


# Baum-Welch Algorithm for Markov Models Using Algebraic Decision Diagrams

Sebastian Aaholm\*, Lars Emanuel Hansen†, Daniel Runge Petersen<sup>‡</sup>,



**Abstract**—The Baum-Welch (BW) algorithm is a widely used method for training Hidden Markov Models (HMMs) and Markov Chains (MCs) from observation sequences. However, traditional implementations using recursive or matrix-based methods often struggle with scalability due to redundancy and high memory consumption. This thesis proposes a novel, symbolic implementation of the BW algorithm using Algebraic Decision Diagrams (ADDs), which provide a compact and efficient representation of probabilistic models. We present CUPAAL, a C++ library that implements the BW algorithm entirely with ADDs, and integrate it into the JAJAPY library, resulting in a new symbolic learning tool referred to as JAJAPY 2.

Our approach enables efficient learning from multiple observation sequences and supports both HMMs and MCs. Through experiments on models from the QComp benchmark set, we demonstrate that the symbolic implementation significantly improves performance for larger observation sets and models with repeated structures, while maintaining learning accuracy. These results affirm the potential of ADD-based symbolic computation as a scalable alternative for probabilistic model learning.

## 1 INTRODUCTION

THE Baum-Welch algorithm is a widely used method for training Markov models in applications such as speech recognition, bioinformatics, and financial modeling [1–3].

Traditionally, the Baum-Welch algorithm relies on matrix-based or recursive approaches to estimate model parameters from observed sequences.

An example of this is the JAJAPY library [4], which implements the Baum-Welch algorithm using a recursive matrix-based approach. This library is designed to learn probabilistic models from partially observable executions, producing observation sequences - also known as traces.

The key strength of JAJAPY lies in its flexibility to accommodate various learning scenarios, along with seamless integration into standard verification workflows using tools like STORM and PRISM. However, the performance of JAJAPY’s Baum-Welch algorithm implementation has been a significant limitation, because of the inherent redundancy in matrix-based representations, which leads to inefficiencies particularly in terms of time and memory consumption, which restricts its scalability to larger models.

To address these challenges, we propose a novel approach that replaces conventional matrices and recursive formulations

with ADDs. ADDs provide a compact, structured representation of numerical functions over discrete variables, enabling efficient manipulation of large probabilistic models.

By leveraging ADDs, we can exploit the sparsity and structural regularities of HMMs and MCs, significantly reducing memory consumption and accelerating computation.

This paper presents several contributions toward efficient learning of HMMs and MCs models, by leveraging ADDs:

First, we extend the BW algorithm for these models using symbolic computation, reformulating each algorithm step as operations on ADDs, leveraging the CUDD library to carry out these operations symbolically using ADDs. This reformulation enables efficient calculation of the Markov models in a compact and scalable form.

Secondly, our approach extends previous work on symbolic calculation by accommodating learning from multiple observation sequences for both types of Markov models, broadening the applicability of symbolic learning.

Thirdly, we conduct an experimental evaluation of the scalability of the symbolic Baum-Welch algorithm for a MC from the QComp benchmark set [5], which serves as a standard reference for evaluating the performance of probabilistic model checking algorithms. Our findings suggest that replacing matrices and recursive formulations with ADDs offers a scalable alternative, making Markov model-based learning feasible for larger and more complex datasets.

Additionally, we implement Python bindings for the CUPAAL tool, making it accessible and usable within Python-based machine learning and model-checking workflows, such as JAJAPY.

Finally, using these python bindings we integrate CUPAAL into JAJAPY as JAJAPY 2, enabling users to seamlessly run symbolic probabilistic learning algorithms within JAJAPY.

## 2 PREVIOUS WORK

In this section, we provide a brief overview of previous work that has influenced our research and has been iterated upon. Specifically, we discuss what these tools are, how they function, who utilizes them, and the motivations behind integrating them into our research. The focus will be on four primary tools: PRISM STORM JAJAPY and CUPAAL.

### 2.1 Jajapy

JAJAPY provides learning algorithms designed to construct accurate models of a system under learning (SUL) from

• All authors are with the Dept. of Computer Science, Aalborg University, Aalborg, Denmark  
• E-mails: \*saahol20, †leha20, ‡dpet20@student.aau.dk

observed traces. Once learned, these models can be directly exported for formal analysis in tools such as STORM and PRISM.

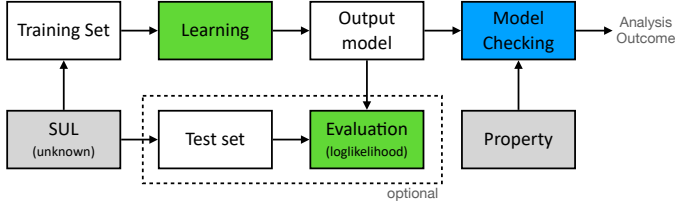


Fig. 1. Modeling and verification workflow using JAJAPY. Phases involving JAJAPY are highlighted in green, while the blue phase represents verification using STORM or PRISM.

In this context, we refer to the *training set* as the collection of observation traces used to infer a model of the SUL, and the *test set* as a separate set of traces used to evaluate the quality of the learned model.

JAJAPY supports learning various types of models, depending on the structure of the training data. For clarity, this paper focuses specifically on the new features introduced in JAJAPY 2, which primarily target Markov chains. However, these improvements are equally applicable to other classes of Markov models supported by the tool.

At the core of JAJAPY’s learning capabilities are several variants of the Baum-Welch (BW) algorithm [6, 7], adapted for discrete-time Markov chains (MCs), Markov decision processes (MDPs)[8], and continuous-time Markov chains (CTMCs)[9].

