Computing Semicommutation Closures: a Machine Learning Approach

Maxime Bride, Pierre-Cyrille Héam, Isabelle Jacques

Computing Semicommutation Closures: a Machine Learning Approach

Maxime Bride Pierre-Cyrille Héam Isabelle Jacques

LORIA, INRIA, CNRS UMR 7503, Université de Lorraine FEMTO-ST, INRIA, CNRS UMR 6174, Université de Franche-Comté

Abstract

Semicommutation relations are simple rewriting relation on finite words using rules of the form $ab \to ba$. In this paper we present how to use Angluin style machine learning algorithms to compute the image of regular language by the transitive closure of a semicommutation relation.

1 Introduction

Semicommutation relations are simple rewriting relations on finite words using rules of the form $ab \to ba$. Computing the image of a language by the transitive closure by a semicommutation relation is a challenging problem connected to regular-model-checking [6, 7, 10], trace theory issues [28, 21, 22, 17] or language theory [12, 15, 16, 18]. Several works in the literature investigate the problem of pointing out classes of regular languages whose closure under semicommutation are still regular [6, 7, 10, 17, 16, 1].

In this paper we address the general problem of computing the closure of a regular language under a semicommutation relation using a machine learning approach. Indeed, several recent works show that using a machine learning algorithm is frequently an efficient pratical way to compute unkown regular languages, particularly in a software analysis context (test, verification,...); see, for example, [31, 5, 3, 26, 8, 30]. General tool implementing some learning algorithms have been developped [4, 19] for this purpose. Online machine learning algorithms require an oracle (providing counter-examples) to work. The main contribution of this paper is to develop such an oracle for the computation of closures under semicommutation relations and to experiment it on several examples.

The paper is organised as follows: the useful formal background is defined in Section 1.1. Next, Section 1.2 presents the online machine learning approach for computing semicommutations closures. Section 2 is dedicated to the main contributions of the article by presenting the algorithms defining an oracle. Section 3 presents experimental results on several classes of examples. Conclusion and future works are exposed in Section 4.

1.1 Formal Background

The reader is assumed to be familiar with basic language theory notions [29]. In this paper, Σ denotes a finite alphabet and Σ^* the set of finite words over Σ. A *language* is a subset of Σ^*. The cardinal of a finite set X is denoted $|X|$.

A *finite automaton* on Σ is a tuple (Q, Σ, E, I, F), where Q is a finite set of states, Σ is a finite alphabet, $E \subseteq Q \times \Sigma \times Q$ is the set of transitions, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. A finite automaton is *determinisitic* if there is a unique initial state and if for each state p and each letter a there is at most one state q such that $(p, a, q) \in E$. A *successful path* in a finite automaton is a sequence $(p_0, a_1, q_1), \dots, (p_{n-1}, a_n, q_n)$ of transitions such that p_0 is an initial state, q_n is a final state and for every $1 \leq i \leq n-1$, $q_i = p_i$. The word $a_1 \dots a_n$ is called the *label* of the path. A word is *accepted* by a finite automaton if it is the label of a successful path. The set of accepted (or recognized) words of an automaton $\mathcal{A}$ is denoted $L(\mathcal{A})$.

A *letter-to-letter transducer* or simply a *transducer* is a tuple (Q, Σ, E, I, F), where Q, Σ, I, F are defined as for finite automata. The set E is a subset of $Q \times (\Sigma \times \Sigma) \times Q$. Successful paths are defined as for finite automata but their labels are of the form $(a_1, b_1) \dots (a_n, b_n)$, which is also denoted $(a_1 \dots a_n, b_1 \dots b_n)$. Therefore a transducer accepts a subset of $\Sigma^* \times \Sigma^*$, which is a relation on Σ^*. If $\mathcal{T}$ is a transducer and $\mathcal{A}$ a finite automaton, one can construct in polynomial time (by a product) a finite automaton $\mathcal{T}(\mathcal{A})$ accepting the set of words w such that there exists a word $u \in L(\mathcal{A})$ satisfying that (u, w) is accepted by $\mathcal{T}$ [29].

