Exploring L* for Process Mining

Karnika Shivhare[1], Rushikesh K. Joshi[1]

[1]Department of Computer Science Engineering, Indian Institute of Technology Bombay, Mumbai, India.

Abstract

When processes execute through their business logic, their activities generate event logs, which contribute to trace sets. Since its introduction, the field of process mining has evolved, however, accuracy issues persist. In this paper, we explore the L* algorithm in the context of process mining, especially towards development of interactive process mining techniques. We discuss the results of our experiments, the limitations of the approach, and possible future directions towards adapting the technique for covering richer set of features.

Keywords

Process, Traces, Trace Sets, Process Mining, Interactiveness, L* Algorithm, Automatons, Grammar Inferencing

1. Introduction

Processes are progressions of activities that execute and generate event logs in the form of trace sets. These event logs present executed traces of the processes, and they are routinely used by Process Mining algorithms to mine processes from them. Process Mining has been explored for more than a decade, but discovering a process accurately from its given trace set is still an attractive research proposition that needs attention. A *Trace* from a process represents a control flow of the process during an execution run, and it can be represented as a *string of activities*. Such a string can be obtained from a *process execution engine* through its logs. Similarly, *automatons* can be considered as execution engines where strings of alphabets are produced by virtue of transitions between states.

The set of strings generated by an automaton represents the language accepted by the automaton, while the process execution engine generates strings of activities representing the trace set of the process. The former is a complete set, whereas, the latter is a set generated by a finite number of executions. The L* algorithm [1] is an algorithm that generates automatons from given regular language represented in terms of set of strings. The algorithm learns a deterministic finite automaton (DFA) from a set of positive and negative examples. It begins with an initial hypothesis DFA that accepts all of the positive examples and none of the negative examples. It then iteratively refines the hypothesis through interaction with an oracle to tailor the alterations to obtain a finite automaton for the given automaton language. It generates minimalistic DFAs [1].

Other related works in this direction of automata learning and grammar extraction include learning of context free languages such as by Omphalos [2], extracting context free languages in extended Backus–Naur form [3] and learning of probabilistic finite state machines such as by PAutomaC [4]. Several other works [5] [6] [7] provide surveys in this field of grammar inferencing [8] with a focus on constructing grammar for the underlying language by learning through examples.

We observe that the field of Process Mining can benefit by placing automaton learning in proximity with Process Discovery. Process models can be learnt like formal grammars by using the idea of positive examples in learning automatons and applying them to event log trace sets.

2. Our Approach

We present the observations and results (Table 1) of our exploration of the L* algorithm to mine processes. In our approach, processes are mined as DFAs with the help of an oracle that validates the generated DFA by providing counterexamples for incorrect DFAs. The algorithm is unaware about the process behavior to be mined, the correctness of its mined DFA, it's iteration stage, or the point of incorrectness of its result. It is an interactive learning (or refining) algorithm that continues to take counterexamples and learn. This brings flexibility to the algorithm as to repeat its learn-and-refine cycle until a correct DFA is constructed, in contrast to other process mining algorithms that generate either a correct or an incorrect process. L* is an iterative approach to extract regular grammar, and we utilize its iterations for extracting processes. We view process traces as strings in L* algorithm, and begin with a hypothesis of an empty transition as the only trace in the trace set of the process to be mined. The hypothesis is then iteratively refined until

International Workshop on Petri Nets and Software Engineering 2023, PNSE'23

karnika@cse.iitb.ac.in (Karnika); rkj@cse.iitb.ac.in (Rushikesh K.)

https://www.cse.iitb.ac.in/~karnika/ (Karnika); https://www.cse.iitb.ac.in/~rkj/ (Rushikesh K.)

0000-0001-6490-0380 (Karnika); 0000-0002-2712-1406 (Rushikesh K.)

Table 1

Microconfigurations	Process Trace Set	Number of DFA Constructions	DFA Developments			Count of Successes from [9]	Correct Mining by L* for Process Mining
			Correct DFA Obtained	Counterexample Provided	Type of Counterexample		
Bowtie [9]	{*acd, bce*}	04	✗	*acd*	Positive	00	✓
			✗	*bce*	Positive		
			✗	*cce*	Negative		
Critical Section [9]	{*abcd, cdab*}	04	✗	*abcd*	Positive	00	✓
			✗	*cdab*	Positive		
			✗	*bdab*	Negative		
Crossover [9]	{*abcd, pqrs, pcd, ars*}	05	✗	*pqrs*	Positive	02	✓
			✗	*pcd*	Positive		
			✗	*bcd*	Negative		
			✗	*abcd*	Positive		
Early Completion [9]	{*acb, abd*}	04	✗	*abd*	Positive	00	✓
			✗	*bce*	Positive		
			✗	*adb*	Negative		
Initial Bypass [9]	{*abcde, de*}	03	✗	*abcde*	Positive	02	✓
			✗	*cbcde*	Positive		
Intertwined Active Bypass [9]	{*abcde, abge, afde*}	03	✗	*abcde*	Positive	04	✓
			✗	*acde*	Negative		
Intertwined Vanilla Bypass [9]	{*abcde, abde, acde*}	03	✗	*abcde*	Positive	01	✓
			✗	*abcdeabcde*	Negative		
Intertwined Long Bypass [9]	{*abcde, abe, ade*}	03	✗	*abcde*	Positive	02	✓
			✗	*abcdeabcde*	Negative		
Seniority [9]	{*a, b, ab*}	02	✗	*bb*	Negative	01	✓
Ordered Subsequences [9]	{*ab, bc, ac*}	04	✗	*ac*	Positive	00	✓
			✗	*bc*	Positive		
			✗	*cc*	Negative		
Asynchronous Service Loop [9]	{*ac, abc, acb abbc, abcb, acbb, abbbc, abbcb, abcbb, acbbb* }	03	✗	*ac*	Positive	00	✓
			✗	*bac*	Negative		
Concurrent Branching [9]	{*abc, acb, cab* }	05	✗	*abc*	Positive	02	✓
			✗	*acb*	Positive		
			✗	*aab*	Negative		
			✗	*cab*	Positive		
Ticketed Service [9]	{*ad, abd, acd, abcd, acbd, abbd, accd, abbcd, abcbd, acbbd, accbd, acbcd, abccd* }	03	✗	*ad*	Positive	00	✓
			✗	*bad*	Negative		
$a^n b^n$ (Later negative counterexample)	{ϵ, *ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb*}	not defined	✗	*aabb*	Positive	-	✗
			✗	*aaabbb*	Positive		
			✗	*aaaabbbb*	Positive		
			✗	*aaaaabbbbb*	Positive		
			✗	*bbbbbb*	Negative		
$a^n b^n$ (Early negative counterexample)	{ϵ, *ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb*}	not defined	✗	*aab*	Negative	-	✗
			✗	*bbb*	Negative		
			✗	*aaabbb*	Positive		
			✗	*aaaabbbb*	Positive		
			✗	*aaaaabbbbb*	Positive		

