# Active Learning on Visibly One Counter Systems

**Aymeric Fages**
(supervisor: Adrien Pommellet)

Technical Report $n^o$output, July 2021
revision bbac2f2b

Abstract: This report focuses on an active learning algorithm for the class of visibly one-counter automata and its implementation in Python and C++. The algorithm allows creating such automata using queries to a teacher, with no knowledge on the final automaton. We then discuss the possible improvements regarding the algorithm execution speed and the automaton representation. We also give an overview of how benchmarks could be perform on such a semi-automatic program, and how the algorithm could be applied to the class of more general one-counter automata.

Abstract: Ce rapport se concentre sur l'explication d'un algorithme d'apprentissage actif sur la classe de automates visiblement à un compteur et les automates à un compteur, et son implémentation en Python et C++. L'algorithme en lui même permet de créer ce type d'automates en disposant seulement d'un professeur, sans connaître l'automate d'arrivée. Nous discutons ensuite des améliorations possibles de l'algorithme concernant sa vitesse d'execution et la représentation d'un automate. Nous donnons aussi une idée de la manière dont un tel programme semi-automatique peut être évalué par un benchmark, et comment l'algorithme pourrait être appliqué à la classe des automates à un compteur plus généraux.

**Keywords**
Model Checking, Active Learning, Automaton, Automata, V1CA, R1CA


Laboratoire de Recherche et Développement de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
magentaafages@lrde.epita.fr – magentahttp://www.lrde.epita.fr/

# Copying this document

# Contents


**1 Introduction** **4**

**2 Definitions and Notations** **6**

**3 The core algorithm** **9**
3.1 Active learning principles . . . . . 9
3.2 Learning a finite state automata . . . . . 9
3.3 Learning a V1CA . . . . . 10
3.4 Learning a R1CA . . . . . 11
3.5 Personal work and improvements . . . . . 12
3.5.1 A new V1CA representation . . . . . 12
3.5.2 A faster algorithm with useless state removal . . . . . 13
3.5.3 A faster isomorphism search . . . . . 14
3.6 Benchmarking . . . . . 14
3.6.1 Automation of the equivalence query . . . . . 15
3.6.2 Automation of the partial equivalence query . . . . . 16

**4 Implementation and results** **17**
4.1 Implementation details . . . . . 17

**5 Conclusion** **19**

**6 Bibliography** **20**

# Introduction

Nowadays, it is very common for a programmer to code a program along with tests, and verify whether the program works by executing these tests. They ensure that the program works as expected. However, programs can often take a very large number of possible inputs, and testing all those inputs is impossible. Moreover, tests can sometimes be complex, and an error can be made while writing tests. Programmers have to settle for the knowledge that the programs works on some specific test cases, and can never be sure of the perfect reliability of their tests or their program. This issue becomes even more important when programming very critical systems, such as aircrafts, nuclear power plant controllers, etc...

In fact, researchers in model checking came out with an interesting way of creating a program: program synthesis. Letting a human create a program unavoidably leaves room for that human to make mistakes in the program code. Instead, program synthesis stems from the idea that a program should come from a computer, from another program, and that the programmer should only give the specifications of the program he wants to make. Such specifications usually have to be formalized in a specific unambiguous language, such as Linear Temporal Logic (LTL), introduced in 1977 by Amir Pnueli in [Pnueli, 1977]. It is this need of creating rigorous programs without tests that lead to this report.

Note that model checking can help with testing as well. Instead of writing tests, it allows verifying whether a given program verify specifications, which are written in a similar unambiguous language. Note that such an analysis is a *static analysis* of the program, meaning that there is no need to execute the program. Instead, model checking uses models made out from the source code of the program. This is particularly handy when it comes to verifying properties on potential malware programs for example.

Program synthesis allows programmers to create a program only from specifications, and model checking allows verifying rigorous properties on a program. But specification languages such as LTL are non-trivial to use. We can easily imagine that some programmers would rather use a programming language they know than learning new specific formal languages in order to use program synthesis. Hence the need to introduce a new way to create a program with something else than a complex specification language. Let us consider an expert of a domain, that knows exactly how a program should behave. That expert would not necesseraly be capable of writing precise specifications using a language he does not know.

In [Angluin, 1987], Dana Angluin showed a way to obtain a regular automaton by only asking two different kinds of questions to the user: questions of equivalence and questions of membership. Using Angluin's algorithm, one could get an algorithm (more specifically a deterministic finite state automaton) recognizing words from a regular language by only answering questions. In fact, where our expert in a given domain would not be able to write down specifi-

cations, he is surely capable of saying whether an execution of the program is right or not, and whether a program does what he wants it to do. This is active learning: instead of asking an expert to write down specifications, we ask him to answer questions, **queries**, about the program, and we generate **a model** of the program from the answers.

The goal of this report is to develop a learning algorithm for the class of *visibly one-counter automata* as the program model, which forms a subclass of classic *one-counter automata*. These *one-counter automata* can manipulate a counter, by incrementing it or decrementing it, and can tell at any time if the counter value is zero. Using this counter, they can recognize languages more complex than regular languages recognized by *finite state automata*, which is an interesting addition to the work of [Angluin, 1987]. *Visibly one-counter automata* also form a subclass of *visibly pushdown automata*. They are both closed under intersection [Alur, 2004], and proved their utility in XML parsing for instance [Kumar and Viswanathan, 2007]. Where *visibly pushdown automata* are more generic and can handle more complex languages, *visibly one-counter* automata have the advantage of being way more convenient for active learning, and the theory behind them is easier to apprehend. We will also use this work as an opportunity to try to see if this learning technique works for the class of *one-counter automata* languages where the counter change of the automaton does not rely on the symbol read by the automaton.

This report heavily relies on the work of Daniel Neider and Christof Loding [Neider and Loding, 2010], which introduces an active learning algorithm on *visibly one-counter automata* that runs in *polynomial time*. This report focuses on the implementation of the algorithm, and the improvements that can be made.

