Scalable Learning of One-Counter Automata via State-Merging Algorithms

Shibashis Guha
Tata Institute of Fundamental Research, Mumbai, India

Anirban Majumdar
Independent researcher, Kolkata, India

Prince Mathew
Indian Institute of Technology Goa, Ponda, India

A.V. Sreejith
Indian Institute of Technology Goa, Ponda, India

Abstract

We propose One-counter Positive Negative Inference (OPNI), a passive learning algorithm for deterministic real-time one-counter automata (DROCA). Inspired by the RPNI algorithm for regular languages, OPNI constructs a DROCA consistent with any given valid sample set.

We further present a semi-algorithm for active learning of DROCA using OPNI, and provide an implementation of the approach. Our experimental results demonstrate that this approach scales more effectively than existing state-of-the-art algorithms. We also evaluate the performance of the proposed approach for learning visibly one-counter automata.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases active learning, passive learning, one-counter automata, RPNI

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2025.35

Related Version *Extended Version*: https://arxiv.org/abs/2509.05762

Supplementary Material *Software*: https://doi.org/10.5281/zenodo.17310292 [9]
Software: https://doi.org/10.5281/zenodo.14604419 [8]

Funding Prince Mathew and A.V. Sreejith would like to thank CEFIPRA for the project IFC/6602-1/2022/77. A.V. Sreejith would also like to thank the DST Matrics grant for the project MTR/2022/000788.

1 Introduction

Automata learning and verification. Automata learning constitutes a correct-by-construction synthesis technique aimed at inferring formal models – such as finite-state machines or Mealy machines – from observed system behaviour. It ensures that the inferred model remains consistent with the observed data. Closely linked to formal verification, automata learning plays a key role in model inference, facilitating formal analysis and validation of system behaviour. It may also be viewed as an inductive synthesis framework tailored for the construction of finite-state programs.

Active and passive learning of automata. The L^* algorithm [3] of Angluin is a foundational algorithm in active automata learning. It learns a minimal deterministic finite automaton (DFA) that accepts an unknown regular language, using a query-based learning model. The learner interacts with a teacher (oracle) through two types of queries, called membership queries and equivalence queries. The L^* algorithm guarantees the learning of the minimal DFA in a number of steps polynomial in the size of the minimal DFA and the length of the longest counterexample.

Passive learning of DFA, for example, Regular Positive and Negative Inference (RPNI) [13], on the other hand, involves inferring a DFA from a fixed, finite dataset consisting of positive and negative examples of strings. The output of the algorithm is a DFA that accepts all positive examples and rejects all negative ones. The constructed DFA may depend on the input positive and negative examples. For arbitrary inputs, the DFA that is consistent with the input data may not be minimal.

One-counter automata and visibly one-counter automata. Deterministic one-counter automata [14] are a subclass of deterministic pushdown automata that operate with a single integer counter, which can be incremented, decremented, or tested for zero during transitions. They serve as a simple model for programs with minimal memory – more powerful than finite automata, but less expressive than general deterministic pushdown automata. Deterministic real-time one-counter automata (DROCA) [4] form a subclass of deterministic one-counter automata that do not have ε-transitions.

Visibly pushdown automata (VPA) [1, 2] are a restricted subclass of pushdown automata where the type of stack operation (push, pop, or no operation) is determined by the input symbol. This restriction allows VPA to retain much of the expressive power of pushdown automata while ensuring desirable closure properties and algorithmic properties similar to finite automata. Unlike pushdown automata, deterministic visibly pushdown automata are equally expressive as nondeterministic ones. Deterministic visibly one-counter automata (VOCA) are both DROCA and deterministic VPA.

Contributions. We present two new methods for learning DROCA, addressing both passive and active learning settings:

- Our first contribution (Section 3) is a passive learning algorithm, called OPNI. Given a labelled set S of words – partitioned into accepting and rejecting examples – along with counter values for all prefixes, OPNI synthesises a DROCA that is consistent with both the counter information and the acceptance labels.
- Our second contribution (Section 4) is an active learning procedure, OCA-L*. In this setting, the learner interacts with a teacher who possesses a target DROCA. The learner may issue membership, equivalence, and counter-value queries to infer the target automaton. OCA-L* adapts the MinOCA framework [11] by replacing its SAT-based subroutine with our passive learner OPNI, leading to a simpler and significantly more scalable method.

We implemented OCA-L* in Python and evaluated it on randomly generated DROCA. Our experiments (Section 5) show that OCA-L* outperforms the state-of-the-art MinOCA algorithm in terms of scalability: while MinOCA fails on most DROCA with more than 10 states, OCA-L* successfully learns most DROCA having 18 or fewer states. Additionally, we evaluated OCA-L* for the class of VOCA. In this case, OCA-L* demonstrates remarkable scalability, learning most VOCA with up to 60 states and nearly half with 100 states. In contrast, MinOCA fails to learn most VOCA with more than 20 states.

However, unlike MinOCA, OCA-L* need not necessarily learn the minimal DROCA.

Related works. In the passive learning framework, the learner receives a set of labelled examples and synthesises a model consistent with them. The classical RPNI algorithm [13] learns a DFA in polynomial time from a set of accepting and rejecting words.

Active learning of finite automata via membership and equivalence queries was pioneered by Angluin's L^* algorithm [3]. In the context of learning one-counter automata, Fahmy and Roos [7] established the decidability of learning DROCA. Later, Neider and Löding [12] proposed an algorithm for learning VOCA, employing an additional partial-equivalence query.

Building on Neider and Löding's techniques, Bruyére et al. [5] proposed learning DROCA by combining counter-value queries – which return the counter value after processing a word. Mathew et al. [11] proposed an alternate algorithm (MinOCA) to learn DROCA. It employs SAT solvers to compute a minimal separating DFA. These algorithms have worst-case exponential runtime. Recent work [10] shows that active learning of DROCA is possible in polynomial time. However, the degree of the polynomial is large, making it impractical.

Organisation. The remainder of the paper is organised as follows. Section 2 presents the necessary preliminaries. In Section 3, we introduce our passive learning algorithm, OPNI. Section 4 describes the active learning procedure, OCA-L*. Section 5 reports on our experimental evaluation of OCA-L* and compares its performance with the state-of-the-art MinOCA algorithm. Finally, Section 6 concludes the paper.

2 Preliminaries

For a finite set S, we denote by $|S|$ the cardinality of S. Non-negative integers are denoted by $\mathbb{N}$, and $[i, j]$ denotes the interval $\{i, i+1, \ldots, j\} \subseteq \mathbb{N}$. The sign of a non-negative integer d (denoted by $sign(d)$) is 0 if $d = 0$ and 1 otherwise.

An alphabet denoted by Σ is a finite set of letters and Σ^* (resp., Σ^+) represents the set of all words including (resp., excluding) the empty word ε over the alphabet Σ. For a word $w = a_0a_1a_2\ldots a_n$ and non-negative integers $i < j$, we use $w[i \cdots j]$ for the factor $a_ia_{i+1}\cdots a_j$ and $w[i]$ for the letter a_i. Given a set $S \subseteq \Sigma^*$ of words, we write $\mathsf{Pref}(S)$ to denote the set of prefixes of all words in S.

We now define the *length-lexicographic order* (denoted by $<_{\mathsf{llex}}$) on words. First, we fix an arbitrary total order $<_{\mathsf{lex}}$ on the letters in Σ. The order of words is inductively defined as follows: $u <_{\mathsf{llex}} v$ if either $|u| < |v|$, or $|u| = |v|$ and there exists $x, y, z \in \Sigma^*$ and $\sigma_1, \sigma_2 \in \Sigma$ such that $u = x\sigma_1 y,\ v = x\sigma_2 z$ and $\sigma_1 <_{\mathsf{lex}} \sigma_2$. Note that this ordering is total and well-founded. Note also that this order can naturally be extended to pairs of words as follows: $(u_1, v_1) <_{\mathsf{llex}} (u_2, v_2)$ if either $u_1 <_{\mathsf{llex}} u_2$, or $u_1 = u_2$ and $v_1 <_{\mathsf{llex}} v_2$.

2.1 One-Counter Automata

A *deterministic finite automaton* (DFA) is a tuple $D = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, $\delta \colon Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting (final) states.

Given a DFA D, we sometimes write $q \xrightarrow{a} q'$ to denote $\delta(q, a) = q'$. The transition function δ extends naturally to words in Σ^* in the usual way: for $w = a_1a_2\cdots a_n \in \Sigma^*$, we define $\delta(q, w) = \delta(\cdots\delta(\delta(q, a_1), a_2), \ldots, a_n)$. We will write $D(w)$ to denote $\delta(q_0, w)$. A word $w \in \Sigma^*$ is *accepted* by D if $D(w) \in F$. The language recognised by D, denoted as $\mathcal{L}(D)$, is the set of all accepted words. Two DFA D_1 and D_2 are equivalent if $\mathcal{L}(D_1) = \mathcal{L}(D_2)$.