A *semicommuation* relation I is a subset of $\Sigma \times \Sigma$ such that for every a, $(a, a) \notin I$. Each semicommutation relation I induces a relation R_I on Σ^* defined by $(u, v) \in R_I$ iff there exists two words x, y and two letters a, b such that $u = xaby$ and $v = xbay$ and $(a, b) \in I$. The reflexive-transitive closure of any relation R on Σ^* is denoted R^*. For any language L on Σ, any relation R, $R(L)$ denotes the set of words v such that there exists $u \in L$ satisfying $(u, v) \in R$. A semicommutation relation is *antisymetric* if there is no a, b such that $(a, b) \in I$ and $(b, a) \in I$.

1.2 Machine Learning for Computing Closures under Semicommutation

There exists two main kinds of algorithms to learn regular languages: offline algorithms working from sets of positive and negative examples [24, 11]; and online algorithms based on an oracle guessing whether the learned language is correct and providing counter-examples if not [2, 20, 27]. Our work is based on the online approach, using the `libalf` tool [4] for the experiments.

The general working way of the online approach is depicted in Fig. 1 in the context of our problem: the goal is to find a finite automaton $\mathcal{K}$ such that $L(\mathcal{K}) = R_I^*(L(\mathcal{A}))$. It is required to have an *Oracle* that can say if $L(\mathcal{K}) = R_I^*(L(\mathcal{A}))$. If this equality holds, $\mathcal{K}$ is returned and it's finished. If not, the oracle points out a counter-example $u \in L(\mathcal{A}) \setminus L(\mathcal{K}) \cup L(\mathcal{K}) \setminus L(\mathcal{A})$. With this counter-example, the online algorithm produces a new $\mathcal{K}$ such that u is no more

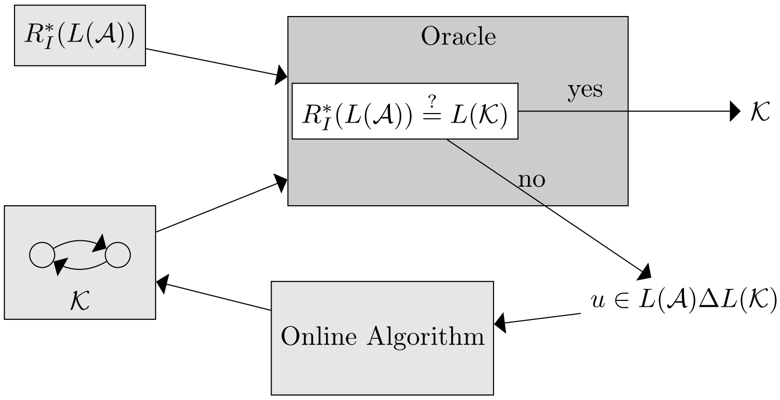


Figure 1: Online Machine Learning

a counter-example for $L(\mathcal{K}) = R_I^*(L(\mathcal{A}))$ (if the equality still doesn't hold). In this paper, online algorithms are used in a blackbox way and we only address the problem of developping an oracle. Testing if $L(\mathcal{K}) = R_I^*(L(\mathcal{A}))$ is done using the following result which is a particular case of a result of [14].

Theorem 1 *Let L, K be languages on Σ and R an antisymetric semicommutation relation. One has $R^*(L) = K$ if and only if $R(K) \cup L = K$.*

Now the remaining questions to build an oracle are:

1. How to check whether $R_I^*(L(\mathcal{A})) = L(\mathcal{K})$, when R_I is not antisymetric?
2. How to efficiently provides $u \in L(\mathcal{A})\Delta L(\mathcal{K})$ if $\mathcal{K}$ is not the good one?
3. How the approach practically works?

Section 2 is dedicated to the presentation of some algorithmst to answer this question. The overall approach with the proposed oracle is experimented in Section 3.

2 Oracle for Learning Semicommutation Closures

2.1 General Scheme for Antisymetric Relations

Algorithm 1 is the oracle algorithm presented in a general way. Several algorithmic issues raised in this description are solved in Sections 2.2 to 2.5. Notice that testing inclusion or equality of regular languages, computing intersection and union of regular languages are done using classical construction of finite automata [29].