# Definitions and Notations

The following definitions are equivalent to those in [Neider and Loding, 2010].

An *alphabet* $\Sigma$ is a finite set of *symbols*. A *word* is a finite sequence of symbols $w = a_1 \dots a_n, a_i \in \Sigma$, $i = 1 \dots n$. The empty word is denoted $\varepsilon$. The concatenation of two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_n$ is the word $u \cdot v = uv = a_1 \dots a_n b_1 \dots b_n$. A word $u$ is a prefix of $w$ if there is a word $v$ with $w = uv$. The set of all prefixes of $w$ is denoted $\mathrm{pref}(w)$.
The set of all finite words over $\Sigma$ is called $\Sigma^*$. Any subset $L \subset \Sigma^*$ is called a *language*.

A *pushdown alphabet* is a tuple $\widetilde{\Sigma} = (\Sigma_c, \Sigma_r, \Sigma_{int})$ of three disjoint alphabet. $\Sigma_c$ is the set of *calls*, $\Sigma_r$ the set of *returns* and $\Sigma_{int}$ the set of *internal actions*. For any pushdown alphabet $\widetilde{\Sigma}$, let $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$.

**Visibly One-Counter Automata 2.0.1.** A Visibly One-Counter Automata (V1CA) with threshold $m$, i.e. a $m$-V1CA, over a pushdown alphabet $\widetilde{\Sigma}$ is a tuple $\mathscr{A} = (Q, \Sigma, q_0, \delta_0, ..., \delta_m, F)$ where $Q$ is a finite, non-empty set of states, $\Sigma$ is the input alphabet induced by $\widetilde{\Sigma}$, $q_0 \in Q$ is a initial state, $\delta_i$: $Q \times \Sigma \to Q$ is a *transition function* for every $i = 0, ..., m$, and $F \subset Q$ is the set of final states.

The intuition is that a visibly one-counter automaton has to increment its counter on reading a call and decrement its counter on reading a return from $\widetilde{\Sigma}$. An internal action leaves the counter unchanged. With this intuition we can define the sign $\chi(a)$, $a \in \Sigma$, where:

1. if $a \in \Sigma_c$, $\quad \chi(a) = 1$.
2. if $a \in \Sigma_r$, $\quad \chi(a) = -1$.
3. if $a \in \Sigma_{int}$, $\quad \chi(a) = 0$.

A *configuration* of the V1CA $\mathscr{A}$ is a pair ($q$, $k$) where $q \in Q$ is a state and $k \in \mathbb{N}$ is a counter value.
There is a *a-transition* from ($q$, $k$) to ($q'$, $k'$), denoted by (q,k) $\xrightarrow{a}_{\mathscr{A}}$ (q',k'), if $k' = k + \chi(a) \geq 0$ and $\delta_k(q, a) = q'$ if $k < m$ and $\delta_m(q, a) = q'$ if $k \geq m$. Note that a return cannot be processed if the counter is 0.
A run of $\mathscr{A}$ on a word $w = a_1 ... a_n \in \Sigma^*$ is a sequence $(q_0, k_0), ..., (q_n, k_n)$ of configurations such that $(q_{i-1}, k_{i-1}) \xrightarrow{a_i}_{\mathscr{A}} (q_i, k_i)$ holds for $i = 1, ..., n$. We write $(q_0, k_0) \xrightarrow{w}_{\mathscr{A}} (q_n, k_n)$. A word $w$ is accepted by $\mathscr{A}$ if there is a run $(q_0, 0) \xrightarrow{w}_{\mathscr{A}} (q', 0)$ where $q_0$ is the initial state and $q' \in F$. $L(\mathscr{A})$ is the set of all words accepted by $\mathscr{A}$.

During a run of a V1CA on a word $w \in \Sigma^*$ the V1CA has no control over its counter. The counter value is only determined by the prefix $u$ of $w$ that has been read so far. Hence the counter value of a word $w = a_0...a_n \in \widetilde{\Sigma}$ being defined as follows: $\mathbf{cv}(w) = \sum_{i=0}^{n} \chi(a_i)$ and $\mathbf{cv}(\varepsilon) = 0$.

Moreover, a word $w$ can only be accepted by a V1CA if $\forall u \in \mathrm{pref}(w), \mathbf{cv}(u) \geq 0$ and $\mathbf{cv}(w) = 0$. This means that for any prefix $u$ of the word $w$ read so far, the counter value of $u$ must always be positive for $w$ to be possibly accepted by the V1CA. We thus call:

- $\Sigma^*_{\geq 0}$, the set of all words $\{w \in \Sigma^* | \forall u \in \mathrm{pref}(w), \mathbf{cv}(u) \geq 0\}$. It is the set of all words processable by the V1CA, i.e the set of words that can be possibly accepted by a V1CA.
- $\Sigma^*_{0,t}, t \geq 0$, the set of all words $\{w \in \Sigma^* | \forall u \in \mathrm{pref}(w), 0 \leq \mathbf{cv}(u) \leq t\}$. It is the set of all words processable by V1CAs whose heights (the counter value) do not exceed a fixed value t.

Note that the behaviour of a V1CA under a certain value of its counter is distinct, depending on the value of the counter. But above a certain value, the V1CA always act the same regardless its counter value. The intuition is that a V1CA has a finite number of states and transitions, but an arbitrary large possible value for its counter.

