ALMA: Automata Learner using Modulo 2 Multiplicity Automata

Nevin George

Yale University, New Haven CT 06511, USA
nevin.george@yale.edu

Abstract. We present ALMA (Automata Learner using modulo 2 Multiplicity Automata), a Java-based tool that can learn any automaton accepting regular languages of finite or infinite words with an implementable membership query function. Users can either pass as input their own membership query function, or use the predefined membership query functions for modulo 2 multiplicity automata (M2MAs) and nondeterministic Büchi automata. While learning, ALMA can output the state of the observation table after every equivalence query, and upon termination, it can output the dimension, transition matrices, and final vector of the learned automaton. Users can test whether a word is accepted by performing a membership query on the learned automaton.

ALMA follows the polynomial-time learning algorithm of Beimel et al. (Learning functions represented as multiplicity automata. J. ACM 47(3), 2000). ALMA also implements a polynomial-time learning algorithm for strongly unambiguous Büchi automata by Angluin et al. (Strongly unambiguous Büchi automata are polynomially predictable with membership queries. CSL 2020), and a minimization algorithm for M2MAs by Sakarovitch (Elements of Automata Theory. 2009).

ALMA is unique from other similar tools in that hypotheses during the learning algorithm and the outputted learned automata are represented using M2MAs. The tool enables researchers to explore the practical and theoretical advantages of M2MAs, a relatively unexplored automaton representation which we argue is useful in the verification community for representing regular ω-languages.

Keywords: automata theory · finite automata · Büchi automata · multiplicity automata · learning

1 Introduction

Angluin's exact learning model [1] has been studied extensively in the context of learning theory, and it can be used to learn automata representing regular languages of finite and infinite words. In the model, a learner interacts with an oracle to learn a regular language using membership and equivalence queries. In a membership query, the learner learns from the oracle whether a word is in

the language. In an equivalence query, the learner forms a hypothesis on what the language is, and the oracle either confirms that the hypothesis is correct or returns a counterexample, i.e, a word for which the hypothesis and language differ.

As one application of the exact learning model, Beimel et al. [4] detail a polynomial-time algorithm to learn multiplicity automata using membership and equivalence queries. The algorithm can be generalized by replacing the membership query function for multiplicity automata with that of any other automaton accepting regular languages of finite or infinite words. We provide a high-level overview of the algorithm: the algorithm begins with a trivial hypothesis of the language, represented as a multiplicity automaton of dimension 1 defined over some field $\mathcal{K}$. On each iteration of a loop, the algorithm performs an equivalence query. If the hypothesis is equivalent to the language, the algorithm terminates and outputs the hypothesis. Otherwise, the algorithm receives a counterexample from the oracle, which is used to improve the hypothesis. The algorithm terminates after d iterations, where d is the dimension of the smallest possible multiplicity automaton that can represent the language.

Strongly unambiguous Büchi automata (SUBAs) (defined in Section 2.1) are a type of non-deterministic Büchi automaton (NBA) first introduced by Bosquet and Löding [6]. SUBAs are useful for modeling reactive systems, as they are fully expressive, i.e., they can represent any regular ω-language, and can often represent regular ω-languages more succinctly than other NBA representations [2]. Angluin et al. [2] also showed that SUBAs are learnable in polynomial time, further increasing their importance.

In this paper, we present ALMA, a Java-based tool that implements the algorithm of Beimel et al. to learn any arbitrary automaton representing regular languages of finite or infinite words with an implementable membership query function. Hypotheses during the learning algorithm and outputted learned automata are represented using M2MAs. Membership query functions have been implemented for M2MAs and NBAs, and users can enter as input any arbitrary membership query function that 1) takes as input any possible word that can be formed from the given alphabet, and 2) outputs 0 or 1 to indicate whether the word is in the language.

To improve the runtime of learning SUBAs, ALMA does not use the general learning algorithm of Beimel et al., but rather the SUBA learning algorithm of Angluin et al. [2]. Also when learning M2MAs, ALMA first implements the algorithm of Sakarovitch [10] to minimize the M2MA before running the learning algorithm. This is because in the SUBA learning algorithm, converting the input SUBA into an equivalent M2MA incurs a quadratic increase size. Minimizing M2MAs before learning enables the algorithm to learn significantly larger SUBAs, especially because empirically the M2MAs have been seen to often minimize to M2MAs of much smaller dimension. The effect of the minimization algorithm on a sample of input SUBAs can be seen in Table 2 in Appendix B.