Input If $R_I(L(\mathcal{A})) \subseteq L(\mathcal{A})$, then $L(\mathcal{A})$ is R_I-closed and $R_I^*(L(\mathcal{A})) = L(\mathcal{A})$: the problem of computing $R_I^*(L(\mathcal{A}))$ is solved. Therefore one can assume, without loss of generality, that $R_I(L(\mathcal{A})) \not\subseteq L(\mathcal{A})$.

Lines 1-3 For an antisymetric relation I, checking whether the conjecture is correct (i.e. $R_I^*(L(\mathcal{A})) = L(\mathcal{K})$?) is solved using Theorem 1 with $L = L(\mathcal{A})$ and $K = L(\mathcal{K})$. The way to construct a finite automaton recognizing $R_I(L(\mathcal{K}))$ will be described in Section 2.3. Therefore, lines 1-3 of Algorithm 1 check whether the conjecture is correct. In this case *null* is returned.

Algorithm 1 `Oracle Algorithm` (antisymetric relation)

Input: I an antisymetric semicommutation relation, $\mathcal{A}$ and K two finite automata such that $R_I(L(\mathcal{A})) \not\subseteq L(\mathcal{A})$

Output: *null* if $R_I^*(L(\mathcal{A})) = L(\mathcal{K})$, $u \in R_I^*(L(\mathcal{A}))\Delta L(\mathcal{K})$ otherwise.

1: **if** $R_I(L(\mathcal{K})) \cup L(\mathcal{A}) = L(\mathcal{K})$ **then**
2:     **return** *null*
3: **end if**
4: **if** $L(\mathcal{A}) \not\subseteq L(\mathcal{K})$ **then**
5:     **return** $u \in L(\mathcal{A}) \setminus L(\mathcal{K})$
6: **end if**
7: **if** $L(\mathcal{K}) \subseteq L(\mathcal{A})$ **then**
8:     **return** $u \in R_I(L(\mathcal{A})) \setminus L(\mathcal{A})$
9: **end if**
10: **if** $R_I(L(\mathcal{K})) \subseteq L(\mathcal{K})$ **then**
11:     **return** `Search1(`I`,`$\mathcal{A}$`,`$\mathcal{K}$`)`
12: **else**
13:     **return** `Search2(`I`,`$\mathcal{A}$`,`$\mathcal{K}$`)`
14: **end if**

Lines 4-6 If $R_I^*(L(\mathcal{A})) \neq L(\mathcal{K})$, one first checks (line 4) whether $L(\mathcal{A}) \not\subseteq L(\mathcal{K})$. If this condition is satisfied, since $L(\mathcal{A}) \subseteq R_I^*(L(\mathcal{A}))$, any $u \in L(\mathcal{A}) \setminus L(\mathcal{K})$ is a counter-example in $R_I^*(L(\mathcal{A}))\Delta L(\mathcal{K})$. Such a u is obtained using a breadth-first search algorithm working on a finite automaton recognizing $L(\mathcal{A}) \cap L(\mathcal{K})^c$ (note that any search algorithm can be used).

Lines 7-9 Now if $L(\mathcal{K}) \subseteq L(\mathcal{A})$, any $u \in R_I(L(\mathcal{A})) \setminus L(\mathcal{A})$ is in $R_I^*(L(\mathcal{A}))\Delta L(\mathcal{K})$. Since it is assume that $R_I(L(\mathcal{A})) \not\subseteq L(\mathcal{A})$, such a u exists and can be also found by a Breadth-first search algorithm working on a finite automaton recognizing $R_I(L(\mathcal{A})) \cap L(\mathcal{A})^c$ (how to compute $R_I(L(\mathcal{A})$ is described in Section 2.3).

Lines 10-14 To finish, it is tested (line 10) whether $L(\mathcal{K})$ is R_I-closed. Since $L(\mathcal{A}) \subseteq L(\mathcal{K})$ (line 4), if $L(\mathcal{K})$ is R_I-closed, then, by a direct induction, $R_I^*(L(\mathcal{A})) \subseteq L(\mathcal{K})$. The Algorithm 2, called `Search1` and described in Section 2.2, returns $u \in L(\mathcal{K}) \setminus R_I^*(L(\mathcal{A}))$.

Notice that the `Search2` algorithm would be used directly at the first step of the oracle, but since it has an ugly complexity, the described particular cases (lines 5,8,11) are dedicated to simpler cases in order to speed up the procedure.