Each algorithm requires two inputs: a training set and the desired number of states for the output model. The process begins with the creation of a randomly initialized model (e.g., a Markov chain) and iteratively updates its transition probabilities, increasing the likelihood of transitions that better explain the observed traces.

The efficiency and accuracy of the learning process depends heavily on the choice of the initial hypothesis. To improve convergence and model quality, JAJAPY allows users to supply a custom initial hypotheses in several formats, including STORMPY sparse models, PRISM-files, or native JAJAPY model definitions.

An example of using JAJAPY to learn a 10-state Markov chain from a training set, starting from a random initial hypothesis, is shown in Listing 1.

```

1 import jajapy
2 training_set = jajapy.loadSet("Path/to/data")
3 type(training_set) # list
4
5 learned_model = jajapy.BW().fit(training_set,
6     nb_states=10)
7 type(learned_model) # stormpy.SparseDtmc

```

Listing 1. Example of using JAJAPY’s BW implementation to learn a 10-state Markov chain from a training set.

JAJAPY supports reading PRISM files, using STORM (through STORMPY), as well as direct verification of learned model, through properties, provided the properties are supported. Alternatively, the model can be exported to PRISM’s format for verification using the PRISM model checker.

## 2.2 CuPAAL

CUPAAL is a tool developed in C++ that extends the work done in JAJAPY by implementing the Baum-Welch algorithm with an ADD-based approach instead of a recursive method. The goal of CuPAAL is to leverage ADDs to improve the efficiency of learning Markov models, particularly in large-scale applications where traditional recursive methods may become computationally expensive.

CuPAAL has undergone multiple iterations. Initially, it implemented a partial ADD-based approach, where only the calculation of the alpha and beta values of the Baum-Welch algorithm were implemented using ADDs. This partial implementation served as an initial proof-of-concept to determine whether incorporating ADDs could yield performance benefits compared to the recursive approach employed by JAJAPY.

Following promising results from the partial implementation, further development led to a fully ADD-based version of CuPAAL for HMMs. This iteration replaced all recursive computations with ADDs, enabling more efficient execution, particularly for large models. The transition to a fully ADD-based approach demonstrated the potential for significant computational savings and scalability improvements, reinforcing the viability of this method for broader applications beyond our initial research scope.

Because there is no notion of HMM in the PRISM formalism, we’ve implemented the BW algorithm for use with MCs. Given the similarities between these model types, we’ve reused a lot of the previous work in the implementation.

By building upon JAJAPY and developing CuPAAL, we have been able to evaluate the impact of using ADDs in probabilistic model learning.

## 3 PRELIMINARIES

This section provides an overview of the theoretical background necessary to understand the rest of the article. For ease of reference Appendix B contains a table of symbols used in the paper.

We begin by defining the key concepts of a HMM and a MC, which are the two main models used in this report, then go on to introduce the BW, which is a widely used algorithm for training HMMs, and showing how it can be adapted to handle multiple observation sequences using matrix operations.

### 3.1 Hidden Markov Model

Hidden Markov Models (HMMs) were introduced by Baum and Petrie in 1966 [10] and have since been widely used in various fields, such as speech recognition [1], bioinformatics [2], and finance [3].

A HMM is a statistical model that describes a system that evolves over time. The system is assumed to hold the Markov property, meaning that the future state of the system only depends on the current state and not on the past states. The system is also assumed to be unobservable, meaning that the states are hidden and cannot be directly observed. Instead, the system emits observations, which are used to infer the hidden states.

**Definition 1** (Hidden Markov Model). *A Hidden Markov Model (HMM) is a tuple  $\mathcal{M} = (S, L, \omega, \tau, \pi)$ , where:*

- $S$  is a finite set of states.
- $L$  is a finite set of labels.
- $\omega : S \rightarrow D(L)$  is the emission function.
- $\tau : S \rightarrow D(S)$  is the transition function.
- $\pi \in D(S)$  is the initial distribution.

$D(X)$  denotes the set of probability distributions over a finite set  $X$ . The emission function  $\omega$  describes the probability of emitting a label given a state. The transition function  $\tau$  describes the probability of transitioning from one state to another. The initial distribution  $\pi$  describes the probability of starting in a given state.

An example of a HMM is a weather model where the hidden state represents the actual weather (sunny, rainy, or cloudy), but we only observe indirect signals, such as whether someone is carrying an umbrella or wearing sunglasses.

### 3.2 Markov Chain

A Markov Chain (MC), named after Andrei Markov, is a stochastic model widely used in different fields of study [7].

**Definition 2** (Markov Chain). *A Markov Chain (MC) is a tuple  $\mathcal{M} = (S, L, \omega, \tau, \pi)$  identical to the HMM structure above except that the emission function is deterministic: for every  $s \in S$  there is a single label  $l = \omega(s)$  emitted with probability 1.*

In other words, the emission function  $\omega$  is a function that maps each state to a single label  $l \in L$ , meaning that each state emits exactly and only one label. Two distinct states may emit the same label.

An example of a MC is a board game where a player moves between squares based on dice rolls. Each square corresponds to a state, the dice rolls determine the transition probabilities.

### 3.3 Conversion between MCs and HMMs

In this section, we will discuss the conversion between MCs and HMMs. This conversion is important because it allows us to use the same algorithms and techniques from the original CUPAAL implementation for both model types, even though they have different properties.

In our case, we're interested in trace-equivalent models. By trace-equivalent, we mean that the probability distribution over observed sequences is the same for both models. i.e. the labels emitted by moving through the probabilistic models follow the same distribution.

From the definition of a MC, we can see that it is a special case of an HMM where the emission function is deterministic, which makes this conversion very simple.