**The refined Nerode congruence 2.0.2.** Let $L$ be a language, the refined Nerode congruence applied to words in $\widetilde{\Sigma}$, noted $\sim_L \subseteq \widetilde{\Sigma}^*_{\geq 0} \times \widetilde{\Sigma}^*_{\geq 0}$ is defined as follows: Two words $u, v \in \widetilde{\Sigma}^*_{\geq 0}$ are $L$-equivalent, if and only if $\mathbf{cv}(u) = \mathbf{cv}(w)$ and $\forall w \in \Sigma^*, uw \in L \iff vw \in L$. We note $u \sim_L v$. The intuition is that, for an automaton $\mathscr{A}$, if $u \sim_{L(\mathscr{A})} v$, then it does not matter whether $\mathscr{A}$ read $u$ or $v$ so far, its future behaviour will be the same either way. The *equivalence class* of a word $w$ is defined as $[\![w]\!]_L = \{u \in \widetilde{\Sigma}^*_{\geq 0} \mid u \sim_L w\}$.

**Behaviour Graph 2.0.3.** For a V1CA language $L$, the behaviour graph $\mathscr{B}(L)$ of $L$ is induced by the refined Nerode equivalence. Indeed, $\forall w \in L$, $[\![w]\!]_L$ is a state of $\mathscr{B}(L)$. Unlike the configuration graph, which represent every possible state of a V1CA, the behaviour graph only has one state for all configurations that are equivalent according to the Nerode congruence. It is infinite, but has a periodic structure. Constructing such a graph of a language will be one part of the goal of our algorithm.

Since the behaviour graph is induced by the refined Nerode congruence, and the latter distinguishes words with different counter values, states of the behaviour graph are distinguished by the counter values of the word accessing them. We also call this value the *levels* of the states of the behaviour graph, which are defined as follow: given a behaviour graph $\mathscr{B}$, and one if its states $q$, characterized by the equivalence class $[\![w]\!]_L$ of a word $w \in \Sigma^*$, $\mathbf{level}(q) = \mathbf{cv}(w)$.

**Realtime One-Counter Automaton 2.0.4.** A realtime one-counter automaton (R1CA) is a tuple $\mathscr{A} = (Q, \Sigma, q_0, F, \delta)$ where $Q$ is a finite set of states, $\Sigma$ is an alphabet, $q_0 \in Q$ an initial state, $F \subseteq Q$ a set of final states, and $\delta$ a transition function $\delta : Q \times \Sigma \to Q \times \{-1, 0, 1\}$. The second component of the function output is the counter operation to apply when using a transition.

Note that, in contrary of a V1CA, $\Sigma$ here is not necessarily a pushdown alphabet. Moreover, the counter operation does not rely on the symbol read by the automaton, but rather on the transition used, i.e. the structure of the automaton itself. Because of that, R1CAs are more expressive than V1CAs, and we can write $L(\texttt{V1CA}) \subset L(\texttt{R1CA})$.

We define a configuration of $\mathscr{A}$ and a run of $\mathscr{A}$ the same way we would define it for a V1CA. For a word $w = a_0...a_n \in \Sigma^*$, $\mathbf{cv}(w)$ is defined as the sum of all counter operation if $w$ was processed by $\mathscr{A}$: the counter value we would get after processing each $\{a_i \mid i \in [0, n]\}$.

We still define $\Sigma^*_{\geq 0} = \{w \in \Sigma^* | \forall u \in \mathrm{pref}(w), \mathbf{cv}(u) \geq 0\}$ the set of all words processable by $\mathscr{A}$, as just like V1CAs, the counter value of a R1CA can never go below 0 for a word to be accepted. $\Sigma^*_{0,t}, t \geq 0$ still denotes the set of all words processable by $\mathscr{A}$ whose heights (the counter value) do not exceed a fixed value t as well.

The behaviour graph of a R1CA is, just like for V1CA, induced by the refined Nerode equivalence. It is also **infinite** and **periodic**.

# The core algorithm

## 3.1 Active learning principles

Our main goal is to guess a model of a language, i.e the V1CA of a V1CA language, from questions and queries. From this issue, we can define the *active learning principle*. We start from nothing else than a black box, called *the teacher*, introduced in [Angluin, 1987], which can only answer questions. Given a V1CA language $L$, theses queries are:

1. The *membership* query: given a word $w \in \Sigma^*$, does $w \in L$?
2. The *equivalence* query: given a V1CA $\mathscr{A}$, does $L = L(\mathscr{A})$? If not, the teacher gives a *counter example*, i.e a word $w \in \Sigma^*$ such that ($w \in L(\mathscr{A})$ and $w \notin L$) or ($w \notin L(\mathscr{A})$ and $w \in L$).
3. The *partial equivalence* query: for any behaviour graph bounded to a level $m$, is the behaviour graph, below this level, complete enough to recognize the language? Formally, given a behaviour graph $\mathscr{A}$ and an integer $t \in \mathbb{N}$, $\forall w \in \Sigma^*_{0,t}$, does $w \in L \iff w \in L(\mathscr{A})$ hold? Like the equivalence queries, if the latter condition is not true for all $w$, then the teacher gives a counter-example that does not satisfy the condition. This query was added in [Neider and Loding, 2010] as a third query to be able to handle cases where the algorithm would be unable to learn a V1CA due to an incomplete behaviour graph below a certain level, with the classic equivalence query only giving counter-example increasing the bound $k$ of the behaviour graph but not giving information about lower levels.

Note that the teacher is always able to answer those queries above. The idea is that the teacher is an expert about the V1CA language he wants to obtain, but he is only able to answer questions, because it is way easier than directly trying to describe a V1CA.

## 3.2 Learning a finite state automata

In [Angluin, 1987], Angluin shows a way to learn a finite state automaton only using membership and equivalence queries from a teacher. It relies on the idea that a state of an automaton can be characterized by the words we can use to reach this state. It thus defines, for a regular language $L$, a set $S$ of prefixes, which represent the set of states of the finite state automaton the algorithm tries to find, characterized by their prefixes. But it is sometimes possible to access a single state using different words: each state of the automaton can potentially be characterized by several prefixes.