2.2 Searching Counter-Examples

When $L(\mathcal{K}) \setminus R^*(L) \neq \emptyset$, Algorithm 2 (`Search1`) points out an element u of $L(\mathcal{K}) \setminus R^*(L)$. This algorithm looks for a word of minimal length belonging to $L(\mathcal{K}) \setminus R_I^*(L)$ by a brute force approach. How to test whether $u \notin R_I^*(L)$ is described in Section 2.4. Enumerating the words in $L(\mathcal{K}) \cap \Sigma^n$ can be done by computing a finite automaton recognizing $L(\mathcal{K}) \cap \Sigma^n$. This automaton will be acyclic since it recognizes a finite language. Notice that the automaton recognizing $L(\mathcal{K}) \cap \Sigma^{n+1}$ can be construct from the one recognizing $L(\mathcal{K}) \cap \Sigma^n$, reducing computation times.

Algorithm 2 `Search1`

Input: I an antisymetric semicommutation relation, $\mathcal{A}$ and K two finite automata such that $L(\mathcal{K}) \setminus R_I^*(L)$.

Output: $u \in L(\mathcal{K}) \setminus R_I^*(L(\mathcal{A}))$.

1: n=0
2: **while** true **do**
3:     **for** $u \in L(\mathcal{K}) \cap \Sigma^n$ **do**
4:         **if** $u \notin R_I^*(L)$ **then**
5:             **return** u
6:         **end if**
7:     **end for**
8:     n=n+1
9: **end while**

When $L(\mathcal{K}) \Delta R_I^*(L) \neq \emptyset$, Algorithm 2 (`Search1`) points out an element u of $L(\mathcal{K}) \Delta R_I^*(L)$. This is also a brute force approach looking for a counter-example of the minimal length. Once again, all constructions in the algorithm are classical but the computation of $R_I^*(u)$ (line 9) described in Section 2.4.

2.3 Computing $R_I(L(\mathcal{A}))$

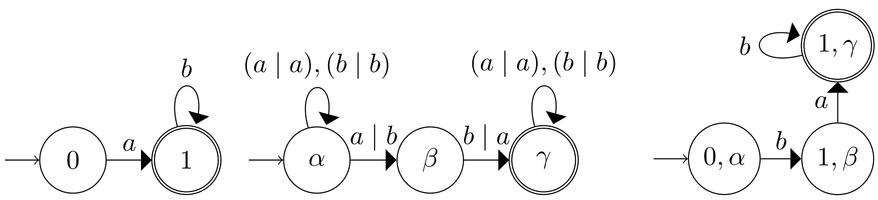
Computing $R_I(L(\mathcal{A}))$ can be easily done using a transducer: R_I is recignized by a transducer. Consider for instance the automaton $\mathcal{A}_1$ depicted on Fig. 2 and the relation R_I associated to $I = \{(a, b)\}$. A transducer recognizing R_I is depicted in Fig. 2. In general case, this transducer has $|I|+2$ states. Computing a product of T_{R_I} and $\mathcal{A}_1$ provides (after trimming) the automaton $T_{R_I}(\mathcal{A}_1)$ accepting $R_I(L(\mathcal{A}_1))$.

Algorithm 3 `Search2`

Input: I an antisymetric semicommutation relation, $\mathcal{A}$ and K two finite automata such that $L(\mathcal{K})\Delta R_I^*(L) \neq \emptyset$.

Output: $u \in L(\mathcal{K})\Delta R_I^*(L(\mathcal{A})) \neq \emptyset$.

n=0
while true **do**
  for $u \in L(\mathcal{K}) \cap \Sigma^n$ **do**
    if $u \notin R_I^*(L)$ **then**
      return u
    end if
  end for
  for $u \in L(\mathcal{A}) \cap \Sigma^n$ **do**
    if $R_I^*(u) \cap L(\mathcal{K})^c \neq \emptyset$ **then**
      return $v \in R_I^*(u) \cap L(\mathcal{K})^c$
    end if
  end for
  n=n+1
end while



(a) $\mathcal{A}_1$ (b) T_{R_I} (c) $T_{R_I}(\mathcal{A}_1)$

Figure 2: Computing $R_I(L(\mathcal{A}))$.