▶ **Definition 1** (DROCA). *A deterministic real-time one-counter automaton (DROCA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta_0, \delta_1, F)$, where Q is a finite nonempty set of states, Σ is the input alphabet, $q_0 \in Q$ is the initial state, $\delta_0 : Q \times \Sigma \to Q \times \{0, +1\}$ and $\delta_1 : Q \times \Sigma \to Q \times \{0, +1, -1\}$ are the transition functions, and $F \subseteq Q$ is the set of final states.*

Consider a DROCA $\mathcal{A}$. A configuration of $\mathcal{A}$ is a pair $(q, n) \in Q \times \mathbb{N}$, where q denotes the current state and n is the counter value. The configuration $(q_0, 0)$ is called the *initial configuration* of $\mathcal{A}$. For an $e \in \{-1, 0, +1\}$ and letter a, the *transition* between the configurations (p, n) and $(q, n+e)$ on the symbol a is defined if $\delta_{sign(n)}(p, a) = (q, e)$. We use $(p, n) \xrightarrow{a} (q, n+e)$ to denote this. The run of a word $w = a_1 \ldots a_n$ in I'm not not $\mathcal{A}$, if it exists, is the sequence of configurations $(q_0, m_0) \xrightarrow{a_1} (q_1, m_1) \ldots \xrightarrow{a_{n-1}} (q_n, m_n)$ where $m_0 = 0$. We will write $(q_0, m_0) \xrightarrow{w} (q_n, m_n)$ to denote such a sequence. Further, we say m_n is the *counter-effect* of w (denoted by $\mathsf{ce}_{\mathcal{A}}(w)$). Note that the counter values always stay non-negative, implying a decrement is not permitted from a configuration with a zero counter value. We will say that w is accepted by $\mathcal{A}$ if and only if $q_n \in F$.

The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of all words accepted by $\mathcal{A}$. Similar to the case of DFA, we will write $\mathcal{A}(w)$ to denote the state in $\mathcal{A}$ that is reached after reading w. The definition of language equivalence is also similar to that above. Note that there are no ϵ-transitions in a DROCA. We say two DROCA $\mathcal{A}$ and $\mathcal{B}$ are *counter-synchronous* if $\mathsf{ce}_{\mathcal{A}}(w) = \mathsf{ce}_{\mathcal{B}}(w)$ for all words w.

A VOCA is a DROCA[1] where the input alphabet Σ is a union of three disjoint sets Σ_{call}, Σ_{ret}, and Σ_{int}. The VOCA increments (resp. decrements) its counter on reading a symbol from Σ_{call} (resp. Σ_{ret}). The counter value is unchanged on reading a symbol from Σ_{int}.

▶ **Definition 2** (VOCA). *A visibly one-counter automaton (VOCA) is a tuple $\mathcal{A} = (Q, \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{int}, q_0, \delta_0, \delta_1, F)$, where Q is a finite nonempty set of states, $\Sigma = \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{int}$ is the input alphabet, $q_0 \in Q$ is the initial state, $\delta_0 : Q \times \Sigma \to Q \times \{0, +1, -1\}$, $\delta_1 : Q \times \Sigma \to Q \times \{0, +1, -1\}$ are the transition functions, and $F \subseteq Q$ is the set of final states.*

The notions of counter-effect, runs and transitions for VOCA remain the same as those of DROCA. From the definition, VOCA are deterministic. The counter-effect of a transition is solely based on Σ, making the starting state and counter value irrelevant. For $\sigma \in \Sigma$:

$$\mathsf{ce}_{\mathcal{A}}(\sigma) = \begin{cases} 1, & \text{if } \sigma \in \Sigma_{call} \\ -1, & \text{if } \sigma \in \Sigma_{ret}, \text{ and} \\ 0, & \text{if } \sigma \in \Sigma_{init} \end{cases}$$

If $w = \sigma_1\sigma_2 \ldots \sigma_n$ for some $n \in \mathbb{N}$ and $\sigma_1, \ldots, \sigma_n \in \Sigma$, then $\mathsf{ce}_{\mathcal{A}}(w) = \mathsf{ce}_{\mathcal{A}}(\sigma_1) + \mathsf{ce}_{\mathcal{A}}(\sigma_2) + \ldots + \mathsf{ce}_{\mathcal{A}}(\sigma_n)$. Since the counter value can never go below zero, there will be words that do not have a valid run in a VOCA. These words will be considered as rejected.

2.2 RPNI: a passive learning algorithm for DFA

Learning framework. A pair of sets of words $\mathsf{S} = \mathsf{S}_+ \cup \mathsf{S}_-$ over an alphabet Σ is called a *sample set*, and words in S_+ (resp., S_-) are called *positive* (resp., *negative*) samples. A sample set S is called *consistent* if it satisfies the condition: $\mathsf{S}_+ \cap \mathsf{S}_- = \emptyset$. Typically, in a *passive learning* framework for DFA, a *learner* is given an alphabet Σ, and a sample set S over Σ, that is consistent, and the objective is to construct an automaton D such that D accepts all the words in S_+ and rejects all the words in S_-. In that case, overloading the notation, we will say that D is *consistent* with the sample set S.

RPNI algorithm. Here we recall RPNI, a state-of-the-art passive learning algorithm for DFA, originally proposed in [13]. Intuitively, the algorithm starts with a prefix tree acceptor (PTA) constructed from the positive samples S_+, and then iteratively merges states of the PTA in a specific order (precisely, the length-lexicographic order), keeping the resulting DFA consistent with the samples set S. A pseudo-code of our version of RPNI is given in Algorithm 1. Below, we explain different steps of the algorithm.

[1] Syntactically, in a VOCA, the δ_0 transition function includes -1. However, this is not semantically allowed since the counter value can never go below zero.

Algorithm 1 RPNI: a passive learning algorithm for DFA.

```
Input: An alphabet $\Sigma$, and a consistent sample set $\mathsf{S} = \mathsf{S}_+ \cup \mathsf{S}_-$ over $\Sigma$.
Output: A DFA $D$ consistent with the sample.
1 Construct PTA: Build a prefix tree acceptor $\mathsf{PT}$ from $\mathsf{S}_+$.
2 Pairing: $P = \{(u, v) \mid v <_{\mathsf{llex}} u \text{ and } u, v \in \mathsf{Pref}(\mathsf{S}_+)\}$.
3 Ordering $P$: Sort the elements of $P$ according to the increasing $<_{\mathsf{llex}}$ order.
4 Initialize $D \leftarrow \mathsf{PT}$.
5 while $P$ is not empty do
6     Pop $(u, v)$ the minimal element from $P$.
7     if $D(u)$ and $D(v)$ can be merged then
8         $D \leftarrow \text{Merge}(D; D(u), D(v))$.
9 return $D$.
```

We first construct a PTA (line 1) from the positive samples S_+, which can formally be defined as a DFA $\mathsf{PT} = (Q, \Sigma, \delta, q_\varepsilon, F)$, where $Q = \{q_u \mid u \in \mathsf{Pref}(\mathsf{S}_+)\}$ represents the set of states corresponding to all prefixes of words in S_+, δ is defined as follows: for all $u, u\sigma \in \mathsf{Pref}(\mathsf{S}_+)$ with $\sigma \in \Sigma$, we have $\delta(q_u, \sigma) = q_{u\sigma}$, the initial state q_ε is the state corresponding to the empty word ε, and $F = \{q_u \mid u \in \mathsf{S}_+\}$ is the set of all states that correspond to words in S_+. We call a prefix u the representative of the state q_u. Note that PT accepts exactly the words in S_+.

We then aim to merge states of the PTA in the $<_{\mathsf{llex}}$ ordering of their representatives, while preserving the consistency of the resulting DFA with the negative samples. To this end, we consider all pairs of prefixes (u, v) in $\mathsf{Pref}(\mathsf{S}_+)$ such that u is greater than v with respect to the $<_{\mathsf{llex}}$ ordering (line 2), and sort this set in the increasing $<_{\mathsf{llex}}$ order (line 3). We then initialise the main loop of RPNI with PT (line 4), and continue until P becomes empty (line 5). Let D be the DFA at the beginning of an iteration. First, pop (u, v) which is the minimal element from P (line 6). Then perform a merge of the states $D(u)$ and $D(v)$ of D, *i.e.*, check whether the merged DFA is consistent with respect to the sample set. If that is indeed (resp., not) the case, we accept (resp., discard) the merge, and update D accordingly (lines 7-8). Finally, we terminate when P becomes empty, and return the corresponding DFA.

The procedure *Merge* (line 8) takes as input a DFA D and two states $D(u)$ and $D(v)$, and returns a new DFA D' that is obtained from D by merging those two states. We give a high-level idea of the procedure. First, if $D(u) = D(v)$, then $D' = D$. Otherwise, let us denote $D(u)$ and $D(v)$ by q_u and q_v, respectively. Also, recall that since (u, v) is in the ordered set, $v <_{\mathsf{llex}} u$. Let q' be such that there exists a transition $q' \xrightarrow{a} q_u$ in D for some $a \in \Sigma$. Then construct D' as follows: delete the state q_u from D, redirect all incoming transitions of q_u to q_v, and all outgoing transitions from q_u will now be outgoing transitions of q_v. To make D' deterministic, we may need to recursively *fold* the subtree of q_u into q_v: if q_u and q_v have successors q'_u and q'_v on the same letter a, respectively, then merge q'_u and q'_v as well, and repeat this process until there is no common successor remaining.

The termination and correctness of Algorithm 1 were established in [13]. In Section 3, we will use RPNI as a black-box in our passive learning algorithm for DROCA.

3 OPNI: a passive learning algorithm for DROCA

In this section, we introduce OPNI, a passive learning algorithm for DROCA.