**Definition 3** (Markov Chain to Hidden Markov Model). *For each MC  $\mathcal{M} = (S, L, \omega, \tau, \pi)$ , there exists a trace-equivalent HMM  $\mathcal{M}' = (S', L', \omega', \tau', \pi')$ , where:*

- $S' = S$ .
- $L' = L$ .
- $\omega'(s)(l) = \begin{cases} 1 & l = \omega(s) \\ 0 & \text{otherwise} \end{cases}$
- $\tau' = \tau$ .
- $\pi' = \pi$ .

The only difference in this case is the structure of the emission functions, thus preserving the probabilistic trace equivalence, i.e., for an arbitrary trace  $O \in L^*$  we have  $P[O | \mathcal{M}] = P[O | \mathcal{M}']$ .

### 3.4 Baum-Welch Algorithm

The BW is a special case of the Expectation-Maximization (EM) framework used to estimate the parameters of a HMM given a set of observed sequences.

Since the underlying states are not directly observable, the algorithm iteratively refines the model parameters  $\pi$ ,  $\omega$ , and  $\tau$  to maximize the likelihood of the observations. Each iteration of the algorithm consists of two steps:

- E-step Compute the expected values of the hidden variable given the current parameters.
- M-step Update the model parameters to maximize the expected complete-data log-likelihood.

Convergence is typically achieved when the change in the likelihood (or parameters) between iterations falls below a threshold [7].

We can represent the parameters of a HMM as matrices for computational efficiency.

They are defined as follows:

- $\pi$  is the initial state distribution vector, where  $\pi_i = \pi(s_i)$  is the probability of starting in state  $s_i$ , this is a column vector of size  $|S|$ .
- $\tau$  is the transition matrix, where  $\tau_{ij} = \tau(s_i)(s_j)$  is the probability of transitioning from state  $s_i$  to state  $s_j$ , this is a square matrix of size  $|S| \times |S|$ .
- $\omega$  is the emission matrix, where  $\omega_{ij} = \omega(s_i)(l_j)$  is the probability of emitting label  $l_j$  given state  $s_i$ , this is a matrix of size  $|S| \times |L|$ .

To illustrate our symbolic implementation, we describe a single Baum-Welch iteration in terms of matrix operations, assuming familiarity with the algorithm. For an introductory treatment, see [6, 11].

Let  $\mathcal{M}$  denote the current HMM hypothesis and let  $O = o_1 \dots o_T$  be a sequence of observations, where each  $o_t \in L$  and the observation sequence has the length  $T$ . Suppose  $\mathcal{M}$  has  $n$  states and  $m$  labels, i.e.,  $S = s_1, \dots, s_n$ , with parameters represented as follows:

- $\pi \in [0, 1]^n$  is the initial state distribution column vector.
- $\tau \in [0, 1]^{n \times n}$  is the transition probability matrix.
- $\omega \in [0, 1]^{n \times m}$  is the emission probability matrix.

The forward and backward algorithms are implemented using dynamic programming, as shown in Listing 2. For a given time step  $t$ , let  $\omega(t)$  be the column vector of emission probabilities for label  $o_t$  for each state, and  $\odot$  the Hadamard (element-wise) product.

#### FORWARD-ALGORITHM

```

1  $\alpha(1) = \omega(1) \odot \pi$ 
2 for  $t = 2$  to  $T$ 
3    $\alpha(t) = \omega(t) \odot (\tau^T \alpha(t-1))$ 
```

#### BACKWARD-ALGORITHM

```

1  $\beta(T) = \mathbf{1}$ 
2 for  $t = T-1$  to 1
3    $\beta(t) = \tau (\beta(t+1) \odot \omega(t+1))$ 
```

Listing 2. Computation of the forward and backward coefficients



The above procedures compute the column vectors  $\alpha(t)$  and  $\beta(t) \in [0, 1]^n$  for  $t = 1 \dots T$  which are later used to compute the coefficients  $\gamma(t) \in [0, 1]^n$  and  $\xi(t) \in [0, 1]^{n \times n}$  as follows:

$$\gamma(t) = \tau[O|\mathcal{M}]^{-1} \cdot \alpha(t) \odot \beta(t) \quad (1)$$

$$\xi(t) = (\tau[O|\mathcal{M}]^{-1} \cdot \tau) \odot (\alpha(t) \otimes (\beta(t+1) \odot \omega(t+1))^T) \quad (2)$$

Here,  $\otimes$  is the Kronecker product and the probability  $\tau[O|\mathcal{M}]$  to observe  $O$  in  $\mathcal{M}$  is computed as  $\mathbf{1}^T \alpha(T)$ . We calculate  $\gamma(t)$  from  $t = 1$  to  $T$  and  $\xi(t)$  from  $t = 1$  to  $T - 1$ .

Finally, the initial probability vector, the transition probability matrix and emission matrix are updated as follows:

$$\hat{\pi} = \gamma(1) \quad (3)$$

$$\hat{\omega} = (\mathbf{1} \oslash \gamma) \cdot \left( \sum_{t=1}^T \gamma(t) \otimes [[o_t]] \right) \quad (4)$$

$$\hat{\tau} = (\mathbf{1} \oslash \gamma) \cdot \xi \quad (5)$$

Where  $\cdot$  is the transposed Khatri-Rao product (i.e., row-by-row Kronecker product), and  $[[o_t]] = ([[o_t = l]])_{l \in L}$  is the one-hot encoding of the observation  $o_t$ , meaning that it is a row vector of size  $|L|$  with a 1 in the position corresponding to the observation  $o_t$  and 0 elsewhere.  $\oslash$  is the element-wise division, and  $\otimes$  is the Kronecker product. The  $\gamma$  and  $\xi$  are defined as follows:

$$\gamma = \sum_{t=1}^T \gamma(t) \quad (6)$$

$$\xi = \sum_{t=1}^{T-1} \xi(t) \quad (7)$$

These update rules form the standard BW for training HMMs on a single observation sequence. However, the approach can be naturally extended to multiple sequences.