To distinguish prefixes, it introduces a set $E$ of suffixes of words. Given a suffix $v \in E$, $v$ will distinguish prefixes of $S$ by saying, for a prefix $u \in S$, whether $u \cdot v$ is accepted by the language:

$E$ is an approximation of the Myhill-Nerode congruence. By applying every suffix to every prefix, we form a table of accepted/not accepted (boolean) values, where rows are labeled by the prefixes of $S$ and columns are labeled by suffixes of $E$. To fill the table, the algorithm uses the membership query of the teacher on each permutation of prefix/suffix.

Given an alphabet $\Sigma$, the algorithm starts with $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$. To tell whether the table is complete enough to find the result automaton, we try to find if the table is *closed* and *consistent*. The table is closed if we cannot find a a new prefix to add to $S$ which has a different behaviour regarding the suffixes of $E$, i.e a different row in the table than every other prefixes. The idea is that if the table is closed, then we do not need to explore for new rows (i.e new states) in the table. It is consistent if we cannot find a new suffix to add to $E$ that can distinguish two prefixes of $S$, i.e a suffix that would make two prefixes with the same row have a different row. If the table is not closed or consistent, it is filled by taking elements of $S$ and $E$, and by concatenating elements of $\Sigma$ to them as a new prefix or suffix.

Once the table is closed and consistent, we can construct the resulting finite state automaton. The set of rows of the table forms the set of states of the automaton. Duplicated rows are removed from the table as they denote a state already in present in the table. The automaton formed by the table is complete, so the transitions are deduced by linking, for every row and its prefix $p \in S$, for every symbol $a \in \Sigma$, the state of $p$ to the state corresponding to $p \cdot a$ in the table.

Now that the automaton is formed, the latter does not necessarily represent the language wanted by the teacher. Using the equivalence query, it asks the teacher if the automaton is correct. If not, the counter-example given by the teacher and its prefixes are added to $S$, and its suffixes are added to $E$. Then the algorithm restarts by trying to make the table closed and consistent again.

## 3.3 Learning a V1CA

Angluin's algorithm cannot be directly applied to the class V1CAs. However, we can see that as long as the counter of a V1CA does not change, it acts like a finite state automaton. Another way to see it is that a V1CA behaves as a finite state automaton for a given fixed level. From this assumption, the idea of Neider and Loding's algorithm is to apply Angluin's algorithm on every level of the V1CA, using as many tables as there are levels in the tables prefixes. Using this technique, which is a refined version of Angluin's method, we obtain a behaviour graph of the V1CA language, up to a level $m$.

As explained in 3.1, in order to make sure the behaviour graph is correct, we use the partial equivalence query of the teacher. If it is not complete, we use the counter example to complete the tables.

The behaviour graph we obtain is infinite, but it is periodic. Note that it is periodic as the counter value, i.e the level of the graph, increases: the levels of the behaviour graph repeat themselves as the counter value increases. The next step in the algorithm is then to find a period in the behaviour graph be identifying an isomorphism in the graph. To do so, we pick a possible level $n$ where the period may start, and a possible width $k$ such that the period would be between level $n$ and $n+k$. We then split the behaviour graph into two sub-graphs: the graphs of level $n$ to $n + k$, and the graph of levels $n + k$ to $n + 2k$. If those two graphs are isomorphic, then we have found a potential period in the behaviour graph.

Once a period was found, the last step of the algorithm is to make the behaviour graph "loop to itself" to get a V1CA, that is to say, link the states of the repeating part together so it mimics the behaviour of the infinite period. The strategy is the following: let $Q_1$ be the states of the sub-graph from $n$ to $n + k$, $Q_2$ the states of the sub-graph from $n + k$ to $n + 2k$. We have couples

of states $(q_0, p_0), ..., (q_n, p_n)$ ($n \in \mathbb{N}$, $q_i \in Q_1, p_i \in Q_2$, $0 \leq i \leq n$) such that $q_i$ is isomorphic to $p_i$.

For a state $q_i \in Q_1$, we look for transitions $\delta(q_i, a) = q_{dest}$ ($q_{dest} \in Q, a \in \Sigma$ and $\chi(a) = 1$) that go from $q_i$ to another state using a symbol $a$ that increments the counter. If such a transition is found, we then add a new transition from $p_i$ to $q_{dest}$ using $a$. This new transition has no condition over the counter value.

Then, for a state $p_i \in Q_2$, we look for transitions $\delta(p_i, a) = q_{dest}$ ($q_{dest} \in Q, a \in \Sigma$ and $\chi(a) = -1$) that go from $p_i$ to another state using a symbol $a$ that decrements the counter. If such a transition is found, we add a new transition from $q_i$ to $q_{dest}$ using $a$. This transition has a condition over the counter value: it can only be used if the counter value is strictly above the value $n$, which is the value of the level of the period that was previously found. For every other existing transitions that start from $q_i$, those transitions now have a new condition over the counter value: they can only be used if the counter value is equal or below $n$.

Now that we have a finite V1CA, all states of levels greater that $n + k$ can be removed from the automaton. The last step is to ask the teacher if the V1CA is correct by using the equivalence query. If not, like the partial equivalence query, we use the counter example given by the teacher to complete the tables and find a new behaviour graph.

Note: it is possible that no periodic structure is found in the behaviour graph. This can be caused by a behaviour graph not large enough, that has not enough levels. Using counter examples from the teacher, the graph should grow and a periodic structure may be found later. It is also possible that the language the algorithm tries to figure out does not accept any word with a counter value greater than a bound $k$, in which case the behaviour graph does not need to grow above $k$, and the V1CA can actually be represented by a finite state automaton, as its language is regular. In any case, if no isomorphism is found, we return the behaviour graph as it is to the teacher for the equivalence test, as a V1CA that is actually a finite state automaton.

## 3.4 Learning a R1CA

We just described an algorithm to learn V1CA that:

1. Learns a behaviour graph using Angluin's algorithm.
2. Asks the teacher if the behaviour graph is correct.
3. Finds a periodic structure inside the behaviour graph.
4. Makes the behaviour graph loop to itself.