Learning framework. For passive learning of DROCA, we enhance the input of the learning algorithm with the counter values of prefixes of all words in the sample set. More formally, we assume that along with a consistent sample set $\mathsf{S} = \mathsf{S}_+ \cup \mathsf{S}_-$ over an alphabet Σ, the learning algorithm is also provided with the counter-effects for all words in $\mathsf{Pref}(\mathsf{S})$ defined by the function $\mathsf{ce} : \mathsf{Pref}(\mathsf{S}) \to \mathbb{N}$. The learner aims to construct a DROCA $\mathcal{A}$ that satisfies the following two properties: (1) it accepts all words in S_+ and rejects all words in S_-, and (2) for all $w \in \mathsf{Pref}(\mathsf{S})$, $\mathsf{ce}(w) = \mathsf{ce}_{\mathcal{A}}(w)$. In this case, we will say that $\mathcal{A}$ is *consistent* with the sample set S.

As in [11], we define the following two functions.

1. Let $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$. Define $\mathsf{Act} : \Sigma^* \to \{0,1\} \times \{0,1,-1,\mathtt{x}\}^k$ which, given a word w, returns a tuple consisting of the sign of the counter value reached after reading w along with the effect on the counter on reading each letter from Σ after w. Formally, for any word w, and $i \in [0,k]$,

$$\mathsf{Act}(w)[i] = \begin{cases} sign(\mathsf{ce}(w)) & \text{if } i = 0 \\ \mathsf{ce}(w\sigma_i) - \mathsf{ce}(w) & \text{if } i > 0 \text{ and } \mathsf{ce}(w\sigma_i) \text{ is known} \\ \mathtt{x} & \text{if } i > 0 \text{ and } \mathsf{ce}(w\sigma_i) \text{ is not known} \end{cases}$$

   Given words w_1, w_2, we say that $\mathsf{Act}(w_1)$ is similar to $\mathsf{Act}(w_2)$ (denoted $\mathsf{Act}(w_1) \sim \mathsf{Act}(w_2)$) if one of the following happens: (a) for all $i \in [1,k]$, we have $\mathsf{Act}(w_1)[i] = \mathtt{x}$ or $\mathsf{Act}(w_2)[i] = \mathtt{x}$ or $\mathsf{Act}(w_1)[i] = \mathsf{Act}(w_2)[i]$, or (b) the sign of the counter values for words w_1 and w_2 are different, that is, $\mathsf{Act}(w_1)[0] \neq \mathsf{Act}(w_2)[0]$. (As it will be clear later, when $\mathsf{Act}(w_1) \sim \mathsf{Act}(w_2)$, we might be able to merge the states reached after reading w_1 and w_2 respectively. The merging may be possible if the signs of the counter values are different, regardless of the other components in Act.) Otherwise, $\mathsf{Act}(w_1)$ and $\mathsf{Act}(w_2)$ are not similar, and we write $\mathsf{Act}(w_1) \not\sim \mathsf{Act}(w_2)$. For $\sigma \in \Sigma$, we write $\mathsf{Act}(w)|_\sigma$ to denote the entry in $\mathsf{Act}(w)$ that corresponds to the letter σ.
2. Define a function that, intuitively, given a word w, annotates each letter $w[i]$ of w with the sign of the counter-value reached, upon reading the prefix $w[0 \cdots i-1]$. To that end, for an alphabet Σ, we define the modified alphabet $\widetilde{\Sigma} = \bigcup_{\sigma\in\Sigma}\{\sigma^0, \sigma^1\}$. Then, given a word $w \in \Sigma^+$ and the counter-values ce of all its prefixes, we define the encryption function as follows: for $i \in [0, |w|-1]$,

$$\mathsf{Enc}(w)[i] = \begin{cases} w[i]^0 & \text{if } i = 0 \\ w[i]^{sign(\mathsf{ce}(w[0\cdots i-1]))} & \text{if } i > 0 \end{cases}$$

   Additionally, $\mathsf{Enc}(\epsilon) = \epsilon$.

▶ **Example 3.** Let $\Sigma = \{a, b\}$ and suppose the following counter-value information are given: $\mathsf{ce} = \{\varepsilon \to 0, a \to 1, b \to 0, ab \to 0, bb \to 1\}$. Then $\mathsf{Act}(a) = (sign(\mathsf{ce}(a)), \mathsf{ce}(aa) - \mathsf{ce}(a), \mathsf{ce}(ab) - \mathsf{ce}(a)) = (1, \mathtt{x}, -1)$. On the other hand, the effect of the Enc function on the word ab, for example, will be a^0b^1.

Algorithm 2 OPNI: a passive learning algorithm for DROCA.

```
Input: An alphabet $\Sigma$, a consistent sample set $\mathsf{S} = \mathsf{S}_+ \cup \mathsf{S}_-$ over $\Sigma$, the function
       $\mathsf{ce} : \mathsf{Pref}(\mathsf{S}) \to \mathbb{N}$.
Output: A DROCA $\mathcal{A}$ consistent with the sample.
1 Incorporate counter-actions: Consider the modified sample set $\widehat{\mathsf{S}} = \widehat{\mathsf{S}}_+ \cup \widehat{\mathsf{S}}_-$ by
    replacing every word $u$ in $\mathsf{S}$ with $\mathsf{Enc}(u)$.
2 Enriching the alphabet: Let $\Sigma_{\mathsf{Act}} = \{\mathsf{Act}(w) \mid w \in \mathsf{Pref}(\mathsf{S})\}$. Then, enrich the
    alphabet $\Sigma$ with $\Sigma_{\mathsf{Act}}$ as follows: $\widehat{\Sigma} = \widetilde{\Sigma} \cup \Sigma_{\mathsf{Act}}$.
3 Enriching the sample set: For every word $w \in \mathsf{Pref}(\mathsf{S})$: add $\mathsf{Enc}(w) \cdot \mathsf{Act}(w)$ to $\widehat{\mathsf{S}}_+$,
    and for every other $\mathsf{op} \in \Sigma_{\mathsf{Act}}$ such that $\mathsf{op} \not\sim \mathsf{Act}(w)$, add $\mathsf{Enc}(w) \cdot \mathsf{op}$ to $\widehat{\mathsf{S}}_-$.
4 Apply RPNI on $\widehat{\mathsf{S}}$: Run RPNI on the sample set $\widehat{\mathsf{S}}$ over $\widehat{\Sigma}$. Let $\widehat{D} \leftarrow$ RPNI $(\widehat{\mathsf{S}}; \widehat{\Sigma})$.
5 Construct a DROCA $\mathcal{A}$ from $\widehat{D}$ using the Procedure ConstructOCA.
6 return $\mathcal{A}$.
```

OPNI algorithm. We now introduce OPNI, a passive learning algorithm for DROCA, that uses RPNI as a subroutine. A pseudo-code of the algorithm is given in Algorithm 2. The algorithm can be broadly divided into two major steps: (1) a preprocessing step that creates an enriched sample set (lines 1-3, Algorithm 2), and (2) the inference step that constructs a DROCA consistent with the sample set with the help of RPNI.

In the preprocessing step, we incorporate the counter information about prefixes of S to construct an input $\widehat{\mathsf{S}}$ for RPNI. To that end, we first replace every word u in S with its encoding $\mathsf{Enc}(u)$ (line 1). Let us denote by $\mathsf{Enc}(\mathsf{S})$ the set $\{\mathsf{Enc}(u) \mid u \in \mathsf{S}\}$. We then consider the set $\Sigma_{\mathsf{Act}} = \{\mathsf{Act}(w) \mid w \in \mathsf{Pref}(\mathsf{S})\}$, and the modified alphabet $\widehat{\Sigma} = \widetilde{\Sigma} \cup \Sigma_{\mathsf{Act}}$ (line 2). Recall that, for an alphabet Σ, the alphabet $\widetilde{\Sigma}$ is defined as the set $\bigcup_{\sigma \in \Sigma} \{\sigma^0, \sigma^1\}$. Finally, for every prefix $w \in \mathsf{Pref}(\mathsf{S})$, we add the word $\mathsf{Enc}(w) \cdot \mathsf{Act}(w)$ to $\widehat{\mathsf{S}}_+$, and for every other $\mathsf{op} \in \Sigma_{\mathsf{Act}}$ such that $\mathsf{op} \not\sim \mathsf{Act}(w)$, we add the word $\mathsf{Enc}(w) \cdot \mathsf{op}$ to $\widehat{\mathsf{S}}_-$ (line 3).

In the inference step, we run RPNI on the sample set $\widehat{\mathsf{S}}$ over the alphabet $\widehat{\Sigma}$ (line 4). Let $\widehat{D}$ denote the output DFA of RPNI, then we construct a DROCA $\mathcal{A}$ from $\widehat{D}$ using the construction given in the Procedure ConstructOCA (line 5).

Henceforth, we fix a $\Sigma, \mathsf{S}, \mathsf{ce}$ and $\widehat{D}$. In the remainder of this section, we present the Procedure ConstructOCA and prove the correctness of the OPNI algorithm. We start with the following lemmas (Lemmas 4 and 5) about the DFA $\widehat{D}$.

▶ **Lemma 4.** *The* DFA $\widehat{D}$ *satisfies the following two properties:*

1. *for all words $w \in \mathsf{S}_+$ (resp. $w \in \mathsf{S}_-$), we have $\mathsf{Enc}(w) \in \mathcal{L}(\widehat{D})$ (resp. $\mathsf{Enc}(w) \notin \mathcal{L}(\widehat{D})$),*
2. *for any two words $w_1, w_2 \in \mathsf{Pref}(S)$, if the runs on $\mathsf{Enc}(w_1)$ and $\mathsf{Enc}(w_2)$ reach the same state in $\widehat{D}$, then $\mathsf{Act}(w_1) \sim \mathsf{Act}(w_2)$.*

Proof.