The BW algorithm runs until convergence, in this case until the difference in log-likelihood between iterations is less than  $\ell(\mathcal{M}; O) - \ell(\hat{\mathcal{M}}; O) < \epsilon$ . The log-likelihood of an iteration can be calculated using the  $\alpha$  probabilities in this way:

$$\ell(\mathcal{M}; O) = \sum_{s=1}^S \log(\alpha(T)_s) \quad (8)$$

### 3.5 Multiple Observation Sequences

Suppose we are given a multiset of independently identically distributed (i.i.d.) observation sequences  $\mathcal{O} = O_1, O_2, \dots, O_{|\mathcal{O}|}$ , where each  $O_i = (o_{i1}, o_{i2}, \dots, o_{iT})$  is of length  $T$ . The E-step remains unchanged: for each sequence, we compute the corresponding  $\alpha_i(t)$ ,  $\beta_i(t)$ ,  $\gamma_i(t)$ , and  $\xi_i(t)$  values independently.

In the M-step, we aggregate statistics across all sequences to update the parameters. Specifically, we define:

$$\gamma = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^T \gamma_i(t) \quad (9)$$

$$\xi = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T-1} \xi_i(t) \quad (10)$$

With these aggregate quantities, the update rules for the initial distribution (see Equation 3) and transition matrix (see Equation 5) remain unchanged, because they are already defined in terms of the sum over all sequences. However, the emission matrix update needs to account for all sequences.

The emission matrix is updated as follows:

$$\omega_s(o) = (\mathbf{1} \oslash \gamma) \cdot \left( \sum_{i=1}^N \sum_{t=1}^T \gamma_i(t) \otimes [[o_{it}]] \right) \quad (11)$$

This mirrors the single-sequence case (see Equation 4) but extends the summation in the left side of the Kronecker product to cover all sequences and all time steps. This allows us to compute the emission probabilities for each state across all sequences, ensuring that the model captures the overall distribution of observations.

### 3.6 Baum-Welch Algorithm for Markov Chains

Since MCs can be seen as HMMs with deterministic emissions, where each state emits a unique observation with probability 1, the BW simplifies accordingly when applied to MCs.

In this case:

- The forward and backward algorithms are computed identically to the HMM case, but without weighting by emission probabilities, as these are implicitly handled by the observation sequence.
- The **E-step** computations for  $\gamma(t)$  and  $\xi(t)$  remain structurally the same, though emission terms are omitted due to determinism.
- The **M-step** updates for the initial distribution  $\pi$  and the transition matrix  $\tau$  are unchanged.
- The emission matrix  $\omega$  is not updated, as it is fixed by the model's structure and need not be learned.

This simplification avoids unnecessary computation and reflects the reduced uncertainty in the model: there is no need to marginalize over emissions, as each state deterministically produces a known label. Consequently, the BW becomes more efficient when applied to MCs.

### 3.7 Decision Diagrams

Binary Decision Diagrams (BDDs) are data structures for efficiently representing and manipulating Boolean functions. They are a compressed representation of truth tables, capturing the logical structure of a function in a graph-based format by eliminating redundancy, reducing memory usage, and improving computational efficiency [12].

A Binary Decision Diagram (BDD) is a directed acyclic graph derived from a decision tree, where each non-terminal node represents a Boolean variable, edges correspond to binary assignments (0 or 1), and terminal nodes store function values (0 or 1). To reduce the size of the decision tree, BDDs exploit redundancy by merging equivalent substructures, resulting in a canonical form (when reduced and ordered) that allows for efficient operations such as function evaluation, equivalence checking, and Boolean operations [12].

BDDs have been widely used in formal verification, model checking, and logic synthesis due to their ability to compactly represent large Boolean functions while maintaining efficient computational properties. However, in rare cases BDDs can

suffer from exponential blowup. This can occur particularly when dealing with functions that lack inherent structure or when representing numerical computations that go beyond Boolean logic.

### 3.8 ADDs

Algebraic Decision Diagrams (ADDs) generalize the concept of BDDs by allowing terminal nodes to take values beyond Boolean constants (0 and 1). Instead of restricting values to true/false, ADDs can store arbitrary numerical values, making them useful for representing and manipulating functions over discrete domains [13]. This generalization enables the efficient representation of functions such as cost functions [14], probabilities [15], and other numerical relationships that arise in probabilistic reasoning.

The fundamental structure of an ADD remains similar to a BDD, where a decision tree is compacted by merging redundant substructures. However, instead of performing Boolean operations, ADDs allow for arithmetic operations such as addition and multiplication, making them well-suited for representing matrices [13].

## 4 IMPLEMENTATION

This section will provide an overview of the implementation of Cupaal. This will include the Baum-Welch algorithm, and how CUPAAL is integrated into JAJAPY, creating JAJAPY 2.

### 4.1 Motivation for CuPAAL

The motivation for CUPAAL is to provide a more efficient and scalable implementation of the Baum-Welch algorithm for parameter estimation. Specifically, we aim to improve the performance of the algorithm when dealing with large and complex models, and improve upon the existing limitations of the Baum-Welch algorithm in JAJAPY.

#### 4.1.1 Recursive vs. Matrix vs. ADD-based Approaches

When working with the Baum-Welch algorithm, different approaches can be taken to optimize computational efficiency. Three common strategies are recursive, matrix-based, and ADD-based approaches, each with distinct advantages and limitations.