Let us now ask: "Can we use the same algorithm to learn a R1CA?".

The first issue about learning a R1CA is that for a word $w \in \Sigma^*$, we cannot know $\mathbf{cv}(w)$ as it relies on the R1CA we look for and not on the alphabet $\Sigma$. Being able to know the counter value of any word of $\Sigma^*$ is crucial for building Angluin's tables, and the algorithm cannot work without such information.

The idea is to introduce a new kind of teacher for R1CA learning. Like the previous teacher, this teacher has a membership query, an equivalence query and a partial equivalence query that remain just the same. However, we introduce a new query:

- **The counter query**: given a word $w \in \Sigma^*$, what is $\mathbf{cv}(w)$?

It is very important to point out that in order to answer such a query, the teacher must know about the R1CA we look for, as the counter value relies on the automaton, whereas our previous teacher only needed to know a V1CA language, and did not require any knowledge about an automaton.

Now that we introduced a way to know a word counter value, we were able to make the assumption that learning a R1CA is possible, and we could empirically try to do so using our V1CA learning algorithm. We wrote a few R1CA automata and gave them to a R1CA teacher, and for every language, we get the same result: Angluin tables grow forever and we can never get a closed and consistent set of tables.

The reason behind this result is that, when creating tables, we do not only fusion states with similar acceptance condition, but also states with the same counter value, unlike V1CA where a state has a unique counter value anyway. Such a restriction may cause an infinite loop where a specific word prefix never accepts any suffix, and keeps on growing its counter value.

There does exist a way to avoid this issue by introducing a special value to differentiate two types of suffixes: those for which we only look for acceptance, and those for which we look for acceptance and counter value. But we will not get into any further details as this would make the algorithm even more complex, and we conclude that the active learning algorithm for V1CAs cannot be easily and directly applied to R1CAs.

## 3.5 Personal work and improvements

### 3.5.1 A new V1CA representation

Since Neider and Loding visually represent behaviour graphs as a graph with an initial part, and a periodic part repeating along the graph levels, there is no real two dimensions representation of V1CAs, the reason being that, unlike finite state automata, V1CAs transitions depend on the counter value. However, the number of state and transitions is finite, and only the counter can take any arbitrary value, which makes V1CA possible to draw as a graphic, finite representation. When implementing a program generating V1CAs, it is easy to guess that representing a V1CA as a two dimensional diagram, instead of writing down labeled states, transitions and transition functions, is a very powerful tool to visualize and compare V1CAs.

Since the V1CAs we generate look a lot like a series of finite states automata, generated with Angluin strategy and only linked together with specific condition on the periodic part, these automata, before the final linkage step, are already easily representable in a two dimensional space as we would represent a finite state automaton. Instead of labeling states with arbitrary numbers to distinguish them, we choose to label them with the prefix used in Angluin algorithm as the representative prefix of the state. As for the transitions, we label them with the symbol used to go from one state to another. For reading purposes, we add a “+” or a “-” next to the symbols that increase or decrease the counter value. Alternatively, it is possible to represent the states by level, like a building, with states of the same level on the same floor.

After the behaviour graph gets linked, and becomes the V1CA that we look for, new appear. Such transitions can be seen in red and blue on 3.1. Those new transitions from a state $\delta(q_1, a) = q_2$ usually appear even though a transition $\delta(q_1, a) = q_3$ already exist. Naively showing those new transitions suggests that the V1CA is non-deterministic, where in reality it is deterministic and transitions have a condition over the counter value.

To represent a V1CA, while taking into consideration these conditions on transitions, we add the counter value condition of the transition to the label of the transition. It is now pretty easy to read and interpret the language recognized by V1CA using this visualisation. Note that,

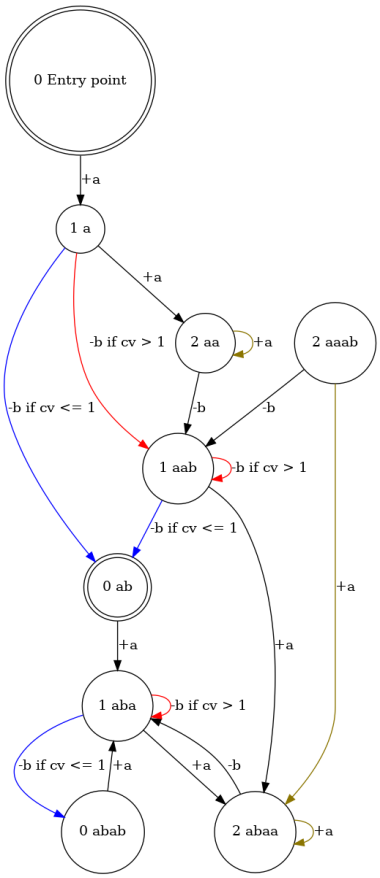


Figure 3.1: Representation of a V1CA accepting the language $a^n b^n$, $n \in \mathbb{N}$

sometimes, useless states may appear after the linkage process, as shown of the figure. Gold transition represent transitions that were not present in the behaviour graph, and that were added during the linkage process, but that have no condition over the counter value.

### 3.5.2 A faster algorithm with useless state removal

Moreover, we know that Angluin's algorithm returns complete automata. Therefore, the behaviour graph obtained using this method is *almost complete*, meaning that, for any behaviour graph $\mathscr{A}$ up to a level $m$, $Q$ its set of states and $\delta_0, ..., \delta_m$ its transitions function , $\forall q \in Q$, if $0 < \textbf{level}(q) < m$, $\forall a \in \widetilde{\Sigma}$, $\exists \delta_i$ such that $\delta_i(q, a) = q'$, $q' \in Q$. As an example, for most behaviour graphs, for every level of the graph, there is a "sink" state, a state from where it is impossible to encounter final states, and therefore recognize any language from this state other than the empty language. Such sink states are called *useless*, because they do not help characterizing the behaviour graph's V1CA language. Note that sub-graphs at level 0 and level $m$ are not necessarily complete because there is no transition towards states whose level is $-1$ or $m + 1$.