1. From line 1 of Algorithm 2, we have that, for all $w \in \mathsf{S}_+$ (resp. $w \in \mathsf{S}_-$), $\mathsf{Enc}(w) \in \widehat{\mathsf{S}}_+$ (resp. $\mathsf{Enc}(w) \in \widehat{\mathsf{S}}_-$). Then the property holds from the correctness of RPNI.
2. Assume, towards a contradiction, that there exist words $w_1, w_2 \in \mathsf{S}$, such that the runs on $\mathsf{Enc}(w_1)$ and $\mathsf{Enc}(w_2)$ reach the same state in $\widehat{D}$, but $\mathsf{Act}(w_1) \not\sim \mathsf{Act}(w_2)$. From line 3 of Algorithm 2, we have that, $\mathsf{Enc}(w_1) \cdot \mathsf{Act}(w_1) \in \widehat{\mathsf{S}}_+$, and $\mathsf{Enc}(w_2) \cdot \mathsf{Act}(w_1) \in \widehat{\mathsf{S}}_-$. Then due to the correctness of RPNI, it follows that $\mathsf{Enc}(w_1) \cdot \mathsf{Act}(w_1) \in \mathcal{L}(\widehat{D})$, and $\mathsf{Enc}(w_2) \cdot \mathsf{Act}(w_1) \notin \mathcal{L}(\widehat{D})$. Now since $\mathsf{Enc}(w_1)$ and $\mathsf{Enc}(w_2)$ reaches the same state in $\widehat{D}$, it must also be the case that, $\mathsf{Enc}(w_2) \cdot \mathsf{Act}(w_1) \in \mathcal{L}(\widehat{D})$, which is a contradiction. ◀

▶ **Lemma 5.** *For all $q, q' \in Q$ and $\sigma \in \Sigma$ such that $q \xrightarrow{\sigma^0} q'$ (resp., $q \xrightarrow{\sigma^1} q'$) is a transition in $\widehat{D}$, there must exist $\mathsf{act} \in \Sigma_{\mathsf{Act}}$ and $q_f \in Q$ such that $\mathsf{act}[0] = 0$ (resp., $\mathsf{act}[0] = 1$) and $q \xrightarrow{\mathsf{act}} q_f$. Furthermore, $\mathsf{act}|_\sigma \in \{0, +1\}$ (resp., $\mathsf{act}|_\sigma \in \{-1, 0, +1\}$).*

Proof. Let $q, q' \in Q$ and $\sigma \in \Sigma$ be such that $q \xrightarrow{\sigma^0} q'$. Then, there exists $w\sigma \in \mathsf{Pref(S)}$ such that $q_0 \xrightarrow{\mathsf{Enc}(w)} q$ in $\widehat{D}$ with $\mathsf{ce}(w) = 0$ and $\mathsf{Act}(w)[0] = 0 \in \Sigma_{\mathsf{Act}}$. The latter condition implies that $\mathsf{Enc}(w) \cdot \mathsf{Act}(w) \in \widehat{\mathsf{S}}_+$. Let $\mathsf{act} = \mathsf{Act}(w)$. Therefore, from the correctness of RPNI, we have that $\mathsf{Enc}(w) \cdot \mathsf{act} \in \mathcal{L}(\widehat{D})$, which implies that there exists $q_f \in Q$ such that $q \xrightarrow{\mathsf{act}} q_f$ in $\widehat{D}$. Since $w\sigma$ is in $\mathsf{Pref(S)}$, and $\mathsf{ce}(w\sigma)$ is given in the input, we have that $\mathsf{act}|_\sigma \in \{0, +1\}$. The proof for the other case, when the counter-effect of w is positive, is similar. ◀

Procedure ConstructOCA. Let $\widehat{D} = (Q, \widehat{\Sigma}, q_0, \delta, F)$. We then define the DROCA $\mathcal{A} = (Q, \Sigma, q_0, \delta_0, \delta_1, F)$ where δ_0 and δ_1 are specified as follows. For every $q \in Q$, $\sigma \in \Sigma$, if there is a transition $q \xrightarrow{\sigma^0} q'$ for some q' in $\widehat{D}$, then, from Lemma 5, there exist $\mathsf{act} \in \Sigma_{\mathsf{Act}}$ and $q_f \in Q$ such that $\mathsf{act}[0] = 0$ and $q \xrightarrow{\mathsf{act}} q_f$ with $\mathsf{act}|_\sigma \in \{0, +1\}$. Let $c = \mathsf{act}|_\sigma$. Then define $\delta_0(q, \sigma) = (q', c)$. The definition of δ_1 is analogous.

▶ **Lemma 6.** *Let $\mathcal{A} =$ ConstructOCA$(\widehat{D})$. Then, $\mathcal{A}$ is consistent with the sample set* S*.*

Proof. We know, from Lemma 4, that for any word $w_1, w_2 \in \Sigma^*$, if $\widehat{D}$ on reading $\mathsf{Enc}(w_1)$ and $\mathsf{Enc}(w_2)$ reaches the same state, then $\mathsf{Act}(w_1)$ is similar to $\mathsf{Act}(w_2)$. This implies that the transition functions of $\mathcal{A}$ are well-defined. We will now show that $\mathcal{A}$ is consistent with the sample S.

▷ Claim 1. For all $w \in \mathsf{Pref(S)}$, $\mathsf{ce}(w) = \mathsf{ce}_{\mathcal{A}}(w)$.

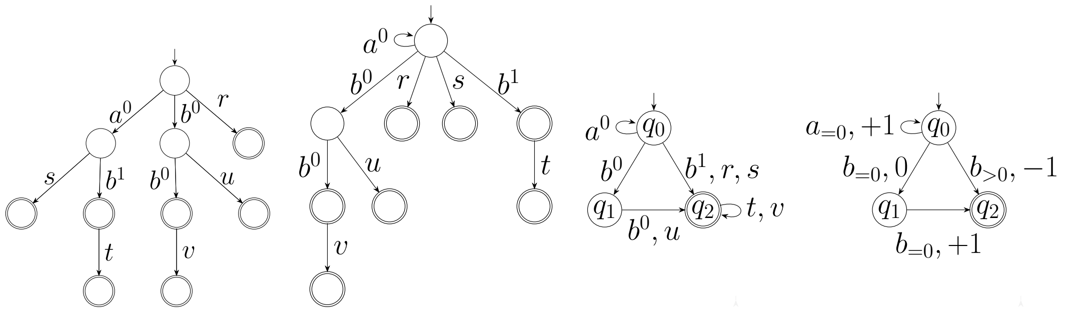
Proof. We will prove this claim by induction on the length of w. This is trivial for $w = \varepsilon$ (base case), since by definition, $\mathsf{ce}(\varepsilon) = \mathsf{ce}_{\mathcal{A}}(\varepsilon) = 0$. Now, assume that the claim is true for all words u of length less than or equal to l, for some $l \geq 0$. Let $u\sigma$ be a prefix in $\mathsf{Pref(S)}$. We will show: $\mathsf{ce}(u\sigma) = \mathsf{ce}_{\mathcal{A}}(u\sigma)$. Let $q, q' \in Q$ be such that $\widehat{D}(\mathsf{Enc}(u)) = q$, and $q \xrightarrow{\sigma^c} q'$ is a transition in $\widehat{D}$, where $c = \mathit{sign}(\mathsf{ce}(u))$. Then, by the construction of $\mathcal{A}$, we have that $\delta_c(q, \sigma) = (q', t)$ where $t = \mathsf{ce}(u\sigma) - \mathsf{ce}(u)$. Therefore, $\mathsf{ce}_{\mathcal{A}}(u\sigma) = \mathsf{ce}_{\mathcal{A}}(u) + t$. Using the induction hypothesis, we can rewrite the above equation as $\mathsf{ce}_{\mathcal{A}}(u\sigma) = \mathsf{ce}(u) + t = \mathsf{ce}(u\sigma)$. This concludes the proof of the claim. ◁

Claim 1 also ensures that $\mathsf{Enc}(w) = \mathsf{Enc}_{\mathcal{A}}(w)$. By construction of $\mathcal{A}$, for all $w \in \mathsf{Pref(S)}$, $\widehat{D}(\mathsf{Enc}(w)) = \mathcal{A}(w)$. Since, from Lemma 4, for all $w \in \mathsf{S}_+$, we have $\mathsf{Enc}(w) \in \mathcal{L}(\widehat{D})$, thus we conclude that $w \in \mathcal{L}(\mathcal{A})$. Similarly for $w \in \mathsf{S}_-$, we have $w \notin \mathcal{L}(\mathcal{A})$. Together with Claim 1, this concludes the proof of the lemma. ◀

Let us now illustrate the OPNI algorithm with an example.

▶ **Example 7.** Let $\mathsf{S} = \mathsf{S}_+ \cup \mathsf{S}_-$ be a sample set over an alphabet $\Sigma = \{a, b\}$ with $\mathsf{S}_+ = \{ab, bb\}$ and $\mathsf{S}_- = \{a, b\}$. Suppose the counter-values for words in $\mathsf{Pref(S)}$ are: $\mathsf{ce} = \{\varepsilon \to 0, a \to 1, b \to 0, ab \to 0, bb \to 1\}$. Below, we illustrate different steps of OPNI on this example.