- **Recursive Approach:** Conceptually simple, recursion follows a divide-and-conquer strategy, and makes use of a dynamic programming approach. Previous calculations are used to build upon future calculations. These results are stored in a list or a map, so that they can be accessed when needed [16, Chapter 4].
- **Matrix Representation:** Reformulating algorithms using matrix operations leverages algebraic properties for parallel computation and efficient processing. By building upon the recursive approach, matrices provide an efficient method of accessing the stored results leading the faster computations overall [16, Chapter 4, 15 & 28].
- **ADD-based Approach:** ADDs provide a compact representation that eliminates redundancy in recursive computations. By reusing previously computed substructures, they improve efficiency and reduce memory

overhead [13]. Compared to matrices, ADDs can offer a more space-efficient alternative for structured data while extending BDD techniques to handle both Boolean and numerical computations.

In this work we explore the benefits of ADD-based approaches for solving complex problems, focusing on parameter estimation in Discrete Time Markov Chains (DTMCs) and Continuous Time Markov Chains (CTMCs). We compare the performance of ADD-based algorithms against recursive-based implementations, highlighting the advantages of using ADDs for efficient computation and memory management.

### 4.2 What is CuPAAL

CUPAAL is a C++ library that implements the Baum-Welch algorithm for parameter estimation, and has evolved over time.

The initial version of CUPAAL was written in C and called *SUDD*, which was a partial implementation of the Baum-Welch algorithm, using ADDs. This version was mainly focused on displaying the efficiency of ADDs for parameter estimation problems, and was not fully functional. The next iteration was called CUPAAL, which was a complete implementation of the Baum-Welch algorithm, using ADDs, however it only supported HMMs, and was only designed to make use of a single observation.

The current version of CUPAAL has been extended to support DTMCs and can handle multiple observations. This version of CUPAAL is designed to be used in conjunction with JAJAPY, allowing for easy integration and use in parameter estimation problems.

The following sections will provide an overview of what CUPAAL is, and what it can do.

#### 4.2.1 What Does Cupaal Contain

Throughout all its iterations, CUPAAL has made use of Colorado University Decision Diagram (CuDD). Colorado University Decision Diagram (CuDD) is a library for implementing and manipulating BDDs and ADDs developed at the University of Colorado. The CuDD library [17] is a powerful library for implementing and manipulating various types of decision diagrams, including BDDs and ADDs.

Implemented in C, the CuDD library ensures high-performance execution and can be seamlessly integrated into C++ programs, which we utilize in CUPAAL. By leveraging the CuDD library, we demonstrate the benefits of ADD-based approaches for solving parameter estimation problems in DTMCs.

The CuDD library is used to store ADDs and perform operations on them. Its optimized algorithms and efficient memory management enable symbolic handling of large and complex matrices, significantly improving performance compared to traditional methods.

We have not modified or extended the CUDD library. All functionality used in our implementation is available through the standard CUDD library.

#### 4.2.2 From Prism to Cupaal

In the current iteration of CUPAAL, it is possible to use Prism models as input to the Baum-Welch algorithm.

The models are encoded from Prism models to CUPAAL models. This is done by parsing the Prism model to JAJAPY, using Stormpy.

The Jajapy model contains a matrix for it's transitions, a matrix for it's labels, and a vector for the initial state. The model is passed to CUPAAL where these matrices and vectors are encoded into ADDs, as a function  $f : \{0, 1\}^n \times \{0, 1\}^n \rightarrow R$ .

The Transition matrix is a  $S \times S$  matrix, where  $S = States$ , and is encoded to an ADD, by each row and column with a binary value. This value is determined based on the size of the matrix,  $n = \lceil \log_2(S) \rceil$ .

The label matrix is a  $S \times L$  matrix, where  $L = Labels$  and since there is no guarantee that  $S = L$ , the encoding is handled differently. The matrix is instead treated as a list of vectors. Each vector is encoded as square matrices, where each row or column (depending on the vector type) is duplicated, which is then encoded to a list of ADDs. Knowing the exact dimensions of matrices and that they are square helps to simplify some of the symbolic operations. An example of this will be displayed in subsubsection 4.2.3.

The Initial state vector is encoded similarly to the label matrix, but only as a single ADD.

#### 4.2.3 Kronecker Product Implementation

The Kronecker product is implemented in CUPAAL, using the row and column duplication method mentioned in subsubsection 4.2.2.

The structure of Decision Diagrams in CUPAAL, where keeping track of all the new binary values used for encoding from a matrix to an ADD can add a layer of complexity for calculation. Especially when computing operations that translate matrices to new dimensions, such as the Kronecker product. This matrix-based approach enables efficient symbolic operations, as the Kronecker product can be calculated by taking the Hadamard product between a column matrix ADD and a row matrix ADD, simplifying what would otherwise be a more complex operation.

An example of this can be seen with the two vectors  $\hat{A}$  and  $\hat{B}$

$$\text{Let } \hat{A} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \text{ and } \hat{B} = \begin{bmatrix} 3 & 4 \end{bmatrix}.$$

The Kronecker product of these two vectors is computed as follows:

$$\hat{A} \otimes \hat{B} = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}. \quad (12)$$

Another way to calculate the Kronecker product is to expand the vectors into matrices.  $\hat{A}$  and  $\hat{B}$  are expanded to be matrices, similar to how the matrix was treated as a list of vectors and then expanded to square matrices, as seen with the Label matrix.

$$\text{Let } \mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 3 & 4 \\ 3 & 4 \end{bmatrix}.$$

The Kronecker product of  $\hat{A}$  and  $\hat{B}$  can also be calculated, by using the Hadamard product of  $\mathbf{A}$  and  $\mathbf{B}$ . This is done as follows:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}. \quad (13)$$

Hereby showing that the Hadamard product can be used to compute the Kronecker product between two matrices, by

using the row and column duplication method. This is a technique that only works for Kronecker products between vectors, specifically one row and one column vector, as it relies on the structure of the vectors being expanded into square matrices.