the intended behavior of the process is accurately built. It visualizes all next possible traces and their subsequences by spanning the *transition space* from the traces and their subsequences in trace set built by the algorithm at that progression stage. The component of Oracle interactivity is included as a record of all the traces of the given trace set. We experiment this adaptation of L* for process mining with Trace language configurations [9] for testing purposes.

3. Experiments and Results

As part of experimentation, we have implemented the L* algorithm [1] for Process Mining. This adaptation of L*

algorithm is then used to mine the microconfigurations presented by Karnika and Joshi in [9]. They presented thirteen inherent patterns among processes, which they call as micro-configurations [9]. Moreover, they identified them as failure points for some process mining algorithms, and presented the test results of five algorithms that are obtained on testing these micro-configurations using pm4py [10] framework [9].

Table 1 presents experimentation results of applying the L* algorithm for Process Mining over 14 microconfigurations using their corresponding *trace sets* included in the table. The table also includes relevant results and details corresponding to various iterations and DFA constructions witnessed during mining by L*, such as counterexamples provided by Oracle, their types (positive or negative), and the count of DFAs that are constructed by L* algorithm before producing the correct result.

The table also sums up the results of [9] to present a count of process mining algorithms that were found in [9] to successfully discover that micro-configuration. However, the table marks a - (hyphen) to indicate a result of micro-configuration that was not presented by them.

We identify that all the microconfigurations, presented as fracture points for most of the mining algorithms, were successfully mined by *L* algorithm for Process Mining*.

The algorithm correctly identifies processes that generate regular trace sets, and it continues its iterations for processes that generate non-regular trace sets. For example, as shown in the last two entries in Table 1, the two runs of the algorithm on a non-regular trace set $\{a^n b^n\}$ produce correct result only upto the current iteration generating net to accept $\{a^{k-1}b^{k-1}\}$ at k^{th} iteration.

The algorithm generates DFAs, which may have multiple occurrences of transition labels, whereas processes tend to have unique transition labels, since they represent activities in processes.

As the algorithm progresses, behavioral dependencies across transitions are learned by the algorithm. However, sometimes, false positives mislead as it appears at an outset that the algorithm has learned some relationship, but subsequent iterations of DFA miss out on required traces.

The last two entries in the Table 1 represent algorithmic results with change in order of type of counterexamples. In the first attempt, all negative counterexamples are tried before giving any positive counterexample. With an expectation to reduce the number of required negative counterexamples, strategies to have positive counterexamples in the beginning are attempted. However, it can be observed in this case, that the negative counterexamples were still required to eliminate unwanted traces.

4. Conclusion

We explored the L* algorithm in the context of process mining, drawing an analogy between trace sets and languages. We observed that several earlier trace patterns which were found to be fracture points in most of the mining algorithms are correctly identified by L*. However, the algorithm requires an oracle in order to iterate with its help to generate correct process model. In future, the oracle can be replaced by an automated machine that validates the generated results by comparing two trace sets. Since the algorithm works for trace sets expressible with regular expressions, as future work, it can be explored to solve those sub-parts of trace sets that are not solvable by it. Further, since the algorithm generates DFAs, the models generated by the algorithm need to be accurately converted into Petri Nets, and removal of duplicity of transition labels is a challenge in conversion.

Acknowledgments We thank Paritosh K. Pandya for suggesting the L* algorithm for our explorations.

References

[1] D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation 75 (1987) 87–106.

[2] A. Clark, Learning deterministic context free grammars: The omphalos competition, Machine Learning 66 (2007) 93–110.

[3] N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions?, Commun. ACM 20 (1977).

[4] S. Verwer, R. Eyraud, C. de la Higuera, Pautomac: a probabilistic automata and hidden markov models learning competition, Machine Learning (2014).

[5] C. Higuera, A bibliographical study of grammatical inference, Pattern Recognition 38 (2005).

[6] W. Daelemans, Colin de la higuera: Grammatical inference: learning automata and grammars, 2010, Machine Translation 24 (2010) 291–293.

[7] A survey of grammatical inference in software engineering, Science of Computer Programming 96 (2014) 444–459.

[8] M. Young-Lai, Grammar Inference, Springer US, Boston, MA, 2009, pp. 1256–1260.

[9] K. Shivhare, R. Joshi, Trace language: Mining microconfigurations from process transition traces, 2022.

[10] A. Berti, S. van Zelst, W. Aalst, Process mining for python (pm4py): Bridging the gap between process- and data science, 2019.