- Using the Enc function, construct (line 1) the modified sample set $\widehat{\mathsf{S}} = \widehat{\mathsf{S}}_+ \cup \widehat{\mathsf{S}}_-$ where $\widehat{\mathsf{S}}_+ = \{a^0b^1, b^0b^0\}$ and $\widehat{\mathsf{S}}_- = \{a^0, b^0\}$ over the alphabet $\widehat{\Sigma} = \{a^0, a^1, b^0, b^1\}$.
- Then, we construct (line 2) the set Σ_{Act} as follows. We evaluate the Act function for words in $\mathsf{Pref}(S)$. For example, $\mathsf{Act}(\varepsilon) = (\mathit{sign}(\mathsf{ce}(\varepsilon)), \mathsf{ce}(a) - \mathsf{ce}(\varepsilon), \mathsf{ce}(b) - \mathsf{ce}(\varepsilon)) = (0, +1, 0)$. Similarly, we compute $\mathsf{Act}(w)$ for words in $\mathsf{Pref(S)}$, and get $\mathsf{Act} = \{\varepsilon \to$



(a) Prefix Tree. **(b)** Merge(PT;q_ε, q_{a^0}). **(c)** Output of RPNI. **(d)** Output DROCA.

Figure 1 Different steps of the OPNI algorithm on Example 7. Figure 1a represents the prefix tree PT; Figure 1b represents the DFA after merging q_ε and q_{a^0}; Figure 1c is the DFA that is the output of RPNI; Figure 1d is the DROCA output by the OPNI algorithm.

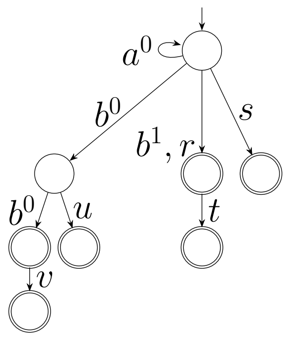
$(0, +1, 0), a \to (1, \mathtt{x}, -1), ab \to (0, \mathtt{x}, \mathtt{x}), b \to (0, \mathtt{x}, +1), bb \to (1, \mathtt{x}, \mathtt{x})\}$. For simplicity and better readability, we assign symbols from the English alphabet to these action tuples, and write: $(0, +1, 0) = r$, $(1, \mathtt{x}, -1) = s$, $(0, \mathtt{x}, \mathtt{x}) = t$, $(0, \mathtt{x}, +1) = u$, and $(1, \mathtt{x}, \mathtt{x}) = v$ and therefore, $\Sigma_{\mathsf{Act}} = \{r, s, t, u, v\}$. Consequently, $\widehat{\Sigma} = \{a^0, a^1, b^0, b^1, r, s, t, u, v\}$.

- We then enrich the sample set $\widehat{\mathsf{S}} = \widehat{\mathsf{S}}_+ \cup \widehat{\mathsf{S}}_-$ as in line 3 of the algorithm. For example, since ε is a prefix of S, we will include $\mathsf{Enc}(\varepsilon) \cdot \mathsf{Act}(\varepsilon) = r$ in $\widehat{\mathsf{S}}_+$, and since u is not similar to r, we include u into $\widehat{\mathsf{S}}_-$. Doing this for other elements in $\widehat{\mathsf{S}}$, we get the following sets: $\widehat{\mathsf{S}}_+ = \{a^0 b^1, b^0 b^0\} \cup \{r, a^0 s, a^0 b^1 t, b^0 u, b^0 b^0 v\}$, and $\widehat{\mathsf{S}}_- = \{a^0, b^0\} \cup \{u, b^0 r\}$.
- Next, we apply RPNI (line 4) to this newly constructed sample set $\widehat{\mathsf{S}}$ over the alphabet $\widehat{\Sigma}$. Different steps of the RPNI algorithm on $\widehat{\mathsf{S}}$ are shown in Figure 1. The first two steps are depicted in Figure 1a and Figure 1b, and Figure 1c represents the output of RPNI on $\widehat{\mathsf{S}}$. The intermediate steps of RPNI are shown in Figure 2.
- Finally, we construct a DROCA $\mathcal{A}$ from $\widehat{D}$ using the Procedure ConstructOCA. For example, consider the transition $q_0 \xrightarrow{b^1} q_2$ in $\widehat{D}$. Due to Lemma 5, there must exist a transition from its source state q_0 on an action act with $\mathsf{act}[0] = 1$. We deduce that $\mathsf{act} = r = (1, \mathtt{x}, -1)$, and the transition is $q_0 \xrightarrow{r} q_2$. Therefore, according to ConstructOCA procedure, we assign $\delta_1(q_0, b) = (q_2, -1)$. Figure 1d represents the output DROCA of ConstructOCA on $\widehat{D}$.

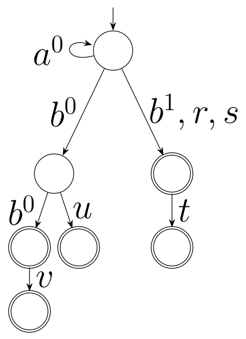
Justification of counter values in the input. We make use of the counter information in the preprocessing step to construct two sets: $\mathsf{Enc}(\mathsf{S})$ and Σ_{Act}. The counter-actions and signs of counter values in the input are required to prevent RPNI on the sample set $\widehat{\mathsf{S}}$ (line 4) from doing some of the merges. Let us illustrate this with an example. Suppose in a certain iteration of RPNI, we are trying to merge two states $D(u)$ and $D(v)$, for some prefixes u and v of S_+, and let both states have outgoing transitions on a letter a. As such, if there is no information on the counter-actions, RPNI will merge these two states, provided that the merged automaton is consistent with S_-. However, suppose that $sign(\mathsf{ce}(u)) = sign(\mathsf{ce}(v))$, but $\mathsf{Act}(u) \not\sim \mathsf{Act}(v)$. In that case, these two states should not be merged, since in any DROCA that is consistent with the sample, u and v must lead to two different states. This justifies the construction of Σ_{Act}. Now suppose that $sign(\mathsf{ce}(u)) \neq sign(\mathsf{ce}(v))$. In that case, those two states could very well be merged. This justifies the preprocessing steps in OPNI.

Since RPNI is guaranteed to terminate, together with Lemma 6, we conclude the following.

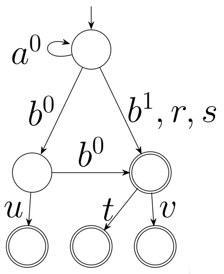
▶ **Theorem 8.** *Algorithm 2 terminates and returns a DROCA consistent with the input.*



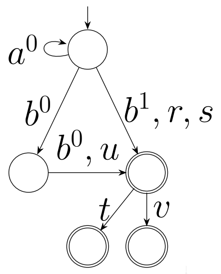
(a) Merge of $a^0 b^1, r$.



(b) Merge of $a^0 s, r$.



(c) Merge of $b^0 b^0, r$.



(d) Merge of $b^0 u, r$.

Figure 2 Intermediate steps of the OPNI algorithm on Example 7.

4 OCA-L*: OPNI-guided active learning procedure for OCA

In this section, we provide a procedure for active learning of DROCA using OPNI.

Learning framework. Following [11, 5], in an *active learning* framework for DROCA, the *learner*'s aim is to construct a DROCA $\mathcal{B}$ by interacting with a *teacher* that has a DROCA $\mathcal{A}$ in mind, such that $\mathcal{A}$ and $\mathcal{B}$ are equivalent. In the process, the learner can ask the teacher three types of queries:

- *Membership queries* $\mathsf{MQ}_{\mathcal{A}}$: the learner provides a word $w \in \Sigma^*$. The teacher returns 1 if $w \in \mathcal{L}(\mathcal{A})$, and 0 if $w \notin \mathcal{L}(\mathcal{A})$.
- *Counter value queries* $\mathsf{CV}_{\mathcal{A}}$: the learner asks the counter value reached on reading a word w. The teacher returns the counter value, *i.e.*, $\mathsf{ce}_{\mathcal{A}}(w)$.
- *Minimal synchronous-equivalence queries* $\mathsf{MSQ}_{\mathcal{A}}$: the learner asks whether a DROCA $\mathcal{C}$ is equivalent and counter-synchronous to $\mathcal{A}$. The teacher returns *yes* only if $\mathcal{C}$ and $\mathcal{A}$ are counter-synchronous and equivalent. Otherwise, the teacher provides a counter-example $z \in \Sigma^*$ with the smallest length such that either $\mathcal{C}(z) \neq \mathcal{A}(z)$ or $\mathsf{ce}_{\mathcal{A}}(z) \neq \mathsf{ce}_{\mathcal{C}}(z)$.

Note that the minimal synchronous-equivalence queries (introduced in [11]) may return a word for which the two machines reach different counter values. This query enables the active learning method to construct a DROCA that is counter-synchronous and equivalent to the teacher's DROCA. As noted in [11], synchronous equivalence is significantly faster than the impractical general equivalence check. Hence, relaxing this condition and having minimal equivalence queries will require the teacher to use a general equivalence check. Thus, in our implementation (described in Section 5), we opt for equivalence queries on minimal counter-synchronous DROCA, as they are faster.

Learning of DROCA

We introduce an active learning procedure for DROCA, called OCA-L*. The pseudocode is presented in Procedure 3, and its key components are detailed below. OCA-L* builds on Angluin's L^* algorithm with modifications inspired by the MinOCA framework from [11].