More generally, behaviour graphs can have multiple sources of useless states. Such states make the search for a sub-graph isomorphism even longer, as we need to try match more states. We suppose that the search of a periodic structure inside the behaviour graph is one of the

longest execution parts of the algorithm as the graph becomes bigger. Removing useless states from the behaviour graph could thus make the algorithm faster. In addition, since the behaviour graph is periodic up to a level $k$, we know that any useless state up to this level will appear indefinitely in the behaviour graph. Therefore, as long as all useless states are removed, removing useless states from the behaviour graph preserve its periodicity, and will not prevent the algorithm from finding a periodic structure in the graph.

Henceforth we can implement a simple useless state removal algorithm which works the same way as useless state removal algorithms in finite state automata. The more naive way is, for every state of the automaton, try to reach a final state with a simple depth-first graph-traversal algorithm. If no final state can be reached, the state (and any transition connected to the state) can be deleted. Then, we start from the initial state of the behaviour graph, and we process the same kind of graph traversal on the entire graph. Any state that was not encountered can be removed as well. Of course, such a naive state removal algorithm could be improved as a lot of solutions exist to make it faster, and this is marked as a potential improvement to the current state of the algorithm implementation.

### 3.5.3 A faster isomorphism search

As described in 3.3, finding a periodic structure, if it exists, in a behaviour graph works by arbitrarily choosing a level $n$ of where the period starts, arbitrarily choosing a width of period $k$, i.e the number of levels you need to go from $n$ to find a repetition of the graph. Then, from the two sub-graphs of levels $n$ to $n+k$ and $n+k$ to $n+2k$, we arbitrarily choose a permutation of states, i.e a set of pairs $\{q_0, q'_0\}, ..., \{q_m, q'_p\}$ in order to find out if the two sub-graphs are isomorphic. If such an isomorphism is invalid, then we try another permutation, or another value of $n$, or another value of $k$. As said before, such an algorithm can have a long execution time depending on the size of the behaviour graph. However, once a periodic structure is found, the algorithm stops even if all values of $n$, $k$ or permutations have not been tried. Also note that the possible values of $k$ depends on the value of $n$, and the possible permutations of states depend on the sub-graphs we chose, and thus depend on $n$ and $k$.

A good improvement to the algorithm would thus be to find the right level $n$, the right width $k$ and the right permutations with as few tries as possible. From there, questions such as "Should I start with the biggest value of $n$ possible first?" or "Should I pick the smallest value of $k$ first" arise. Of course, there is no generic good answer to those questions, as the right choice of those values heavily depends on the graph. But we could argue that maybe prioritizing the choice of specific values could work on *most* behaviour graphs. The idea is to verify that using benchmarks on a large number of V1CA languages and their behaviour graphs.

## 3.6 Benchmarking

Now that we introduced a few leads on how we could make the algorithm faster, we need to actually verify whether our improvements have a major impact on the program execution time. In order to do that, we need to benchmark the algorithm. On most automatic programs, benchmarking is pretty straightforward, as you just need to create two versions of the program, launch them and compare the execution time (of course, this is the most basic benchmarking technique, actual precise benchmarking can be way trickier). But in our case, the program is only semi-automatic, as we need a teacher (the user) to manually answer questions. In order to benchmark the program, we need to **automatize the teacher**, i.e automatically answer the membership, partial equivalence, and equivalence query.

To do so, we need to have a reference V1CA $\mathscr{A}_{ref}$, i.e. an automaton to be learned. Such an automaton can be given to our program by writing its attributes (its number of state, its initial state, its final states and its transitions) in a text file, and parsing this text file into a V1CA object during runtime.

The membership query is the easiest query to be made automatic using our reference V1CA. For a word $w \in \Sigma^*_{0,t}, t \geq 0$, the membership query returns *true* if $w \in L(\mathscr{A}_{ref})$, meaning that the word is accepted by the V1CA. All we need to do is to simulate an execution of the V1CA, while updating for each symbol of $w$ a current state and a current counter value. Remember that a word can only be accepted if the counter value never goes below 0, if the final counter value is 0, and if the last state is a final state.

### 3.6.1 Automation of the equivalence query

Let $\mathscr{A}_{test}$ be the V1CA generated by the program, and $\mathscr{A}_{ref}$ the reference V1CA. We say that the equivalence query is valid if $L(\mathscr{A}_{test}) = L(\mathscr{A}_{ref})$. Since $L(\mathscr{A}_{test}) = L(\mathscr{A}_{ref}) \iff L(\mathscr{A}_{test}) \subseteq L(\mathscr{A}_{ref})$ and $L(\mathscr{A}_{test}) \supseteq L(\mathscr{A}_{ref})$, in order to benchmark our program, we need to be able to tell whether a V1CA language is a subset of another, and then by extension, whether a V1CA is equivalent to another.

Where Srba [2009] shows a way to check V1CA equivalence by bi-simulation, the V1CA we obtain are deterministic, an thus we choose to process them in a simpler way. Indeed, it is common to write that $L(\mathscr{A}_{test}) \subseteq L(\mathscr{A}_{ref}) \iff L(\mathscr{A}_{test}) \cap \overline{\mathrm{L}(\mathscr{A}_{\mathrm{ref}})} = \emptyset$ using set theory. To check V1CA equivalence, we would then need to be able to process V1CA *intersection*, V1CA *complement* and V1CA *emptiness check*.