### 4.3 Implementation to Jajapy

This section will give an overview of how CUPAAL is implemented into JAJAPY, using bindings between C++ and Python. Figure 2 shows the overall architecture of the implementation.

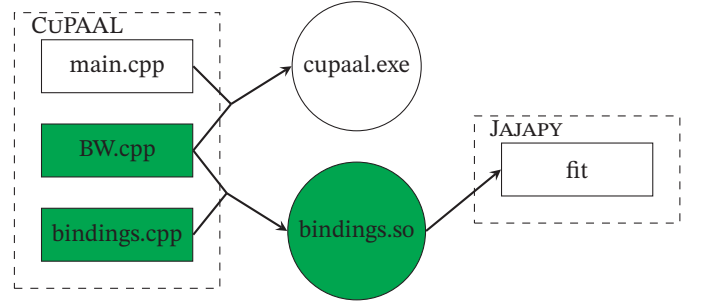


Fig. 2. Architecture of CUPAAL combined with JAJAPY.

CUPAAL consists of two main components, the main function and the BW library. Both of these are compiled to an executable program, called `cupaal.exe`, which can be used to run the Baum-Welch algorithm on a given model.

#### 4.3.1 Bindings

To implement CUPAAL into JAJAPY, we create bindings between C++ and Python using the `pybind11` library [18]. This allows us to call C++ functions from Python, enabling us to use CUPAAL in JAJAPY.

In the code examples, some parts have been removed for brevity and clarity.

```

1 // Some parameters have been removed for brevity
2 cupaal_markov_model
3   ↳ cupaal_bw_symbolic(vector<string>& states,
4   ↳ vector<string>& labels,
5   ↳ vector<vector<string>>& observations,
6   ↳ vector<double>& initial_distribution,
7   ↳ vector<double>& transitions, vector<double>&
8   ↳ emissions, int max_iterations = 100, double
9   ↳ epsilon = 1e-2) {
10    MarkovModel model(states, labels,
11    ↳ initial_distribution, transitions,
12    ↳ emissions, observations);
13    cupaal_markov_model model_data;
14    chrono::seconds time = chrono::seconds(3600);
15    model.baum_welch_multiple_observations(
16    ↳ max_iterations, epsilon, time);
17
18    // Removed output and result path for brevity
19    model_data.initial_distribution =
20    ↳ model.initial_distribution;
21    model_data.transitions = model.transitions;
22    model_data.emissions = model.emissions;
23
24    Cudd_Quit(model.manager);
25    return model_data;
26 }

```

Listing 3. C++ bindings file for CuPAAL



We create a C++ bindings file, that uses the BW library from CUPAAL and define the function we want to expose to Python, we call this function *cupaal\_bw\_symbolic*, seen in Listing 3. This function takes model parameters from a JAJAPY model as input, and transforms them to be used in CUPAAL. The transformation is done at line 3, where all the parameters are inputed to create a `Markov Model` object, which is then used to run the Baum-Welch algorithm at line 6. Each of the values that are relevant from the BW algorithm are then passed into the `model_data` object, which is then returned to JAJAPY, seen in lines 10 through 15.

These being the initial distribution, the transitions and the emissions.

```
1 void baum_welch_multiple_observations(unsigned int
   ↳ max_iterations = 100, double epsilon = 1e-6,
   ↳ chrono::seconds time = chrono::seconds(3600));
```

Listing 4. Prototype of the function used to run the Baum-Welch algorithm on multiple observations in CuPAAL.

The C++ bindings file is then compiled to a shared library, which can be imported in JAJAPY. JAJAPY can call the *cupaal\_bw\_symbolic* function, which will then call the CUPAAL implementation of the Baum-Welch algorithm.

We create a new function in JAJAPY, called *\_bw\_symbolic*, which is used to call the CUPAAL implementation of the Baum-Welch algorithm, as seen in Listing 5. This function is used to prepare the model parameters from JAJAPY, so they are in the correct format for CUPAAL. The preparation is done at lines 7 through 20, after this the CUPAAL implementation is called at line 22, where the *cupaal\_bw\_symbolic* function is called with the prepared parameters, giving the CUPAAL model as a return value.

The values are then extracted from the CUPAAL model and assigned to the JAJAPY model, seen in lines 23 through 25. where they are reshaped to be in line with JAJAPY.

The fit function in JAJAPY is modified to call the *\_bw\_symbolic* function when a new parameter called *symbolic* is set to true, as seen in Listing 6.

A check is made to see if the *symbolic* parameter is set to true at line 4, and when the parameter is true, the JAJAPY model will call the Listing 5 function, which will then call the CUPAAL implementation of the Baum-Welch algorithm.

#### 4.4 Integration Discussion

The integration of CUPAAL into JAJAPY has been successful, allowing us to leverage the Baum-Welch algorithm for parameter estimation for HMMs and DTMCs.

The decision to use pybind11 for creating bindings between C++ and Python has proven to be effective, as it allows us to call C++ functions from Python, but further considerations should be made for the future if this is the best approach.

The exact implementation of the symbolic fit function in JAJAPY is shown in Listing 6, is to be discussed with the JAJAPY creator and some changes are expected to be made.