2.4 Testing whether $u \in R_I^*(L(\mathcal{A}))$

Let I be a semicommutation relation and $u \in \Sigma^*$. One has $u \in R_I^*(L(\mathcal{A}))$ iff $(R_I^{-1})^*(u) \cap L(\mathcal{A}) \neq \emptyset$. Since $(R_I^{-1})^*(u) \subseteq \Sigma^{|u|}$, it is a finite set. Therefore $(R_I^{-1})^*(u)$ can be calculated using finitely many time the algorithm of Section 2.3. However, it is more efficient to use a saturation approach to compute a finite automaton accepting $(R_I^{-1})^*(u)$.

2.5 Non antisymetric Relations

For non antisymetric I's, there is no known criterion (as far as we know) to check whether $L(\mathcal{K}) = R_I^*(L(\mathcal{A}))$. In this case, let I_1 and I_2 be two antisymetric semicommutation relations such that $I = I_1 \cup I_2$. Let $K_1 = R_{I_1}^*(L(\mathcal{A}))$, $K_1' = R_{I_2}^*(K_1)$, and for every $n \geq 2$, $K_n = R_{I_1}^*(K_{n-1}')$ and $K_n' = R_{I_2}^*(K_n)$. Next, all the K_i's and K_i''s are computed until reaching a fixed point using Algorithm 1. Notice that it may be a non terminating computation, but if it terminates, the fixed point is $R_I^*(L(\mathcal{A}))$.

2.6 Complexity Issues

Theoretical complexity of Algorithm 1 is exponential due to the brute force approaches of Algorithms 2 and 3. However, in practice we may hope to find quite short counter-examples. Computing intersections and unions of regular languages can be done in polynomial by classical product based constructions. Testing the inclusion (or the equality) of regular languages given by non deterministic automata is PSPACE-complete, but several practically efficient algorithms are known, as [13]. Inclusion and equality are polynomial time decidable for deterministic automata using classical constructions.

3 Experiments

All the algorithms have been implemented in a Java tool and all the tests have been performed on a personal computer Intel Core 2 Duo T7300 2.00GHz with 2 GBytes of memory, running on a Fedora distribution.

3.1 Partially Ordered Automata and Related Languages

All the reported test values of this section were obtained with $|\Sigma| = 5$ and $|I| = 6$ (randomly generated for each test) and by generating 100 examples each time.

A *partially ordered automaton* is a finite automaton in which there is no simple loop of length greater or equal to 2: if $(p_1, a_1, q_1) \dots (p_n, a_n, q_n)$ is a path such that $p_1 = q_n$, then all the p_i's and q_i's are equal. Automata $\mathcal{A}_1$ and $T_{R_I}(\mathcal{A}_1)$ on Fig. 2 are partially ordered. An *alphabetic pattern constraint*, APC for short, is a regular expression which is a finite union of expressions of the form $e_1 e_2 \dots e_k$ where e_k is either a letter or of the form B^* where $B \subseteq \Sigma$. For

(a) Machine Learning (ms)

n	2	3	4	5	6	7
`Gen1`	2 (3+6)	3 (5+15)	5 (8+23)	10 (13+42)	20 (20+67)	57 (27+93)
`Gen2`	5 (3+8)	4 (5+15)	7 (9+29)	15 (13+39)	33 (19+63)	59 (36+133)
`Gen3`	2 (2+6)	2 (3+10)	5 (3+12)	12 (4+17)	108 (5+20)	209 (5+23)
`Gen4`	3 (23+9)	17 (6+20)	591 (8+32)	1688 (15+61)	3859 (18+75)	36459 (32+136)

(b) Specific Algorithm [10] (ms)

n	2	3	4	5	6	7
`Gen1`	0 (5+13)	1 (16+40)	2 (35+84)	6 (86+207)	14 (144+335)	34 (277+681)
`Gen2`	1 (6+18)	3 (20+48)	4 (55+126)	10 (113+255)	28 (253+506)	48 (445+952)
`Gen3`	0 (3+11)	0 (6+32)	1 (10+63)	1 (17+131)	1 (24+211)	2 (36+351)
`Gen4`	0 (4+13)	1 (8+31)	1 (14+61)	2 (23+107)	3 (29+140)	8 (41+204)

Table 1: Results for APC languages

instance $a\{a,b\}^*\{b,c\}^*$ is an APC, but $(ab)^*$ is not. It can be easily checked that a language is accepted by a partially ordered automaton iff it can be expressed by an APC.