#### Complement of a V1CA

The complement is made by changing the acceptance condition of a V1CA. Indeed, the acceptance condition of a V1CA is, for a V1CA $\mathscr{A}$, $F$ its set of final states, for a word $w \in \widetilde{\Sigma}^*$, there is a run $(q_0, 0) \xrightarrow{w}_{\mathscr{A}} (q', 0)$ where $q' \in F$. We can read this as two conditions, "finish a run on a final state with a counter value of 0". However, this condition cannot just be flipped as "not finishing a run on a final state or finish with a counter value other than 0" because a V1CA, by essence, only accepts words whose counter value is 0, and we need the complementary of a V1CA to be a V1CA. The condition thus becomes: the run of $w$ is $(q_0, 0) \xrightarrow{w}_{\mathscr{A}} (q', 0)$ where $q' \notin F$. Note that because our V1CA are deterministic, we do not care here about multiple possible runs for a single word, and assume there is an unique run for each word.

To create such a graph, all we need to do is, for every state at level 0, if a state is final, make it non-final. Otherwise, make it final. Note that this only works on complete graphs, because a non-complete graph may hide states that would not be final. As said before, the behaviour graphs we obtain during the learning process are complete, so this is not a direct issue here, but it is important to keep this aspect in mind. If we choose to remove useless states from the V1CA obtained from active learning, and thus making it non-complete, then it can be made complete again by adding new sink states where no transition exists.

#### Emptiness check of a V1CA

Again, manipulating deterministic V1CA makes the emptiness check of a V1CA easier. In fact, for a V1CA $\mathscr{A}$, $L(\mathscr{A}) \neq \emptyset \iff \exists w \in \widetilde{\Sigma}^*, (q_0, 0) \xrightarrow{w}_{\mathscr{A}} (q, 0), q \in Q$, i.e the V1CA is "not empty"

if there exist a word which run is accepted by it, meaning that there is a path from the initial state to a final state.

To figure out if a deterministic V1CA's language is empty, we only need to process a graph traversal from the initial state of the V1CA, and check whether we ever encounter a final state. If yes, $L(\mathscr{A}) \neq \emptyset$. If not, $L(\mathscr{A}) = \emptyset$.

#### Intersection of two V1CAs

The intersection of two V1CA is inspired by classic deterministic finite state automata intersection. We define it as follows: let $\mathscr{A}_2$ and $\mathscr{A}_1$ two $n$-V1CAs **of the same level** and $Q_1$, $Q_2$ their set of states, the intersection of those two V1CAs is a V1CA $\mathscr{I}$ whose set of states is $Q$, such that $\forall q_1 \in Q_1$, if for a word $w \in \widetilde{\Sigma}^*$, $k = \mathbf{cv}(w)$ there is a run $(q_{0_1}, 0) \xrightarrow{w}_{\mathscr{A}_1} (q_1, k)$ and $\exists q_2 \in Q_2$ such that there is a run $(q_{0_2}, 0) \xrightarrow{w}_{\mathscr{A}_2} (q_2, k)$, then $\exists q \in Q$ such that there is a run $(q_0, 0) \xrightarrow{w}_{\mathscr{I}} (q, k)$. Thus, every possible run of $\mathscr{I}$ is also a possible run of $\mathscr{A}_1$ and $\mathscr{A}_2$. But a run that is not a run of $\mathscr{A}_1$ or $\mathscr{A}_2$ cannot be a run of $\mathscr{I}$.

The algorithm to create $\mathscr{I}$ is the following: we create an empty graph as the result automaton, then starting from the initial states of both automaton, we go through the automata with a simple graph traversal on both graphs at the same time, while labelling states. We copy all states and transition encountered during the traversal. However, a transition is only kept if it exists in both graphs. If not, graph traversal does not continue through that transition.

#### Increasing a V1CA level

The intersection technique explained above only works if both V1CAs are of the same level. If one of the V1CA is a $n$-V1CA, and the other is a $m$-V1CA, with $n < m$, this algorithm cannot be directly applied, and the $n$-V1CA must be artificially grown up to level $m$.

We can artificially increase the level of $n$-V1CA to a $(n{+}1)$-V1CA by creating a new level $n+1$ with as many states as the level $n$, and by copying every transition $\delta_n$ to the new level $\delta_{n+1}$. We can then repeat this operation to increase the level $n$ of the V1CA to an arbitrary higher level $m$.

### 3.6.2 Automation of the partial equivalence query

Let $\mathscr{B}$ a behaviour graph; even though it is called a graph, writing $L(\mathscr{B})$ makes sense, as a behaviour graph contains all the attributes of a V1CA, and is more of an automaton itself than just a graph structure, and we can apply all V1CA operations to it. Let $\mathscr{B}_{test}$ be the behaviour graph generated by the program, $\mathscr{B}_{ref}$ a reference behaviour graph. The partial equivalence check is valid if $L(\mathscr{B}_{ref} = L(\mathscr{B}_{test})$. The algorithm for the partial equivalence query thus works just as the equivalence query check algorithm.

However, we do not directly possess a reference behaviour graph $\mathscr{B}_{ref}$. We have to build it from $\mathscr{A}_{ref}$. To do that, we process a depth-first search traversal, through every configuration $\{(q, l) \mid q = q_1, ..., q_m \in Q;\ l = 0, ..., n;\ m = |Q|\}$ possible of the V1CA, and building the behaviour graph on-the-go only keeping existing transitions.

If the level of the behaviour graph we obtain is too small, we can increase the level of the V1CA before getting the behaviour graph, or increase the behaviour graph size in a similar manner we would do it for the V1CA.

# Implementation and results

## 4.1 Implementation details

Implementing such an algorithm is not trivial and a lot of questions depending on the programming language and the programming heuristics quickly arose. In order to understand the algorithm and have a clear view of its different steps, it was decided to first develop Angluin's algorithm, and then the V1CA learning algorithm using Python 3, without caring about performance and optimizations. Those implementations were far from being fast and elegant, but gave a first overview of what an active learning algorithm would look like, especially since no implementation of the algorithm was given in [Neider and Loding, 2010].