## 5 EXPERIMENTS

In this section, we present an evaluation comparing the performance of two implementations of the BW algorithm:

the original version from JAJAPY and the new symbolic implementation introduced in JAJAPY 2.

For this comparison, we use a MC model, taken from the QComp benchmark set [19], which is a collection of models used for quantitative verification and learning tasks. The goal is to assess scalability of the symbolic implementation and its performance in terms of runtime and accuracy.

We designed three experiments to evaluate the performance of the symbolic implementation of the BW algorithm in JAJAPY 2:

- **Scalability** — Evaluating how the symbolic implementation scales with increasing model size.
- **Accuracy** — Comparing the accuracy of the symbolic implementation against the original recursive implementation.
- **Extra Scalability** — Evaluating the scalability of the symbolic implementation when adjusting the initialization of the model hypothesis.

The experiments are designed to answer the following research questions:

- **Question 1:** How does runtime scale as model size increases for JAJAPY 2 vs JAJAPY?
- **Question 2:** What is the relative estimation accuracy of the symbolic implementation in JAJAPY 2 compared to the original recursive implementation in JAJAPY?
- **Question 3:** How much does an informed initialization accelerate JAJAPY 2?

The experiments are designed to provide insights into the performance and scalability of the symbolic implementation of the BW algorithm in JAJAPY 2, and to compare it with the original recursive implementation in JAJAPY.

### 5.1 Model

The model used in the experiments is "leader\_sync", which is a DTMC model from the QComp benchmark set [19]. The model is from the PRISM case study [20], which is a collection of models used for quantitative verification and learning tasks.

The "leader\_sync" model is a distributed system model that simulates the behavior of a group of processors that need to choose a leader among themselves. The model's size ranges from 26 to 1050 states, depending on the number of processors in the system, the states increase {26, 69, 147, 61, 274, 812, 141, 1050}.

The non-linear progression in model size arises because the QComp benchmark defines model variants using two parameters: the number of processors and the maximum number a leader can select to be elected.

The model is chosen because of its scalability, making it suitable for evaluating the performance of the symbolic implementation of the BW algorithm in JAJAPY 2. The model was also simple to understand and analyze, allowing us to add more labels to the model to make it suitable for the experiments.

The labels added to the model are shown in Listing 7.

### 5.2 Experimental Setup

All experiments were conducted on the same machine, see Appendix A for full specs and the python environment.

The following steps were taken to set up the experiments:

---

```

1 def _bw_symbolic(self, max_iteration = 100, epsilon = 1e-2, outputPath = "", resultPath = ""):
2     try:
3         import libcupaall_bindings
4     except ModuleNotFoundError:
5         print("Cannot find module")
6
7     states = [str(i) for i in range (self.h.nb_states)]
8     labels = list(set(self.h.labelling))
9     observations = []
10    for times, sequences in zip(self.training_set.times, self.training_set.sequences):
11        for i in range(times):
12            observations.append(list(sequences))
13    initial_state = self.h.initial_state.tolist()
14    transitions = self.h.matrix.flatten().tolist()
15    emissions = zeros((len(labels), self.h.nb_states))
16    for row in range(len(labels)):
17        for col in range(self.h.nb_states):
18            if self.h.labelling[col] == labels[row]:
19                emissions[row][col] = 1
20    emissions = emissions.flatten().tolist()
21
22    cupaal_model = libcupaall_bindings.cupaal_bw_symbolic( states, labels, observations, initial_state,
23        ↳ transitions, emissions, max_iteration, epsilon, outputPath, resultPath)
24    self.h.initial_state = array(cupaal_model.initial_distribution)
25    self.h.matrix = array(cupaal_model.transitions).reshape( self.h.nb_states, self.h.nb_states)
26    self.h.emissions = array(cupaal_model.emissions).reshape( len(labels), self.h.nb_states)
27    return self.h

```

---

Listing 5. Jajapy's implementation of the Baum-Welch algorithm using CuPAAL.

---

```

1 # Some parameters have been removed for brevity
2 def fit(self, output_file: str, output_file_prism:
3     ↳ str, epsilon: float, max_it: int, symbolic:
4     ↳ bool):
5     # Removed preparation and settings number of
6     ↳ processes, for brevity
7     if symbolic :
8         return self._bw_symbolic(max_it, epsilon,
9             ↳ output_file, output_file_prism)
10    else:
11        return self._bw(max_it, pp, epsilon,
12            ↳ output_file, output_file_prism,
13            ↳ verbose, stormpy_output, return_data)

```

---

Listing 6. Jajapy's fit function, which calls the CuPAAL implementation of the Baum-Welch algorithm when symbolic is set to true.

---

```

1 label "reading" = s1=1&s2=1&s3=1;
2 label "deciding" = s1=2&s2=2&s3=2;
3 label "elected" = s1=3&s2=3&s3=3;

```

---

Listing 7. Labels added to the "leader\_sync" model.

- 1) Load the PRISM model
- 2) Generate  $N_{\text{seq}} \in \{25, 50, 100\}$  observation sequences of length 20.
- 3) Create a random initial MC using `MC_random`.
- 4) Run BW for up to 4 hours or until the log-likelihood difference converges to a threshold of 0.01.
- 5) Record runtime and save the model.

We save both the initial models and observations in files, to ensure both implementations use inputs by default. The generation of the training set and the randomization of the model is done using JAJAPY, which provides a convenient way to generate random models and training sets. The training set is generated by creating a set of observation sequences from the original model, which is then used to train the randomized model.

We then run the BW algorithm on the randomized model and the training set for both the original recursive implementation in JAJAPY and the symbolic implementation in JAJAPY 2. We run each implementation for each model size and number of observation sequences ten times to obtain average results.

