)
        end
        if $new\_states \neq \{\}$ then
            if node is a leaf then
                foreach $< state, string > \in new\_states$ do
                    if final(state) then
                        result ← result ∪ $< node, string >$
                    end
                end
            end
            else
                foreach $n \in$ sons(node) do
                    foreach $< state, string > \in new\_states$ do
                        tmp ← tmp ∪ $< n, state, string >$
                    end
                end
            end
        end
    end
    searching ← tmp
until $searching \neq \{\}$
return result
```

ial), *searching* contains the children of the root A, B, and C, paired with the state of the deterministic automaton, and with the string $\#$. We simulate the actions A, B, and C on the automaton. Only B (i.e., "Arrival at the emergency department") is recognized, generating a state saved in *new_states* with the corresponding string $\#B$ (line 17). We then pair the children of node B (E, D, $D-E$) to the item in *new_states* and save these triples into *searching* (lines 27-33). In the stroke management domain, E corresponds to "Neurological examination" and D to "CTA". Continuing the visit, particularly interesting is the case of node $D-E$, which requires consideration of all the possible permutations of actions D and E. Both the permutations DE and ED are initially recognized. However, as the visit proceeds and node $P-Z$ is reached (with P corresponding to "ECG" and Z corresponding to "Chest X-ray"), it turns out that DE must be followed by the permutation PZ to match the query; on the other hand, if the choice ED is made, it must be followed by ZP. Indeed, the query imposes some *constraints that cannot be checked only locally*, i.e., referring to a single node along the branch. After this step of the visit (depth 5 in the tree), the recognized partial strings paired to node $P-Z$ are $\#BDEOPZ$ and $\#BEDOZP$ (with O corresponding to "NMR"). Notably, the patterns $\#BDEOZP$ and $\#BEDOPZ$ do not match the input query.

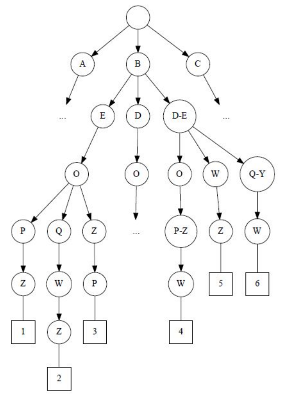

Fig. 1. Trace tree in the example.

If an output leaf node ends a branch which includes one or more nodes with actions to be executed in any order, it is possible that only some of the permutations of these actions are acceptable to answer the query. However, the trace tree leaf node indexes all the traces corresponding to the various support patterns (i.e., considering all possible permutations). Therefore, the support traces must be filtered.

To do so, without the need of operating directly on the input traces, we exploit the fact that, in each node with actions to be executed in any order, every permutation is explicitly stored, and each permutation indexes all and only the support traces corresponding to it. Thus, the basic idea of our filtering step is simple: for each output pair ⟨ Node, String ⟩ of the tree search step, we navigate the trace tree from *Node* back to the root, maintaining, in each any order node, only the (pointers to) the traces corresponding to *String* (this can be easily done through the operation of intersection between sets of pointers).

The complexity of the filtering step is superiorly limited by the number of ⟨ Node, String ⟩ pairs identified as an output of the tree search step, multiplied by the tree depth.

Obviously, if the leaf node ends a branch that contains no nodes with actions to be executed in any order, the leaf support traces can be directly presented to the user, and the filtering step is not required.

3 Related work

Operational support techniques are implemented in the open source framework ProM [4] (developed at the Eindhoven University of Technology), which represents the state of the art in process mining research. In ProM, prediction and recommendation are typically supported by replaying log traces on the transition system [9], a state-based model that explicitly shows the states a process can be in, and all possible transitions between these states. The replay activity allows calculation of, e.g., the mean time to completion from a given state, or the most probable next action to be executed. In ProM's approach, statistics on event log traces are thus used for operational support, but the overall technique is very different from the one we propose in this paper, and no trace retrieval on the basis of complex pattern search is supported.

On the other hand, traces have been recently considered in the CBR literature, as sources for retrieving and reusing user's experience. As an example, at the International Conference on CBR in 2012, a specific workshop was devoted to this topic [10]. In 2013, Cordier et al. [11] proposed trace-based reasoning, a CBR approach where cases are not explicitly stored in a library, but are implicitly recorded as "episodes" within traces. The elaboration step, in which a case is extracted from a trace, is thus one of the most challenging parts of the reasoning process. Zarka et al. [12] extended that work, and defined a similarity measure to compare episodes extracted from traces. In these works, traces are typically intended as observations captured from users' interaction with a computer system. Trace-based reasoning was exploited in recommender systems [13, 14], and to support the annotation of digitalized cultural heritage documents [15]. Leake used execution traces recording provenance information to improve reasoning and explanation in CBR [16]. In the Phala system [17], the authors supported the generation and composition of scientific workflows by mining execution traces for recommendations to aid workflow authors. Finally, Lanz et al. used annotated traces recorded when a human user played video games in order to feed a case-based planner [18].