Observation table. Consider an input alphabet $\Sigma = \{\sigma_1, \dots, \sigma_k\}$ for some $k \in \mathbb{N}$. The learner maintains an observation table $T = (\mathsf{R}, \mathsf{C}, \mathit{Memb}, \mathsf{ce} \upharpoonright_{\mathsf{R} \cup \mathsf{R}\Sigma}, \mathsf{Act})$ over Σ. Here R is a nonempty prefix-closed set of strings, C is a nonempty suffix-closed set of strings, $\mathit{Memb} : (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C} \to \{0,1\}$ is a function that indicates whether a word belongs to the language, $\mathsf{ce} \upharpoonright_{\mathsf{R} \cup \mathsf{R}\Sigma}$: $\mathsf{R} \cup \mathsf{R}\Sigma \to \mathbb{N}$ is the function ce with domain restricted to the set $\mathsf{R} \cup \mathsf{R}\Sigma$, and $\mathsf{Act} : (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C} \to \{0,1\} \times \{0,1,-1\}^k$ is a function representing the sign of the counter value reached and the counter-actions on every letter after reading a word. The function Act is the same as the one defined in Section 3. Given $w \in (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C}$, the function $\mathit{Memb}(w)$ returns 0 (resp., 1) if $\mathcal{A}(w)$ is equal to 0 (resp., 1). The observation table initially has $\mathsf{R} = \mathsf{C} = \{\epsilon\}$ and is updated as the procedure runs. Let $\mathsf{C} = \{c_1, \dots, c_\ell\}$ for some $\ell \in \mathbb{N}$. For any $r \in \mathsf{R} \cup \mathsf{R}\Sigma$, we use $\mathit{row}(r)$ to denote the tuple $(\mathsf{ce}(r), (\mathit{Memb}(rc_1), \mathsf{Act}(rc_1)), \dots, (\mathit{Memb}(rc_\ell), \mathsf{Act}(rc_\ell)))$.

▶ **Definition 9** (Definition 5 in [11]). *Let $d \in \mathbb{N}$ and $T = (\mathsf{R}, \mathsf{C}, \mathit{Memb}, \mathsf{ce} \upharpoonright_{\mathsf{R} \cup \mathsf{R}\Sigma}, \mathsf{Act})$ be an observation table.*

1. *T is said to be d-*closed* if for all $r' \in \mathsf{R}\Sigma$ with $\mathsf{ce}(r') \le d$ there exists $r \in \mathsf{R}$ such that $\mathit{row}(r) = \mathit{row}(r')$.*
2. *T is said to be d-*consistent* if for all $r, s \in \mathsf{R}$, such that $\mathsf{ce}(r) = \mathsf{ce}(s) \le d$ and $\mathit{row}(r) = \mathit{row}(s)$ implies that for all $\sigma \in \Sigma$, $\mathit{row}(r\sigma) = \mathit{row}(s\sigma)$.*

As described in Procedure 3 (see also [11]), the observation table is iteratively expanded until it satisfies the conditions of d-closure and d-consistency. The set of words $w \in (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C}$ is partitioned into S_+ and S_-, based on whether each word is accepted or rejected by the teacher. The goal is to construct a DROCA that solves the corresponding passive learning task: separating S_+ from S_- while being counter-synchronous with the teacher's DROCA for all prefixes of words in $\mathsf{S}_+ \cup \mathsf{S}_-$. This task is delegated to the passive learning algorithm OPNI, which returns a DROCA consistent with the observed data. A DROCA and an observation table corresponding to it are given in Figure 5 and Figure 6 respectively (see Appendix A).

The learner then queries the teacher to verify whether the hypothesis DROCA accepts the same language as the target. If not, the teacher provides a counterexample, which is used to update the observation table. This process repeats until the learner synthesises a DROCA that is equivalent to the teacher's.

It is instructive to contrast our approach with the MinOCA algorithm [11]. While MinOCA uses a SAT solver to solve the passive learning problem and guarantees a minimal separating DROCA, the use of OPNI does not provide any bound on the size of the returned automaton. As a result, OPNI may produce increasingly large automata, and the overall learning procedure OCA-L* may, in principle, fail to terminate. On the other hand, reliance on SAT solvers limits the scalability of MinOCA[2]. In the next section, we show that our approach, based on OPNI, significantly improves scalability compared to MinOCA.

[2] This approach uses the state-of-the-art tool DFAMiner [6] which in turn uses a SAT solver to find the minimal separating DFA.

Procedure 3 OCA-L* for DROCA.

```
Input: Access to membership ($\mathsf{MQ}_{\mathcal{A}}$), counter value ($\mathsf{CV}_{\mathcal{A}}$), and equivalence
       ($\mathsf{MSQ}_{\mathcal{A}}$) queries.
Output: A DROCA $\mathcal{B}$ accepting the same language as $\mathcal{A}$.
1  Initialise $\mathsf{R} \leftarrow \{\varepsilon\}$, $\mathsf{C} \leftarrow \{\varepsilon\}$, $d \leftarrow 0$.
2  Initialise the observation table $T = (\mathsf{R}, \mathsf{C}, \mathit{Memb}, \mathsf{ce}\restriction_{\mathsf{R}\cup\mathsf{R}\Sigma}, \mathsf{Act})$.
3  repeat
4      while $T$ is not $d$-closed or not $d$-consistent do
5          if $T$ is not $d$-closed then
6              Find $r \in \mathsf{R}$, $a \in \Sigma$ such that $\mathsf{ce}(ra) \leq d$, $\mathit{row}(ra) \neq \mathit{row}(r')$ for all $r' \in \mathsf{R}$.
7              Add $ra$ to $\mathsf{R}$.
8          if $T$ is not $d$-consistent then
9              Find $r, s \in \mathsf{R}$, $a \in \Sigma$, $c \in \mathsf{C}$ such that $\mathsf{ce}(r) = \mathsf{ce}(s) \leq d$, $\mathit{row}(r) = \mathit{row}(s)$,
                and $(\mathit{Memb}(rac) \neq \mathit{Memb}(sac)$ or $\mathsf{Act}(rac) \neq \mathsf{Act}(sac))$.
10             Add $ac$ to $\mathsf{C}$.
11         Extend $\mathit{Memb}$ and $\mathsf{Act}$ to $(\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C}$, using membership and counter value
            queries.
12     $\mathsf{S}_+ = \{w \in (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C} \mid \mathit{Memb}(w) = 1\}$.
13     $\mathsf{S}_- = \{w \in (\mathsf{R} \cup \mathsf{R}\Sigma)\mathsf{C} \mid \mathit{Memb}(w) = 0\}$.
14     $\mathcal{B} \leftarrow \mathsf{OPNI}(\mathsf{S}_+ \cup \mathsf{S}_-, \mathsf{ce}\restriction_{(\mathsf{R}\cup\mathsf{R}\Sigma)\mathsf{C}}; \Sigma)$.
15     Ask equivalence query $\mathsf{MSQ}_{\mathcal{A}}(\mathcal{B})$.
16     if teacher gives counter-example $z$ then
17         Add all prefixes of $z$ to $\mathsf{R}$.
18         Update $d \leftarrow \max\{\max(d, \mathsf{ce}_{\mathcal{A}}(z')) \mid z'$ is a prefix of $z\}$.
19 until teacher replies yes to an equivalence query;
20 return $\mathcal{B}$.
```

Learning of VOCA

We customise the OCA-L* method for learning VOCA as well. In this setting, the counter-actions are determined by the input alphabet and can therefore be inferred directly from the word. Consequently, the function Act does not have to be included in the observation table. This simplifies the learning process, as the sets S_+ and S_- constructed by OCA-L* are considerably smaller. Furthermore, there are words that do not have a valid run and are denoted by x in the observation table. The notion of d-closed and d-consistent has to be modified such that an x-marked cell does not have to be compared with a cell in the same column. Moreover, the equivalence check for VOCA is more efficient than for DROCA. A VOCA and an observation table corresponding to it are given in Figure 7 and Figure 8 respectively (see Appendix A).

5 Experimental evaluation

Bruyère et al. [5], were the first to provide a tool for learning DROCA. They employ counter value queries in addition to standard membership and equivalence queries. While effective in principle, this approach does not scale well as the size of the DROCA being learnt increases beyond 7 states.

To address these limitations, Mathew et al. [11] introduced MinOCA, a learning tool capable of inferring DROCA with at most 15 states using a SAT-based approach. Although MinOCA improves upon earlier implementations, its reliance on SAT solvers leads to a significant increase in computational overhead as the size of the automaton grows. In practice, this results in a bottleneck that makes learning larger MinOCA infeasible.

In this work, we present a new approach – OCA-L*– which replaces the SAT-based inference in MinOCA with OPNI (see Section 3). This scales effectively with input size and demonstrates the ability to learn larger DROCA, significantly outperforming existing tools.

We implemented OCA-L* in Python and tested it on randomly generated DROCA. We also implemented a faster learning algorithm for the special case of VOCA. The implementation details and the results obtained are discussed in this section.

Equivalence query. The equivalence of DROCA is known to be in polynomial time [4]. However, as pointed out in [11, 5], the polynomials involved are not suitable for practical applications. Mathew et al. [11] give a practical algorithm for checking the equivalence of VOCA and counter-synchronous DROCA that returns the minimal counter-example. We use their ideas in checking the equivalence of the learnt automaton in our implementation.

Generation of random benchmarks. We generated two datasets for evaluating the performance of the proposed method.

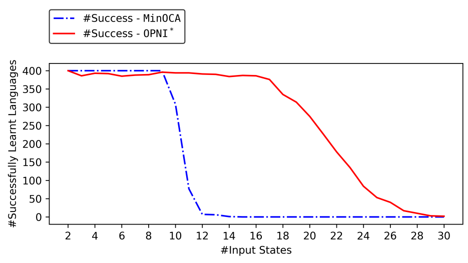
- $\texttt{Dataset}_1$: Random DROCA to compare the performance of OCA-L* with MinOCA.
- $\texttt{Dataset}_2$: Random VOCA to evaluate the performance of OCA-L* for learning VOCA and a similar adaptation of the MinOCA algorithm.