Usefulness/Novelty In the verification community, M2MAs can be useful for representing regular ω-languages, as they are learnable in polynomial time and often relatively succinct. They are also useful for tasks such as model checking, since performing operations such as intersection, union, complementation, emptiness, and equivalence on M2MAs is cheap [3]. Through the minimization and learning algorithms, ALMA enables researchers in the verification community to easily test and verify these properties of M2MAs, promoting further exploration into these useful automata. As an example, ALMA was used in the paper by Angluin et al. [3] to explore the suitableness of representing regular ω-languages using M2MAs. The authors used ALMA to convert SUBAs, NBAs, and DBAs into equivalent M2MAs, and they compared the succinctness of these M2MAs with that of DFAs accepting the same language. ALMA can similarly be used by other researchers to gain insights into M2MAs and their benefits/drawbacks as compared to other representations.

ALMA is the first publicly available implementation of the novel SUBA learning algorithm by Angluin et al. [2]. Using ALMA, users can run the algorithm to learn any regular ω-language. Since learned automata are represented as M2MAs, users can use the many desirable properties of M2MAs to gain insights into the initial SUBA and regular ω-language. In addition, since membership query functions are often relatively easy to implement and ALMA already provides a membership query function for the general NBA case, the scope of what ALMA can learn is large, promoting its usefulness in a wide variety of settings.

Many tools such as ROLL (ω-Regular Language Learning Library) [9] and libalf [5] already exist that can learn automata representing regular languages of finite and infinite words. However, ALMA is the first tool that uses M2MAs to represent hypotheses and the learned automaton in the learning algorithm. ALMA's usefulness lies not with necessarily being the fastest tool available to learn regular languages, but with exploring the practical benefits of M2MAs and the features of the algorithms by Beimel et al., Angluin et al., and Sakarovitch.

2 Useful Definitions

2.1 Finite and Büchi Automata

Let Σ be a finite alphabet. Then Σ^* and Σ^ω are the sets of all finite and infinite words, respectively, that can be formed using elements from Σ. A finite language is a subset of Σ^*, and an ω-language is a subset of Σ^ω. If w is a word in Σ^* or Σ^ω, let $|w|$ be the length of w and $w[i]$ be the i'th character of w.

A finite-state automaton A is represented as a tuple $(\Sigma, Q, I, \Delta, F)$, where Σ is the alphabet, Q is the finite set of states, $I \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times \Sigma \times Q$ is the set of transitions, and $F \subseteq Q$ is the set of final states. The automaton A is deterministic if every pair $(q, \sigma) \in Q \times \Sigma$ appears as the first two elements in at most one triple in Δ.

A run on A for a word w is a series of states $q_0, q_1, \ldots \in Q$ such that $\forall i$ satisfying $1 \leq i \leq |w|$, $(q_{i-1}, w[i], q_i) \in \Delta$. A run for a finite word is final if it

ends in a final state. For infinite words, a run is final if it passes infinitely often through a final state. A run is accepting if it is final and begins at an initial state. The automaton A accepts the finite/infinite word w if there exists an accepting run for w.

Non-deterministic finite automata (NFAs) and non-deterministic Büchi automata (NBAs) are automata accepting finite and infinite words, respectively. Deterministic finite automata (DFAs) are deterministic NFAs, and deterministic Büchi automata (DBAs) are deterministic NBAs. Unambiguous finite automata (UFAs) and unambiguous Büchi automata (UBAs) are NFAs and NBAs, respectively, for which every word has at most one accepting run. A UBA is a strongly unambiguous Büchi automaton (SUBA) if every word has at most one final run.