For this class of languages the approach was experimented on four random generators. The `Generator1`, randomly (and uniformly) generates a word u of a given length n, build the minimal automaton recognizing $\{u\}$ and randomly add $\frac{3n}{2}$ transitions without introducing any loop. The `Generator2` randomly generates a deterministic partially ordered automaton using a Markov Chain based algorithm closed to the one in [9]. The `Generator3` uniformly generates an APC of the form $B_0^*B_1^* \dots B_n^*$ where the B_i's are subset of the alphabet. Finally, the `Generator4` uniformly generates an APC of the form $B_0^*a_1B_1^* \dots a_nB_n^*$ where the B_i's are subsets of the alphabet and the a_i's are letters.

Table 1 reports the average time (ms) to compute the closure under R_I of the generated languages, both with the machine learning approach and with a specific algorithm [10]. The average size (number of states + number of transitions) of the computed automata (for $R_I^*(L(\mathcal{A}))$ is reported under braces).

Note that the specific algorithm is quite better, what is not surprising. However, the machine learning approach is tractable. It should be emphasized that the specific algorithm produces larger automata. Therefore, if the computed results are used in conjunction with another algorithm (as for a model-checking problem for instance), it may be interesting to have smaller automata and to use the machine learning approach.

Moreover, it can be efficiently tested whether a language (given by its minimal automaton) can be recognized by an APC but there is no known efficient algorithm to build such an expression. Therefore, if such a language is given by its minimal automaton, using the specific algorithm [10] will require a possibly ugly pre-processing step. For instance, the automaton depicted in Fig. 3 can be

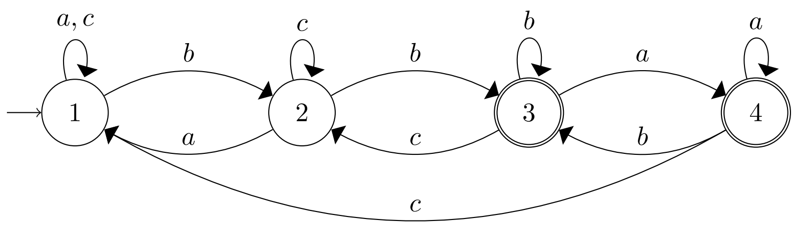


Figure 3: Minimal automaton of an APC language.

n	2	3	4	5	6
`GroupGen`	0.002 (2+6)	0.01 (12+35)	0.04 (40+120)	29.0 (67.3+202)	247.2 (143+430)

Table 2: Results for group languages (seconds), $|\Sigma| = 3$, $|I| = 6$

represented by the APC (see [7] for the complexity of the test):

$$(\{a,b,c\}^*a\{a,c\}^* \cup \{a,c\}^*)\, b\{b,c\}^*b\{a,b\}^*.$$

The Machine Learning Algorithm finds its R_I closure (with $I = \{(a,b),(b,c)\}$) in few milliseconds; finding the above expression from the automaton is not an easy problem since the test is not constructive.

3.2 Languages of PolG and PolC

A regular language is a *group language* if there exists a finite automaton accepting it and for which each letter induces a one-to-one function from the set of states into itself. Equivalently, it is a language accepted by a complete deterministic automaton such that there is no pair of transitions labelled by the same letter pointing the same state. One can prove that a language is a group language iff its minimal automaton has this property. Under some simple conditions on I, if L is a group language, then $R_I^*(L)$ is regular [1]. Notice that the proof of this theoretical result is constructive but lies on Ramsey like results: transforming it into an algorithm is possible but the complexity would be intractable.

We have generated group languages accepted by a n-state automaton in the following way: (1) generate uniformly for each letter of Σ a permutation of $\{1,\ldots,n\}$; 1 is the initial state and each state is final with a probability 1/2 (the reader interested by the random generation of group languages is referred to [23]). The results are reported in Table 2.