The main V1CA active learning algorithm was coded in C++ only using the standard library and the Boost library. Some of the graphs were implemented using boost adjacency list, so the algorithm sometimes relies on Boost API to modify and export the graphs. As for the data structure chosen to use tables, the Python algorithm would benefit from the Pandas library to quickly handle tables as a *data frame* object in an optimized and reliable way. Unfortunately, no such library exist in C++ and custom tables were made using the standard C++ vectors.

The Python version of the code's goal is not to be optimized, but it was made as a tool to apprehend active learning on V1CA and to start having intuitions about possible improvements and how to transform the theory of Neider and Loding's paper into a working program. The tables are stored using the Pandas library, which made it easy to manipulate tables, and the implementation of a V1CA is an object containing a set of states and transitions, but the rest of the program is not really object oriented, and relies more on Python function oriented programming. While the program was easy to modify and quite intuitive, the execution speed could drastically increase when learning more complex languages, with longer lists of larger tables. In fact, testing the speed of the implementation of a non-optimized Python code would clearly not be representative of the possible improvement that could be made on the program. That is why we needed, soon or later, to implement a C++ version of the code, more robust, and where the code would be designed to be optimized, memory and speed wise.

Since the latest version of program is in C++, it benefits from object oriented programming advantages.

In this program, a word is a sequence of ASCII characters, that behave as symbols. An alphabet is then a set of characters, and a pushdown alphabet is map, where keys are the characters, and values are the counter value operations of the matching symbol.

A V1CA object is designed just like its definition, along a pushdown alphabet. It has a number $|Q|$ of state, where each state is an integer between $0$ and $|Q|$. It has a maximal level, an initial

state (by convention, it is usually 0), a set of final states, and a transition function. The transition function is map where keys are an origin state, a character, and a current counter value, and values are a destination state. The operation effect of a transition can easily be deduced from the pushdown alphabet of the V1CA. We also keep track of some properties over states, using a map where values are the name of state (deduced from Angluin's table) and its level.

A R1CA object is designed just like a V1CA, except that its alphabet is not a pushdown alphabet, and the transition function values contain a counter value operation specific to each transition.

Note that a transition where the counter value is equal to the maximum level $m$ of a one-counter automaton is used for any counter value greater or equal to $m$.

A behaviour graph on the other hand is defined as a boost adjacency list, where properties about states and transitions are properties about the adjacency list's vertexes and edges. A behaviour graph is designed as the same object regardless of whether it is the behaviour graph of a V1CA or R1CA.

A teacher is an abstract object containing the membership query, the partial equivalence query and the equivalence query. There is multiple possible implementations of a teacher.

- The most naive one ask every query to the user on the *standard input*.
- The semi-automatic implementation, given a function, automatically answers the membership query using the function to tell whether a word is in a language.
- The fully automatic teacher uses a reference automaton and V1CA equivalence functions to automatically answer queries with no help from the user.
- The R1CA teacher that can be any of those teachers, but it necessarily contains a reference R1CA for counter value queries.

The tables used to find the behaviour graph are a list (a vector) of *dataframes*, an object containing vectors of vectors of same length containing boolean values, in order to form a table. Each row and columns are labeled by a string. Using specific methods, and given a teacher and its membership query, it can be easily filled without caring about the implementation.

The rest of the program is function oriented, and follows the algorithm described in Neider and Loding [2010]. The total C++ program is made of $3500+$ lines of code, cut into $35$ different files. It has changed a lot through time to be coherent with any new features added along the way. The Python version of this code on the other hand contains $938$ lines of code. The Python Angluin's algorithm on finite state automata code is only about $300$ lines of code long.

# Conclusion

In this report, we presented an active learning algorithm applied to the class of visibly one-counter automata. We showed that the algorithm as described in the State Of The Art could be improved, and we introduced a first implementation of this algorithm in Python and C++. We also showed that, unfortunately, this algorithm cannot be directly applied to R1CA. Of course, the algorithm can still be heavily improved, by implementing a faster useless state removal algorithm as an example.

However, when looking at the bigger picture, the use of visibly one-counter automata in literature for real life applications is limited, because V1CA languages lack expressiveness. Moreover, learning R1CA using a teacher that must have knowledge about the resulting automaton has very limited applications. In fact, it is shown in Isberner [2015] that you can apply active learning algorithm on *visibly pushdown automata*, which are more expressive, generic and thus have more real life applications.

# Bibliography


Alur, R. (2004). Visibly pushdown languages. (page 5)

Angluin, D. (1987). Learning regular sets from queries and counterexamples. (pages 4, 5, and 9)

Bermudez, M. E. and Schimpf, K. M. (1990). Practical arbi-trary lookahead lr parsing.

Bomohm, S., Gooller, S., and Jancar, P. (2013). Equivalence of deterministic one-counter automata is nl-complete.

Chitic, C. and Rosu, D. (2004). On validation of xml streams using finite state machines.

Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. (2000). Efficent algorithms for model checking pushdown systems.

Fahmy, A. and Roos, R. (1995). Efficient learning of real time one-counter automata.

Heizmann, M., Hoenicke, J., and Podelski, A. (2013). Software model checking for people who love automata.

Isberner, M. (2015). Foundations of active automata learning: An algorithmic perspective. (page 19)

Kumar, V. and Viswanathan, M. (2007). Visibly pushdown automata for streaming xml. (page 5)

Neider, D. and Loding, C. (2010). Learning visibly one-counter automata in polynomial time. (pages 5, 6, 9, 17, and 18)

Nerode, A. (1958). Linear automaton transformations.

Pnueli, A. (1977). The temporal logic of programs. (page 4)

Srba, J. (2009). Beyond language equivalence on visibly pushdown automata. (page 15)

Valiant, L. G. and Paterson, M. S. (1975). Deterministic one-counter automata.