2.2 Modulo 2 Multiplicity Automata

Assume a field $\mathcal{K}$ and some dimension d. A multiplicity automaton A is represented as a tuple $(\Sigma, v_I, \{\mu_\sigma\}_{\sigma \in \Sigma}, v_F)$, where Σ is the alphabet, v_I is the initial vector, each μ_σ is a transition matrix, and v_F is the final vector. v_I and v_F have dimension $d \times 1$, and each μ_σ has dimension $d \times d$. The set of states is all row vectors $v \in \{0,1\}^d$, and the initial state is $v_I^\top$. For a given word $w = \sigma_1 \sigma_2 \ldots \sigma_n$, let $\mu(w) = \mu_{\sigma_1} \mu_{\sigma_2} \ldots \mu_{\sigma_n}$. The set of reachable states is all vectors of the form $v_I^\top \mu(w)$. Associated with the automaton A is a function $f_A : \Sigma^* \to \mathcal{K}$, where $\forall w \in \Sigma^*$,

$$f_A(w) = v_I^\top \mu(w) v_F.$$

A modulo 2 multiplicity automaton (M2MA) is a multiplicity automaton where $\mathcal{K} = \mathrm{GF}(2)$ and all calculations are done modulo 2. An M2MA A accepts a word $w \in \Sigma^*$ if and only if $f_A(w) = 1$.

As an example, consider the following M2MA M adapted from Angluin et al. [2].

$$M = (\{a, b\}, \begin{pmatrix} 1\ 0\ 0 \end{pmatrix}^\top, \{\mu_a, \mu_b\}, \begin{pmatrix} 1\ 1\ 0 \end{pmatrix}^\top)$$

where

$$\mu_a = \begin{pmatrix} 0\ 0\ 1 \\ 1\ 0\ 0 \\ 1\ 1\ 1 \end{pmatrix} \text{ and } \mu_b = \begin{pmatrix} 0\ 1\ 0 \\ 1\ 0\ 1 \\ 1\ 1\ 0 \end{pmatrix}.$$

M is equivalent to the DFA in Fig. 1. We explain Fig. 1: the computation begins at the initial state $\begin{pmatrix} 1\ 0\ 0 \end{pmatrix}$. On reading an a/b from the initial state, the DFA visits the states

$$\begin{pmatrix} 1\ 0\ 0 \end{pmatrix} \mu_a = \begin{pmatrix} 1\ 0\ 0 \end{pmatrix} \begin{pmatrix} 0\ 0\ 1 \\ 1\ 0\ 0 \\ 1\ 1\ 1 \end{pmatrix} = \begin{pmatrix} 0\ 0\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1\ 0\ 0 \end{pmatrix} \mu_b = \begin{pmatrix} 1\ 0\ 0 \end{pmatrix} \begin{pmatrix} 0\ 1\ 0 \\ 1\ 0\ 1 \\ 1\ 1\ 0 \end{pmatrix} = \begin{pmatrix} 0\ 1\ 0 \end{pmatrix}.$$

Other states are visited similarly by multiplying the current state vector by μ_a or μ_b. $(1\ 0\ 0)$, $(0\ 1\ 0)$, and $(1\ 0\ 1)$ are accepting states since

$$(1\ 0\ 0)\,(1\ 1\ 0)^{\top} = (0\ 1\ 0)\,(1\ 1\ 0)^{\top} = (1\ 0\ 1)\,(1\ 1\ 0)^{\top} = 1,$$

where $v_F = (1\ 1\ 0)^{\top}$.

Fig. 1. DFA for the M2MA M

M2MAs have many important properties described in Section 1 that make them useful in verification communities, e.g., learnable in polynomial time and cheap intersection, union, complementation, emptiness, and equivalence. For more information on M2MAs, we recommend reading the paper by Angluin et al. [3], which studies these properties of M2MAs extensively and performs a detailed analysis of M2MAs' ability to succinctly represent regular finite and ω-languages.