PolG is the class of regular languages that are a finite union of languages of the form $L_o a_1 L_1 \ldots a_k L_k$ (*), where the L_i's are group languages and the a_i's are letters. It is know that if L is in PolG, then, under certain hypothesis on I, $R_I^*(L)$ is regular [1]. We have implemented two generators of elements of PolG: first, for a given k, we generate an expression of the form (*), where the a_i's

k (length of the expressions (*))	2 (4 states)	3 (6 states)	4 (8 states)
`Generator6` First Approach	0.25 (9+27)	6.6 (23+69)	18.9 (41+122)
`Generator6` Second Approach	0.001 (8+29)	0.001 (24+92)	0.003 (43+176)

k (length of the expressions (*))	2 (6 states)	3 (9 states)	4 (12 states)
`Generator6b` First Approach	13.8 (54+161)	26.0 (93+279)	27.1 (548+1643)
`Generator6b` Second Approach	0.22 (148+459)	0.35 (1272+4292)	1.1 (13881+50572)

n (states)	2	3	4	5	6	7
`Generator7`	0.06 (2+10)	0.5 (4+20)	0.5 (9+44)	0.2 (8+38)	1.0 (14.70)	14.1 (25+124)

Table 3: Results for PolG (seconds), $|\Sigma| = 3$, $|I| = 6$

are arbitrarily chosen and the L_i's are generated with the `GroupGen` algorithm described above (with $n = 2$; the generated automaton has $2k$ states). This generator is called `Generator6`. `Generator6b` is similar except that $n = 3$; the generated automaton has $3k$ states. The `Generator7` works as follows: (1) an automaton with n states is generated using `GroupGen`; (2) $\sqrt{n}$ transitions are uniformly removed, providing a deterministic automaton recognizing a language L_0; (3) using classical automata constructions, an automaton recognizing L_0^c is returned. Results of [25] ensure that this language is in PolG, even if there is no known tractable algorithm to compute a related expression of the form (*).

For `Generator6` two approaches have been experimented: first $R_I^*(L)$ is computed by the proposed machine learning technique. Secondly, each $R_I^*(L_i)$ is computed using the machine learning algorithm. Next $R_I^*(L)$ is computed using the R-shuffle algorithm [10]. The results are reported in Table 3. For `Generator7` only the first approach can be applied.

The results show that it is possible to compute the closure under semicommutation of group languages or of languages of PolG when finite automata have few states.

A commutative language is a language closed under all semicommutation relations. The class PolC is the class of regular languages which are a finite union of languages of the form $L_o a_1 L_1 \dots a_k L_k$ (**), where the L_i's are commutative regular languages and the a_i's are letters. Regular commutative languages can be generated by generating a n-state automaton having the diamond property: for any pair of states p, q and any pair of letters a, b, if (p, a, q) and (q, b, r) are transitions, then there exists a state s such that (p, b, s) and (q, a, s) are transitions. Using this generator of commutative languages, the `Generator8` produces expressions of the form (**) in the same way as `Generator6`, with $n = 3$. In Table 4, the results of the proposed approach are compared to the results obtained by the dedicated algorithm [10].

k (length of the expressions (**))	2	3	4
`Generator8` Machine Learning Approach	5.8 (14+51)	14.2 (35+133)	26.3 (34+113)
`Generator8` Specific Approach [10]	0.002 (19+69)	0.003 (63+239)	0.01 (169+712)

Table 4: Results for PolC (seconds), $|\Sigma| = 5$, $|I| = 6$

Like APC, the results show that the specific approach runs faster than the machine learning approach. However the latter is tractable and produces quite smaller automata.

4 Conclusion

In this paper we proposed an algorithm to use online machine learning algorithm to compute the image of a regular language by the transitive closure of a semi-commutation relation. Practical experiments show that this approach is slower than specific algorithms for the APC and PolC class of languages. However, for this two classes the computed automata are smaller (and deterministic) making the approach fruitful for combining it with others model-checking techniques. Moreover, it was possible to compute several closures of regular languages for which there is no known efficient algorithm. In the future we plan to develop specific machine learning algorithm dedicated to the computation of semicommutation closures and to improve the efficiency of involved procedures.

References

[1] G. Guaiana A. Cano and J.-E. Pin. Regular languages and partial commutations. *Information and Computation*, 230:76–96, 2013.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering*, 36(3):390–408, May-June 2010.