1. Generating random DROCA ($\texttt{Dataset}_1$*).* In [11], experiments were carried out on randomly generated DROCA. We follow the same procedure to compare our experimental results with those of [11].

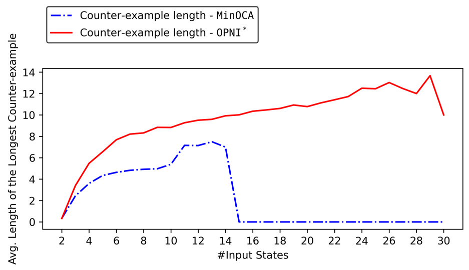
The procedure is explained here for the sake of completeness. Let $n \in \mathbb{N}$ be the number of states of the DROCA to be generated. First, we initialise the set of states $Q = \{q_1, q_2, \ldots, q_n\}$. For all $q \in Q$, we add q to the set of final states F with probability 0.5. If $Q = F$ or $F = \emptyset$ after this step, then we restart the procedure. Otherwise, for all $q \in Q$ and $a \in \Sigma$, we assign $\delta_0(q, a) = (p, c)$ (resp. $\delta_1(q, a) = (p, c)$), with p a random state in Q and c a random counter operation in $\{0, +1\}$ (resp. $\{0, +1, -1\}$). The constructed DROCA is $\mathcal{A} = (Q, \Sigma, \{q_1\}, \delta_0, \delta_1, F)$. If the number of reachable states of $\mathcal{A}$ from the initial configuration is not n, then we discard $\mathcal{A}$ and restart the whole procedure. Otherwise, we output $\mathcal{A}$.

We generated random DROCA with number of states ranging from 2 to 30 and alphabet size ranging from 2 to 5. A total of 100 DROCA were generated for every pair of alphabet and number of states. We reused the dataset used in [8, 11] for DROCA up to 15 states.

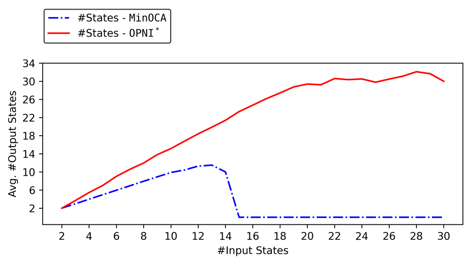
2. Generating random VOCA ($\texttt{Dataset}_2$*).* For the case of VOCA, we use a similar procedure for generating random VOCA. We generate the states in the same way as above. Following this, for all $a \in \Sigma$, we randomly puts it in one of the sets $\Sigma_{call}, \Sigma_{ret}$, or Σ_{init}. The elements in $\Sigma_{call}, \Sigma_{ret}$ or Σ_{init}, will respectively have $+1, -1$ and 0 as their corresponding counter action. If there does not exist at least one symbol in Σ_{call} and one in Σ_{ret}, then we restart the whole procedure, since the language recognised will be regular. The transitions are also generated as in the case of DROCA. Note that the counter action c in a transition depends on whether the symbol a is in $\Sigma_{call}, \Sigma_{ret}$, or Σ_{init}. The constructed VOCA is $\mathcal{A} = (Q, \Sigma_{call} \cup \Sigma_{ret} \cup \Sigma_{init}, \{q_1\}, \delta_0, \delta_1, F)$. If the number of reachable states of $\mathcal{A}$ from the initial configuration is not n, then we discard $\mathcal{A}$ and restart the whole procedure. Otherwise, we output $\mathcal{A}$.



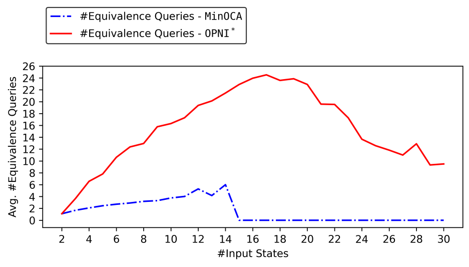
(a) Number of successfully learnt languages by OCA-L* (Out of 400) with 5 minutes timeout.



(b) The average length of the longest counter-example.



(c) Average number of states in the learnt DROCA.



(d) Average number of equivalence queries used.

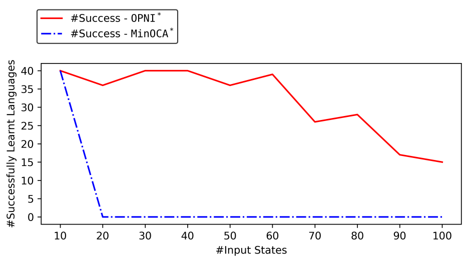
Figure 3 Evaluation of OCA-L* and MinOCA for DROCA on Dataset$_1$.

We generated random VOCA with number of states ranging from 10 to 100 and input alphabet size ranging from 2 to 5. A total of 10 VOCA were generated for every pair of input alphabet and number of states. The number of randomly generated VOCA was limited to 10 per configuration due to the increasing difficulty in constructing a valid automaton where all states are reachable and satisfy the remaining conditions. Moreover, our primary objective is to evaluate how well the proposed approach scales with input size. For this purpose, the selected sample size is sufficient to observe meaningful trends in performance without incurring excessive computational overhead.

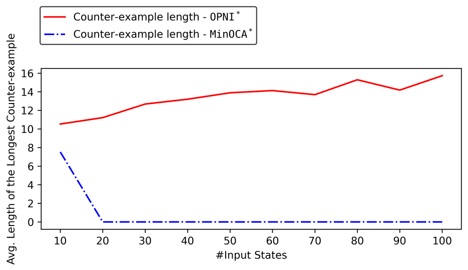
Experimental results. All the experiments were performed on an Apple M1 chip with 8GB of RAM. We implemented the proposed method (OCA-L*) in Python for DROCA and then for the special case of VOCA[3] In Figure 3a (resp. Figure 4a), the total number of languages is out of 400 (resp. 40) for each input size because, for every number of states, we generated 400 (resp. 40) random DROCA (resp. VOCA) for input alphabet sizes 2, 3, 4, and 5; to keep the visualization simple, we omitted the z-axis for alphabet size and instead aggregated the number of successfully learnt languages across all alphabet sizes for each input size – this approach is used consistently across all graphs.

1. *Evaluating the performance OCA-L* and MinOCA on* Dataset$_1$: We first compare the performance of OCA-L* for DROCA with that of MinOCA [11]. A timeout of 5 minutes was allotted for both MinOCA and OCA-L* for learning each DROCA. If the procedure times out,

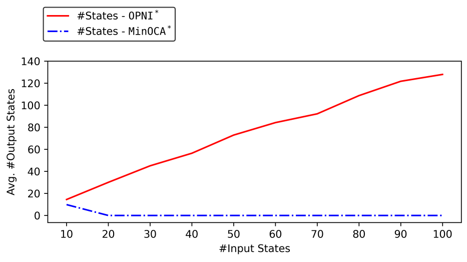
[3] The implementations of OCA-L* for DROCA, OCA-L* and MinOCA for VOCA, the datasets used, and the complete results generated are available at [9]. The implementation of MinOCA for DROCA is provided in [8].



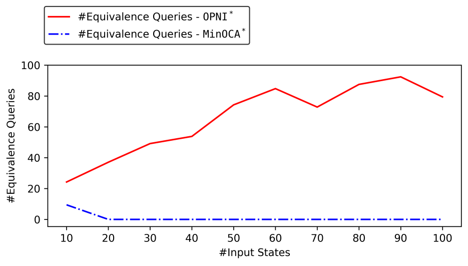
(a) Number of successfully learnt languages (Out of 40) with a timeout of 20 minutes.



(b) The average length of the longest counter-example.



(c) Average number of states in the learnt VOCA.



(d) Number of equivalence queries used.

Figure 4 Evaluation of OCA-L* and MinOCA for VOCA on $\texttt{Dataset}_2$.

we discard that input and process the next one. The number of languages successfully learnt by OCA-L* and MinOCA for different input sizes is depicted in Figure 3a. The proposed method outperforms MinOCA in terms of the number of successfully learnt languages within the given timeout as the number of states increases. The averages presented in the remaining graphs are computed using only those languages that were successfully learnt. Note that the y-values for MinOCA remain zero in all graphs for input sizes with more than 15 states, as it fails to learn any language within the specified timeout.

Figure 3b shows the average length of the longest counter-example. Since MinOCA guarantees learning a minimal counter-synchronous DROCA, whereas OCA-L* provides no such guarantee, the DROCA learnt by OCA-L* typically has more states. Consequently, the length of the counterexamples also tends to be longer for OCA-L*. Figure 3c shows the average number of states in the learnt DROCA. MinOCA learns a minimal counter-synchronous DROCA equivalent to the input. However, the DROCA learnt by OPNI is equivalent and counter-synchronous with respect to the input, but not minimal. Figure 3d shows the average number of equivalence queries used for successfully learning the input DROCA. We also add in Appendix B Figure 9a and Figure 9b the average number of rows and columns in the final observation table.

Note that OCA-L* is a semi-algorithm; its termination is not guaranteed. However, our experimental results on randomly generated benchmarks show that it works well in practice.

2. Evaluating the performance OCA-L* and MinOCA on $\texttt{Dataset}_2$: We now compare the performance of OCA-L* for VOCA with that of MinOCA [11]. We note that OCA-L* for VOCA is faster compared to OCA-L* for DROCA. This is mainly due to the following two reasons:

1. faster algorithm for checking equivalence of VOCA, and
2. the input alphabet itself determines the counter actions, and therefore, step 2 and step 3 of Algorithm 2 can be skipped while passive learning VOCA.