2.3 $L_\$$ Language

Since Büchi automata accept infinite words and M2MAs accept finite words, Büchi automata and M2MAs cannot accept words from the same language L. However, Büchi [7] showed that two regular ω-languages are equivalent if and only if they agree on a set of ultimately periodic words, i.e., words of the form $u(v)^{\omega}$ where u and v are finite words and v is non-empty. We then consider the language of finite words $L_\$ = \{u\$v \mid u(v)^{\omega} \in L\}$, which Calbrix et al. [8] showed is regular. If a Büchi automaton accepts an infinite language L, a finite automaton is said to also represent L if it accepts $L_\$$. The $L_\$$ language is used by Angluin et al. [2] in their SUBA learning algorithm in order to obtain a finite automaton equivalent to the initial SUBA, and the learned M2MA accepts words from $L_\$$.

3 Usage

3.1 Access and How to Run

ALMA is an open-source library freely available at the following GitHub repository: `https://github.com/nevingeorge/Learning-Automata`. The repository contains the executables, source files, and sample input files for ALMA, and the README document contains detailed information on how to use the executables and the required format for the input files.

ALMA is a Java-based tool designed to be used on the command line. To run for example the executable M2MA.jar, a user should enter the command java -jar M2MA.jar within a Terminal/Command Prompt window. This will start the program, and the program will then print instructions on how to input the desired input file and flags.

3.2 Executables

ALMA consists of five main executables: 1) M2MA.jar, 2) SUBA.jar, 3) minimize.jar, 4) NBA.jar, and 5) arbitrary.jar. We provide a basic overview of each of the jar files and their use cases. For each executable, the output is always the dimension, final vector, and transition matrices of the learned/minimized M2MA. Also after the algorithms terminate, each executable allows users to check whether a word is accepted by the outputted M2MA. Infinite words are represented using the $L_\$$ language explained in Section 2.3.

M2MA.jar is used to minimize and learn M2MAs. The alphabet, dimension, final vector, and transition matrices of the input M2MA must be specified (the initial vector is always the vector with all zeros except for a 1 in the first row).

SUBA.jar is used to learn SUBAs using the algorithm of Angluin et al. [2]. The alphabet, number of states, final states, and transitions of the input SUBA must be specified (the only initial state is the first state).

minimize.jar is used to minimize M2MAs using the algorithm of Sakarovitch [10]. The input is the same as that for M2MA.jar. minimize.jar can also be used to output the minimized M2MA equivalent to the input SUBA in the SUBA learning algorithm (i.e., it runs every step of the SUBA learning algorithm except for learning the final M2MA). In this case, the input is the same as that for SUBA.jar.

NBA.jar is used to learn NBAs. The alphabet, number of states, final states, and transitions of the input NBA must be specified (the only initial state is the first state). NBA.jar uses approximate equivalence queries (described in Section 4.5), which requires as input the number of tests to run, maximum length of a test, and maximum limit on the number of equivalence queries.

arbitrary.jar is used to learn arbitrary automata representing regular languages of finite and infinite words with an implementable membership query function. The membership query function must be defined in MQ.java. The only constraints on the function is that it must accept as input any possible word formed from letters in the alphabet, and it must output 0 or 1 depending on whether the

word is contained in the language. arbitrary.jar also uses approximate equivalence queries, which requires the same input parameters as described for NBA.jar.
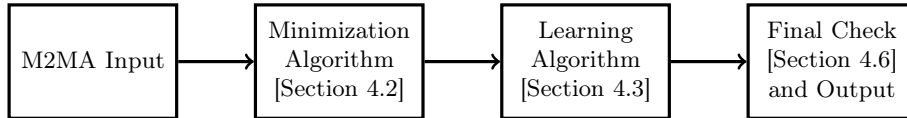
3.3 Flags

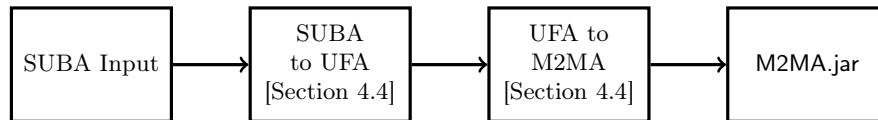Users can enter the following optional flags to add or remove information from the output of the algorithm.