[4] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 360–364. Springer, 2010.

[5] Benedikt Bollig and Dietrich Kuske. Muller message-passing automata and logics. In Zoltán Ésik, Carlos Martín-Vide, and Victor Mitrana, editors, *Preliminary Proceedings of the 1st International Conference on Language and Automata Theory and Applications (LATA'07)*, Tarragona, Spain, March-April 2007.

[6] Ahmed Bouajjani, Anca Muscholl, and Tayssir Touili. Permutation rewriting and algorithmic verification. In *LICS*, 2001.

[7] Ahmed Bouajjani, Anca Muscholl, and Tayssir Touili. Permutation rewriting and algorithmic verification. *Inf. Comput.*, 205(2):199–224, 2007.

[8] Pierre-Christophe Bué, Frédéric Dadeau, and Pierre-Cyrille Héam. Model-based testing using symbolic animation and machine learning. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Workshops Proceedings*, pages 355–360. IEEE Computer Society, 2010.

[9] Vincent Carnino and Sven De Felice. Sampling different kinds of acyclic automata using markov chains. *Theor. Comput. Sci.*, 450:31–42, 2012.

[10] Gérard Cécé, Pierre-Cyrille Héam, and Yann Mainier. Efficiency of automata in semi-commutation verification techniques. *ITA*, 42(2):197–215, 2008.

[11] François Denis, Aurélien Lemay, and Alain Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004.

[12] Volker Diekert and Yves Metivier. *Handbook of Formal Languages*, volume 3, chapter Partial Commutation and Traces. Springer-Verlag, 1997.

[13] Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2010.

[14] Laurent Fribourg and Hans Olsén. Reachability sets of parameterized rings as regular languages. *Electr. Notes Theor. Comput. Sci.*, 9:40, 1997.

[15] Seymour Ginsburg and Edwin Spanier. Semigroups, presburger formulas, and languages. *Pac. J. Math.*, 16(2):285–296, 1966.

[16] Antonio Cano Gómez, Giovanna Guaiana, and Jean-Eric Pin. When does partial commutative closure preserve regularity? In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 209–220. Springer, 2008.

[17] Giovanna Guaiana, Antonio Restivo, and Sergio Salemi. On the trace product and some families of languages closed under partial commutations. *Journal of Automata, Languages and Combinatorics*, 9(1):61–79, 2004.

[18] Pierre-Cyrille Héam. A note on partially ordered tree automata. *Inf. Process. Lett.*, 108(4):242–246, 2008.

[19] Falk Howar, Malte Isberner, Maik Merten, and Bernhard Steffen. Learnlib tutorial: From finite automata to register interface programs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 587–590. Springer, 2012.

[20] Micahel J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[21] Yves Métivier, Gwénaël Richomme, and Pierre-André Wacrenier. Computing the closure of sets of words under partial commutations. In Zoltán Fülöp and Ferenc Gécseg, editors, *ICALP*, volume 944 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1995.

[22] Anca Muscholl and Holger Petersen. A note on the commutative closure of star-free languages. *Inf. Process. Lett.*, 57(2):71–74, 1996.

[23] Cyril Nicaud. *Etude du comportement en moyenne des automates finis et des langages rationnels.* PhD thesis, UniversitÃ© Paris 7, 2000.

[24] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[25] Jean-Eric Pin. Polynomial closure of group languages and open sets of the hall topology. *Theor. Comput. Sci.*, 169(2):185–200, 1996.

[26] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. *STTT*, 11(4):307–324, 2009.

[27] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences (extended abstract). In David S. Johnson, editor, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 411–420. ACM, 1989.

[28] Jacques Sakarovitch. The "last" decision problem for rational trace languages. In Imre Simon, editor, *LATIN*, volume 583 of *Lecture Notes in Computer Science*, pages 460–473. Springer, 1992.

[29] Jacques Sakarovitch. *Elements of Automata Theory.* Cambridge University Press, 2013.

[30] Muzammil Shahbaz and Roland Groz. Analysis and testing of black-box component-based systems by inferring partial models. *Softw. Test., Verif. Reliab.*, 24(4):253–288, 2014.

[31] Abhay Vardhan and Mahesh Viswanathan. Learning to verify branching time properties. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 325–328. ACM, 2005.