A similar modification of MinOCA for VOCA was also implemented. A timeout of 20 minutes was allotted for both procedures for learning each VOCA. If the procedure times out, we discard that input and process the next one. The number of VOCA languages successfully learnt by OCA-L* for different input sizes is shown in Figure 4a. We observe that OCA-L* is able to learn close to 50% of VOCA of 100 states. However, it was not able to learn any VOCA with more than 20 states. The averages presented in the remaining graphs are computed using only those languages that were successfully learnt. Note that the y-values for MinOCA remain zero in all graphs for input sizes with more than 20 states, as it fails to learn any language within the specified timeout.

The results of our experiments are shown in Figure 4. Figure 4b shows the average length of the longest counter-example. Figure 4c shows the average number of states in the learnt VOCA. Figure 4d shows the average number of equivalence queries used for successfully learning the input VOCA. In this case also, the VOCA learnt by MinOCA is minimal, whereas the one learnt by OCA-L* is not. Figure 10a and Figure 10b (see Appendix B) show the average number of rows and columns in the final observation table.

6 Conclusion

This work focuses on passive and active learning of deterministic real-time one-counter automata (DROCA). Inspired by the classical RPNI algorithm, we developed a passive learning algorithm, OPNI, tailored to DROCA. Building on this, we showed how active learning can be guided by a sequence of passive learning tasks, and proposed an active learning method, OCA-L*, which uses OPNI as a subroutine. In contrast, the state-of-the-art MinOCA algorithm employs a SAT solver for this step.

Our experiments demonstrate that OCA-L* scales significantly better than MinOCA in practice. We also used OCA-L* to learn visibly one-counter automata (VOCA), observing that it can successfully learn automata with up to 100 states. Despite its practical advantages, a limitation of OCA-L* is that it does not guarantee termination on all inputs, since OPNI may return increasingly large automata in the absence of minimality guarantees.

A fundamental bottleneck – common to our approach and to prior works such as [5, 11] – is the reliance on counter-value queries. This renders the learning process a grey-box framework, in contrast to Angluin's black-box L^* algorithm for finite automata. Recent work [10] has shown that active learning of DROCA is possible in polynomial time even without counter-value queries. However, the proposed algorithm is not practical due to the high-degree polynomials involved. An interesting direction for future work is to combine insights from the methods: OCA-L*, MinOCA [11], and OL* [10] to develop learning algorithms that are both theoretically efficient and practically usable.

Finally, extending these ideas to richer models such as pushdown automata – or their subclasses, such as visibly pushdown automata – offers a direction for future research.

References

1 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211. ACM, 2004. doi:10.1145/1007352.1007390.

2 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009. doi:10.1145/1516512.1516518.

3 Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi:10.1016/0890-5401(87)90052-6.

4 Stanislav Böhm and Stefan Göller. Language equivalence of deterministic real-time one-counter automata is nl-complete. In *MFCS*, volume 6907 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2011. doi:10.1007/978-3-642-22993-0_20.

5 Véronique Bruyère, Guillermo A. Pérez, and Gaëtan Staquet. Learning realtime one-counter automata. In *TACAS (1)*, volume 13243 of *Lecture Notes in Computer Science*, pages 244–262. Springer, 2022. doi:10.1007/978-3-030-99524-9_13.

6 Daniele Dell'Erba, Yong Li, and Sven Schewe. DFAMiner: Mining minimal separating dfas from labelled samples. In *FM (2)*, volume 14934 of *Lecture Notes in Computer Science*, pages 48–66. Springer, 2024. doi:10.1007/978-3-031-71177-0_4.

7 Amr F. Fahmy and Robert S. Roos. Efficient learning of real time one-counter automata. In *ALT*, volume 997 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 1995. doi:10.1007/3-540-60454-5_26.

8 Prince Mathew. MinOCA, January 2025. doi:10.5281/zenodo.14604419.

9 Prince Mathew. OCA-L*, October 2025. doi:10.5281/zenodo.17310292.

10 Prince Mathew, Vincent Penelle, and A. V. Sreejith. Learning deterministic one-counter automata in polynomial time. In *LICS*, pages 444–457, 2025.

11 Prince Mathew, Vincent Penelle, and A. V. Sreejith. Learning real-time one-counter automata using polynomially many queries. In *TACAS (1)*, volume 15696 of *Lecture Notes in Computer Science*, pages 276–294. Springer, 2025. doi:10.1007/978-3-031-90643-5_14.

12 Daniel Neider and Christof Löding. Learning visibly one-counter automata in polynomial time. *Technical Report, RWTH Aachen*, AIB-2010-02, 2010.

13 José Oncina and Pedro Garcia. Inferring regular languages in polynomial updated time. In *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pages 49–61. World Scientific, 1992.

14 Leslie G. Valiant and Mike Paterson. Deterministic one-counter automata. *J. Comput. Syst. Sci.*, 10(3):340–350, 1975. doi:10.1016/S0022-0000(75)80005-5.

A Section 4. OCA-L*: OPNI-guided active learning procedure for OCA

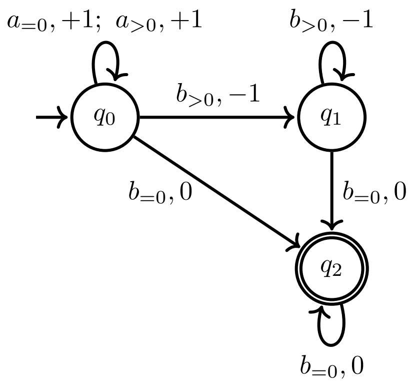


Figure 5 A DROCA recognising the language $\{a^n b^m \mid m > n\}$. Transitions not shown go to a non-final sink state with counter-action 0.

		ce	ϵ	
			Memb	Act
R	ϵ	0	0	$(0, +1, 0)$
	a	1	0	$(1, +1, -1)$
	b	0	1	$(0, 0, 0)$
	ab	0	0	$(0, 0, 0)$
RΣ	aa	2	0	$(1, +1, -1)$
	ba	0	0	$(0, 0, 0)$
	bb	0	1	$(0, 0, 0)$
	aba	0	0	$(0, 0, 0)$
	abb	0	1	$(0, 0, 0)$

Figure 6 Observation table for the DROCA in Figure 5. The table is 1-closed and 1-consistent. Here, $\mathsf{R} = \{\epsilon, a, b\}$ and $\mathsf{C} = \{\epsilon\}$.

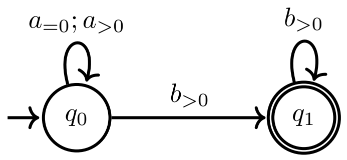
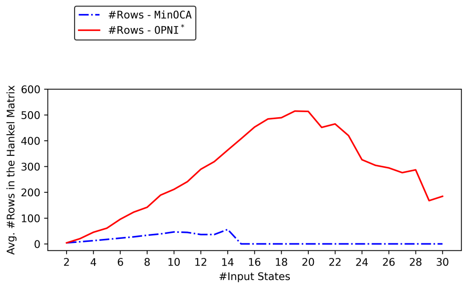


Figure 7 A VOCA over the alphabet $\{a, b\}$, where $\Sigma_{call} = \{a\}$, $\Sigma_{ret} = \{b\}$ and $\Sigma_{int} = \emptyset$. The VOCA recognises the language $\{a^n b^m \mid m \leq n\}$. The transitions not shown go to a non-final sink state.

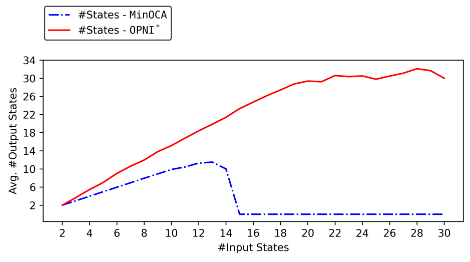
		ce	ϵ
			Memb
R	ϵ	0	0
	a	1	0
	ab	0	1
RΣ	b	x	x
	aa	2	0
	aba	1	0
	abb	x	x

Figure 8 Observation table for the VOCA in Figure 7. The table is both 1-closed and 1-consistent. Here, $\mathsf{R} = \{\epsilon, a, ab\}$ and $\mathsf{C} = \{\epsilon\}$.

B Section 5. Experimental evaluation

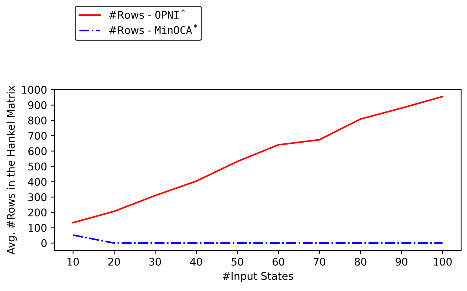


(a) The average number of rows in the observation table.

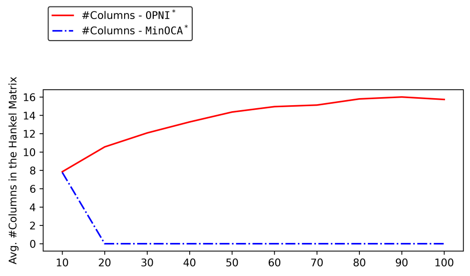


(b) The average number of columns in the observation table.

Figure 9 Evaluation of OCA-L* and MinOCA for DROCA on $\texttt{Dataset}_1$.



(a) The average number of rows in the observation table.



(b) The average number of columns in the observation table.

Figure 10 Evaluation of OCA-L* and MinOCA for VOCA on $\texttt{Dataset}_2$.