-v: The algorithm outputs the state of the observation table (a matrix consisting of the words whose membership in the language is known) after every equivalence query in the learning algorithm.
-m: The algorithm outputs detailed information on the progress of the minimization algorithm. It gives status updates at different points in the algorithm, such as when it finishes creating the state/co-state spaces and the observation table. It also outputs the initial M2MA to be minimized and the final minimized observation table.
-d: When running minimize.jar on a SUBA, the algorithm outputs only the dimension of the minimized M2MA instead of the dimension, final vector, and transition matrices of the M2MA.
-a: After minimizing the M2MA, the algorithm outputs the number of states of the minimal deterministic finite automaton (DFA) that represents the same language as the M2MA.
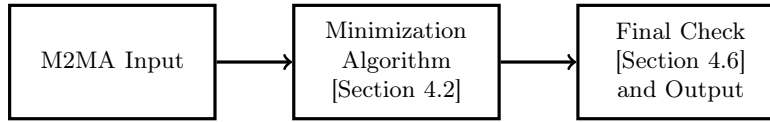
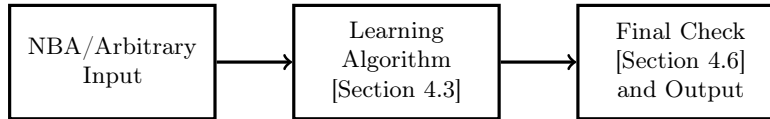## 4 Implementation Details

4.1 Architecture

| M2MA Input | → | Minimization Algorithm [Section 4.2] | → | Learning Algorithm [Section 4.3] | → | Final Check [Section 4.6] and Output |

Fig. 2. M2MA.jar Architecture

| SUBA Input | → | SUBA to UFA [Section 4.4] | → | UFA to M2MA [Section 4.4] | → | M2MA.jar |

Fig. 3. SUBA.jar Architecture

Fig. 4. minimize.jar Architecture

Fig. 5. NBA.jar and arbitrary.jar Architecture

Figures 2-5 provide a high-level overview of the architecture for each of the five executables. The learning algorithm of Beimel et al. [4], described in more detail in Section 4.3, is implemented in every executable except for minimize.jar. The M2MA minimization algorithm of Sakarovitch [10] (described in Section 4.2) is implemented in M2MA.jar, SUBA.jar, and minimize.jar. All five executables first take in as input from the command line the name of the input file and optional flags. The input is passed to a parser, which then reads the input and sends the parameters of the automaton to be learned/minimized to the algorithms. The executables also all run final checks (described in Section 4.6) on the outputted M2MA. Once the checks complete, users can test whether a word is accepted by the automaton.

4.2   M2MA Minimization Algorithm

The M2MA minimization algorithm, implemented in the MINIMIZE function of M2MA.java, is described in abstract terms in the work by Sakarovitch [10] and in more concrete detail in the appendix of [3]. The implemented minimization algorithm follows the pseudo-code in the appendix of [3] closely.

The algorithm requires a heavy use of linear algebra, much of which is done using the Apache Commons Math package. While testing the executables, however, many of the unminimized M2MAs were seen to be relatively sparse with few 1's in the transition matrices. To take advantage of the sparseness, ALMA implements custom linear algebra functions that use a sparse representation of matrices. Matrix rows are represented using arrays containing the locations of the 1's in the row. For example, the row $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$ is represented as $\begin{bmatrix} 2 & 6 \end{bmatrix}$, indicating that the row has a 1 in columns 2 and 6. Matrix multiplication, dot products, and tests for linear independence are implemented using the sparse matrix representation. Since the minimization function deals with M2MAs, the custom linear algebra functions perform all calculations modulo 2, which further improves the runtime.

4.3 General Learning Algorithm

The learning algorithm of Beimel et al. [4] is the core algorithm underpinning ALMA, and it is implemented in the LEARN function of M2MA.java. We provide a high-level overview of the algorithm in Section 1, and more details on the algorithm are found in [4]. Like the minimization algorithm, the linear algebra is implemented using a combination of the Apache Commons Math package and custom functions using the sparse representation for matrices.