All these approaches implement forms of reasoning on traces. However, to the best of our knowledge, a tace-based CBR approach has never been exploited for operational support in Medical Process Management.

4 Conclusions

In this paper, we have introduced a novel framework for trace retrieval, designed to implement operational support tasks in a flexible, efficient and user-friendly way. With respect to existing operational support facilities, our tool is more flexible because it allows to search for traces that exhibit complex query patterns, identified in the input trace. The tool is also efficient and user-friendly, since:

- by allowing for the use of (imprecise) delays in the query language, it enables users to express a very large number of explicit queries in a compact way;
- by providing the trace tree as an input to the automaton:
  - it speeds up retrieval relative to a flat search into the event log;
  - it executes common prefixes of different traces only once on the automaton, avoiding repeated, identical checks.

In the future, we plan to extensively test the overall framework on real world traces, which log the actions executed during stroke patient management in a set of Northern Italy hospitals.

References

1. W. Van der Aalst. *Process Mining. Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
2. A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations and systems approaches. *AI Communications*, 7:39–59, 1994.
3. L. Canensi, S. Montani, G. Leonardi, and P. Terenziani. Chapman: a context aware process miner. In *Workshop on Synergies between Case-Based Reasoning and Data Mining, International Conference on Case Based Reasoning (ICCBR)*, 2014.
4. B. van Dongen, A. Alves De Medeiros, H. Verbeek, A. Weijters, and W. Van der Aalst. The proM framework: a new era in process mining tool support. In G. Ciardo and P. Darondeau, editors, *Knowledge Mangement and its Integrative Elements*, pages 444–454. Springer, Berlin, 2005.
5. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theor. Comput. Sci.*, 48(3):117–126, 1986.
6. M. S. Lam, A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.
7. J. van Leeuwen. *Handbook of Theoretical Computer Science.* MIT Press, 1994.
8. Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, November 1996.
9. B. F. Van Dongen, N. Busi, and G. M. Pinna. An iterative algorithm for applying the theory of regions in process mining.

10. M. W. Floyd, B. Fuchs, P. Gonzalez-Calero, D. Leake, S. Ontanon, E. Plaza, and J. Rubin. *TRUE: Traces for Reusing User's Experiences Cases, Episodes, and Stories, International Conference on Case Based Reasoning (ICCBR).* Lyon, 2012.
11. Amlie Cordier, Marie Lefevre, Pierre-Antoine Champin, Olivier Georgeon, and Alain Mille. Trace-Based Reasoning — Modeling interaction traces for reasoning on experiences. In *The 26th International FLAIRS Conference*, May 2013.
12. Raafat Zarka, Amlie Cordier, Elod Egyed-Zsigmond, Luc Lamontagne, and Alain Mille. Similarity Measures to Compare Episodes in Modeled Traces. In Springer, editor, *International Case-Based Reasoning Conference (ICCBR 2013)*, Lecture Notes in Computer Science, pages 358–372. Springer Berlin Heidelberg, July 2013.
13. Gediminas Adomavicius and Alexander Tuzhilin. Context-aware recommender systems. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, pages 335–336, New York, NY, USA, 2008. ACM.
14. Raafat Zarka, Amélie Cordier, Elöd Egyed-Zsigmond, and Alain Mille. Contextual trace-based video recommendations. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 751–754, New York, NY, USA, 2012. ACM.
15. Reim Doumat, Elöd Egyed-Zsigmond, and Jean-Marie Pinon. User trace-based recommendation system for a digital archive. In *Proceedings of the 18th International Conference on Case-Based Reasoning Research and Development*, ICCBR'10, pages 360–374, Berlin, Heidelberg, 2010. Springer-Verlag.
16. David B. Leake. Case-based reasoning tomorrow: Provenance, the web, and cases in the future of intelligent information processing. In *Intelligent Information Processing V - 6th IFIP TC 12 International Conference, IIP 2010, Manchester, UK, October 13-16, 2010. Proceedings*, page 1, 2010.
17. D. B. Leake and J. Kendall-Morwick. Towards case-based support for e-science workflow generation by mining provenance. In K.D. Althoff, R. Bergmann, M. Minor, and A. Hanft, editors, *Proc. ECCBR 2008, Advances in Case-Based Reasoning*, volume 5239 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.
18. A. Lanz, B. Weber, and M. Reichert. Workflow time patterns for process-aware information systems. In *Proc. BMMDS/EMMSAD*, pages 94–107, 2010.