The default membership query function the learning algorithm uses is that for M2MAs. As described in Section 2.2, an M2MA $A = (\Sigma, v_I, \{\mu_\sigma\}_{\sigma \in \Sigma}, v_F)$ accepts a word w if $v_I^\top \mu(w) v_F = 1$. Equivalence queries for M2MAs are easy, since as a by-product of running the minimization algorithm we get a complete observation table for the automaton. The equivalence query function checks whether the current hypothesis agrees with the M2MA being learned on every word in the observation table, as well as all possible one-letter extensions of the words. If they agree on every word and one-letter extension, the algorithm terminates; otherwise, the algorithm returns a word for which they disagree on as a counterexample.

4.4 SUBA Learning Algorithm

The SUBA learning algorithm of Angluin et al. [2] works as follows: the input SUBA is first converted into an equivalent UFA using a simple construction by Bosquet and Löding [6]. If the SUBA has n states, the constructed UFA has size $2n^2 + n$. Next, the UFA is converted to an equivalent M2MA of the same size. Lastly, the M2MA is learned using the algorithm of Beimel et al. [4]. The algorithms for the SUBA to UFA and UFA to M2MA conversions are found in SUBA.java, and the outputted M2MA is sent to M2MA.java to be minimized and learned.

4.5 Learning NBA and Arbitrary Automata

In M2MA.jar, minimize.jar, and SUBA.jar, the learning algorithm runs membership and equivalence queries on M2MAs. The learning algorithm for NBA.jar, however, uses a membership query function designed specifically for NBAs which we detail in Appendix A. arbitrary.jar passes to the learning algorithm a membership query function specified by the user from MQ.java. Also since the equivalence query function for M2MAs doesn't work in the general setting, NBA.jar and arbitrary.jar use approximate equivalence queries that rely on testing a random sample of words from the language.

Approximate Equivalence Queries The approximate equivalence query function is defined in arbitrary.java. Along with the hypothesis, it requires two parameters n and l as input. The function generates n random words of length at most l, and it tests whether the hypothesis and automaton being learned agrees on

these words. Increasing n improves the accuracy of the function at the expense of the runtime. In the input file, users also give a limit m on the number of equivalence queries that can be run. Since there are no guarantees on the size of the learned M2MA, the parameter m prevents the algorithm from running for an arbitrarily long length of time.

4.6   Checks

The code performs checks at various points of the algorithm. If a check fails, the code throws an exception and terminates the program. Example checks include checking the validity of the input (e.g., correct number of transition matrices, matrices only contain 0's and 1's, etc.), whether a matrix is invertible before running a linear equation solver, and if the dimension of the minimized M2MA equals that of the final learned M2MA.

After learning an M2MA, the code checks whether the outputted M2MA agrees with the input automaton on the membership of a random sample of words. By default, the code generates 1000 random words of length at most 25, but these constants can be modified. To perform this check, the code uses the previously described membership query functions for M2MAs, NBAs, and arbitrary automata, as well as a membership query function for SUBAs described in a paper by Bosquet and Löding [6]. Users can also run membership queries on the outputted M2MA to manually confirm whether the M2MA accepts a given word.

5   Conclusion

In Appendix B, we perform an experimental evaluation of ALMA and analyze its practical capabilities. ALMA has limitations - for example, the runtime of M2MA.jar becomes impractical for random M2MAs of size much larger than 100, and ALMA doesn't implement an exact equivalence query function for learning NBAs and arbitrary automata. However, ALMA is highly efficient for most standard use cases, and it can be used to promote further research into M2MAs and their properties. For example, in the paper by Angluin et al. [3], ALMA is used to find the dimension of the minimum M2MA that can represent a regular ω-language. Angluin et al. compare this dimension with the size of other finite automata that represent the same language to analyze the succinctness of the M2MA representation. Along with serving as a useful tool for investigating M2MAs, ALMA confirms the theoretical results of Beimel et al. [4] and Angluin et. al [2,3], and is the first publicly available tool that can be used to explore these learning algorithms.

References

1. Angluin, D.: Queries and concept learning. Machine Learning **2**(4), 319–342 (1987)
2. Angluin, D., Antonopoulos, T., Fisman, D.: Strongly unambiguous Büchi automata are polynomially predictable with membership queries. In: 28th EACSL Annual Conference on Computer Science Logic, CSL. pp. 8:1–8:17 (2020)
3. Angluin, D., Antonopoulos, T., Fisman, D., George, N.: Representing regular languages of infinite words using mod 2 multiplicity automata. In: Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS. p. 1–20 (2022)
4. Beimel, A., Bergadano, F., Bshouty, N.H., Kushilevitz, E., Varricchio, S.: Learning functions represented as multiplicity automata. J. ACM **47**(3), 506–530 (May 2000)
5. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: `libalf`: The automata learning framework. In: Computer Aided Verification - 22nd International Conference, CAV. pp. 360–364 (2010)
6. Bousquet, N., Löding, C.: Equivalence and inclusion problem for strongly unambiguous Büchi automata. In: Language and Automata Theory and Applications, 4th International Conference, LATA. Proceedings. pp. 118–129 (2010). https://doi.org/10.1007/978-3-642-13089-2_10
7. Büchi, J.: On a decision method in restricted second order arithmetic. In: International Congress on Logic, Methodology and Philosophy. pp. 1–11. Stanford Univ. Press (1962)
8. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational w-languages. In: Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics. pp. 554–566. Springer-Verlag (1994)
9. Li, Y., Sun, X., Turrini, A., Chen, Y.F., Xu, J.: Roll 1.0: ω-regular language learning library. In: Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS, Part I. pp. 365–371 (2019)
10. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, USA (2009)

A Appendix

NBA Membership Query Function The NBA membership query function is described in Algorithm 1. As explained in Section 2.1, an NBA accepts a word if there exists a path for the word that begins at an initial state and passes infinitely often through a final state. The input parameters u, v represent a word $w = u\$v$ in the $L_\$$ language.

Algorithm 1. NBA Membership Query Function

```
 1: function MAIN(u, v)
 2:     S_u ← states reachable from the initial state on reading u
 3:     S_uv ← S_u ∪ REACHABLE(S_u, v)
 4:     for all s ∈ S_uv do
 5:         S'_uv ← REACHABLE({s}, v)
 6:         if s ∈ S'_uv  &  passed a final state to reach s then
 7:             return 1
 8:         end if
 9:     end for
10:     return 0
11: end function
12:
13: function REACHABLE(S, v)
14:     S_v ← states reachable from a state in S on reading v
15:     S'_v ← ∅
16:     repeat
17:         S_v ← S_v ∪ S'_v
18:         S'_v ← states reachable from a state in S_v on reading v
19:     until S'_v ⊆ S_v
20:     return S_v
21: end function
```

In Algorithm 1, the REACHABLE function finds all the states reachable from a set of initial states S on reading some positive number of v's. On every iteration of the loop defined in line 16, the function finds the states reachable on reading another v, and the function terminates after an iteration of the loop where no new states are found.

In the MAIN function, in lines 2-3 Algorithm 1 stores in S_{uv} all the states reachable from the initial state on reading a u and a non-negative number of v's. Then in lines 4-10, the algorithm determines whether there is an accepting loop on any of the states in S_{uv} (i.e., a path from a state in S_{uv} to itself that passes through a final state).

Runtime Analysis Let n be the number of states, m be the number of transitions, and l be the length of $u\$v$. Line 2 runs in $O(nml)$ - we read each of the

$O(l)$ characters in u one at a time, and with each character we calculate the new states that can be reached using the $O(m)$ transitions from the $O(n)$ states reached so far. REACHABLE runs in $O(n^2ml)$ - the loop runs at most n times since there are at most n states to add to S, and line 18 runs in $O(nml)$. The loop in line 4 terminates after $O(n)$ iterations, and since REACHABLE is called in line 5, the loop runs in $O(n^3ml)$. Therefore, Algorithm 1 runs in $O(n^3ml)$.

B   Experimental Evaluation

The GitHub repository contains many input files that can be used to test each of the executables. There exist input files for many different sizes of M2MAs, SUBAs, and NBAs, as well as input files for different edge cases (e.g., M2MAs of dimension 1).

M2MA Dimension	Average Runtime
10	0.33s
20	2.44s
30	9.76s
40	28.57s
50	82.57s
60	177.00s
70	355.09s
80	657.82s
90	1084.85s
100	1819.80s

Table 1. M2MA.jar Runtime

Tables 1-3 give a sense of the runtimes for M2MA.jar, SUBA.jar, and NBA.jar. The experiments were run on a standard laptop with 8 GB RAM, 8 cores, and a CPU frequency of 3200 MHz.

Table 1 details the runtime of M2MA.jar. Fifty random M2MAs with the alphabet $\{a, b, c\}$ were generated for each of the dimensions $10, 20, \ldots, 50$, and the average runtimes were calculated. For the dimensions $60, 70, \ldots, 100$, ten random M2MAs were generated. These results can be reproduced using the executable M2MA_experiments.jar in the Experimental Evaluation folder of the GitHub repository. Instructions on how to use the jar file are printed to standard output once the executable is run.

Table 2 details the runtime of SUBA.jar. The SUBA input files used to generate the results are found in the Experimental Evaluation folder of the GitHub repository.

Table 3 details the runtime of NBA.jar. The NBAs in the table were randomly generated, with one NBA generated per size. Since NBA.jar uses approximate equivalence queries, the table contains a column for the number of tests to run

SUBA Language	SUBA Size	Unminimized M2MA Dim	Learned M2MA Dim	Runtime
$a\Sigma^*(\Sigma^*b\Sigma^*)^\omega$	2	10	5	0.20s
$\Sigma^*a\Sigma^5ab^\omega$	8	136	10	0.30s
$((a+b)^*(a(a+b)a(a+b)c$ $+b(a+b)b(a+b)d))^\omega$	9	171	92	50.86s
$(a^*a^4b)^\omega$	5	55	32	0.57s
$(a^*a^5b)^\omega$	6	78	44	2.04s
$(a^*a^6b)^\omega$	7	105	58	6.72s
a^ω	1	3	3	0.05s
$(ab^5)^\omega$	6	78	43	0.45s
$(ab^{10})^\omega$	11	253	133	25.11s
$(ab^{15})^\omega$	16	528	273	411.48s
$(ab^{20})^\omega$	21	903	463	3064.32s

Table 2. SUBA.jar Runtime

NBA Size	Number of tests/EQ	Learned M2MA Dimension	Runtime
2	10000	3	1.02s
4	10000	7	0.25s
6	10000	24	1.80s
8	100000	27	14.00s
10	100000	83	983.39s

Table 3. NBA.jar Runtime

*The learned M2MA dimensions and runtimes for NBA.jar can vary widely due to the inherent randomness.

for every equivalence query. The maximum length of a test for every NBA in the table is 25. The NBA input files used to generate the results are found in the Experimental Evaluation folder of the GitHub repository.

Practical Capabilities The runtimes for M2MA.jar increase at a roughly cubic rate. For smaller M2MAs, the program runs quickly, but for M2MAs of dimension larger than 100, the program can take hours to terminate. For the purpose of using ALMA to explore the properties of M2MAs, it is unlikely that one will need to work with M2MAs of dimension much larger than 100, so the program's runtime should not pose a significant constraint.

One may think that experimenting with the SUBA learning algorithm, which incurs a $2n^2 + n$ blow up in size going from the initial SUBA to the converted M2MA, will be impractical when dealing with SUBAs of size even greater than 10. However, as can be seen in Table 2, the unminimized M2MA in the SUBA learning algorithm often minimizes to a much smaller M2MA, which is why the

tool implements the minimization algorithm of Sakarovitch [10]. For example, the SUBA of size 21 representing the language $(ab^{20})^\omega$ converts into a large unminimized M2MA of dimension 903. However, it minimizes to an M2MA of dimension 463, and SUBA.jar runs in less than an hour for this SUBA.

The learned M2MA dimensions and runtimes for NBA.jar depend heavily on the number of tests performed every equivalence query. To get decent approximations on the learned M2MA dimension for NBAs of size larger than 10, the number of tests per equivalence query should be at least on the order of 10 to the 5th or 6th power. One can try to change the maximum length of the tests to get better results. For randomly generated NBAs of size much larger than 10, the trade-off between the runtime and accuracy becomes much more apparent, and the number of tests per equivalence query may have to be decreased for the runtime to be practical.