The results of the experiments are recorded, including the runtime, number of iterations, log-likelihood, and error of the estimated transition probabilities.

We do not measure memory usage, as the symbolic implementation is implemented in C++ using Python bindings making it difficult to measure memory usage accurately, therefore we focus on runtime and accuracy.

### 5.3 Experiment 1: Scalability

The first experiment evaluates the scalability of the symbolic implementation of the BW algorithm in JAJAPY 2. The goal is to measure the runtime performance of the symbolic implementation as the size of the model increases. The experiment measures the average runtime of the BW algorithm for each model size and number of observation sequences.

### 5.4 Experiment 2: Accuracy

The second experiment evaluates the accuracy of JAJAPY 2 compared to the original JAJAPY. The goal is to measure the accuracy of the symbolic implementation in terms of log-likelihood and absolute error of the estimated transition probabilities. The absolute error is defined as:

$$\text{Error} = |e - r|,$$

where  $e$  is the estimated transition probability and  $r$  is the reference value from the original model. We use the results from the first experiment, and compare the log-likelihood and absolute error of the properties estimated by the symbolic implementation against the original recursive implementation.



The properties used in this experiment are shown in Listing 8, the properties are taken from the QComp benchmark set [19].

```
1 P>=1 [ F "elected" ]
2 R{"num_rounds"}=? [ F "elected" ]
```

Listing 8. Properties used in the "leader\_sync" model.

### 5.5 Experiment 3: Extra Scalability

The third experiment evaluates the scalability of the symbolic implementation in JAJAPY 2 when adjusting the initialization of the model hypothesis.

This experiment aims to measure the scalability of JAJAPY 2 under circumstances that are theoretically good for the symbolic implementation. The more repeated values values the transition matrix contains, the sparser the ADD representing it will be. By initializing the transition matrix with a reduced amount of different values, we hope that the symbolic approach might benefit.

For the first experiment, the transition matrix was initialized randomly. For this experiment instead, we only use  $|S|$  different values in the transition matrix. It is expected that this improves the speed of each iteration of the BW algorithm, as it reduces the number of unique computations necessary for the symbolic implementation.

## 6 RESULTS

In this section, we present the results of our experiments, which are divided into two main parts: the first part focuses on the scalability of JAJAPY and CUPAAL in terms of time and scalability, while the second part evaluates the accuracy of both tools.

The experiments were conducted on a machine with the specifications and environment listed in section A.

### 6.1 Scalability

These results are the time taken to train a model, based on two parameters: the number of states, and the length of the observations in the model increasing.

The results for the leader sync model are displayed in Table 1 and Figure 3, and show the time it takes to train a model, given the number of states and observation length. Only the training time is considered; the initialization of the programs is not a factor in these numbers.

In Figure 3, simple planes are fit with linear regression from the data in Table 1. This is not an attempt to say anything definitive about the degrees of the scaling, but instead to show the generally observable trend.

Contrary to our expectations, the data does not show a clear difference in the time taken to train the leader sync model between JAJAPY and CUPAAL for DTMCs.

For very small models, the running time does not matter too much, but we observe an initial overhead related to JAJAPY. This is likely related to the general consensus that Python is a slower language than C in general.

Generally, more states mean longer running time, but interestingly, variations with similar number of states may

TABLE 1  
Leader Sync model variations in training time in seconds.

model	states	length	jajapy (s)	cupaal (s)
3.2	26	25	1.38	0.26
3.2	26	50	1.95	0.14
3.2	26	100	4.09	0.23
3.3	69	25	7.95	2.46
3.3	69	50	11.20	1.59
3.3	69	100	19.65	1.75
3.4	147	25	27.10	8.54
3.4	147	50	42.57	9.20
3.4	147	100	84.02	9.90
4.2	61	25	15.68	11.18
4.2	61	50	24.87	13.56
4.2	61	100	52.11	11.24
4.3	274	25	194.88	231.28
4.3	274	50	414.30	379.21
4.3	274	100	447.83	117.78
4.4	812	25	1846.68	3324.83
4.4	812	50	2290.28	1848.44
4.4	812	100	5652.14	3447.56
5.2	141	25	95.59	104.71
5.2	141	50	342.05	553.66
5.2	141	100	798.73	982.97
5.3	1050	25	4586.86	10906.91
5.3	1050	50	7791.95	10405.75
5.3	1050	100	9821.74	5992.51

have very different training times. The most obvious example is the 3.4 and 5.2 models, with 147 and 141 states respectively. The 5.2 model is much slower, especially in CUPAAL, showing a  $\sim 10$  times increase in training time, despite having slightly fewer states.

Initially, we only had data for observations of length 25, and the data under those conditions suggested that JAJAPY scaled quite a bit better than CUPAAL.

To explore this behaviour, we extended the experiment to contain data for observations of different lengths, and now our observations are more in line with our expectations. JAJAPY gets slower at a pace roughly linear with the length of the observations; doubling the observation length doubles the run time of JAJAPY. This is not the case for CUPAAL, where we do not see any particular increase in running time as the observation length increases.

In fact, looking at Figure 4 and Figure 5, the CUPAAL runtimes look a little strange.

### 6.2 Accuracy

This section will showcase the results from the accuracy experiment. Table Table 2 displays that while time between CUPAAL and JAJAPY differs depending on the size of the model, the final results are practically equal.

The experiment makes use of the `Leader Sync` model, which is a model tool used to select a new leader for a given protocol.

For this experiment the thing we were interested how similar CUPAAL was to the original model. We did this by checking the model of how long time it took for the original model to select a new leader on average, measured in number of average rounds.

The property looks like this in PRISM:

```
1 R{"rounds"}=? [ F "elected" ]
```

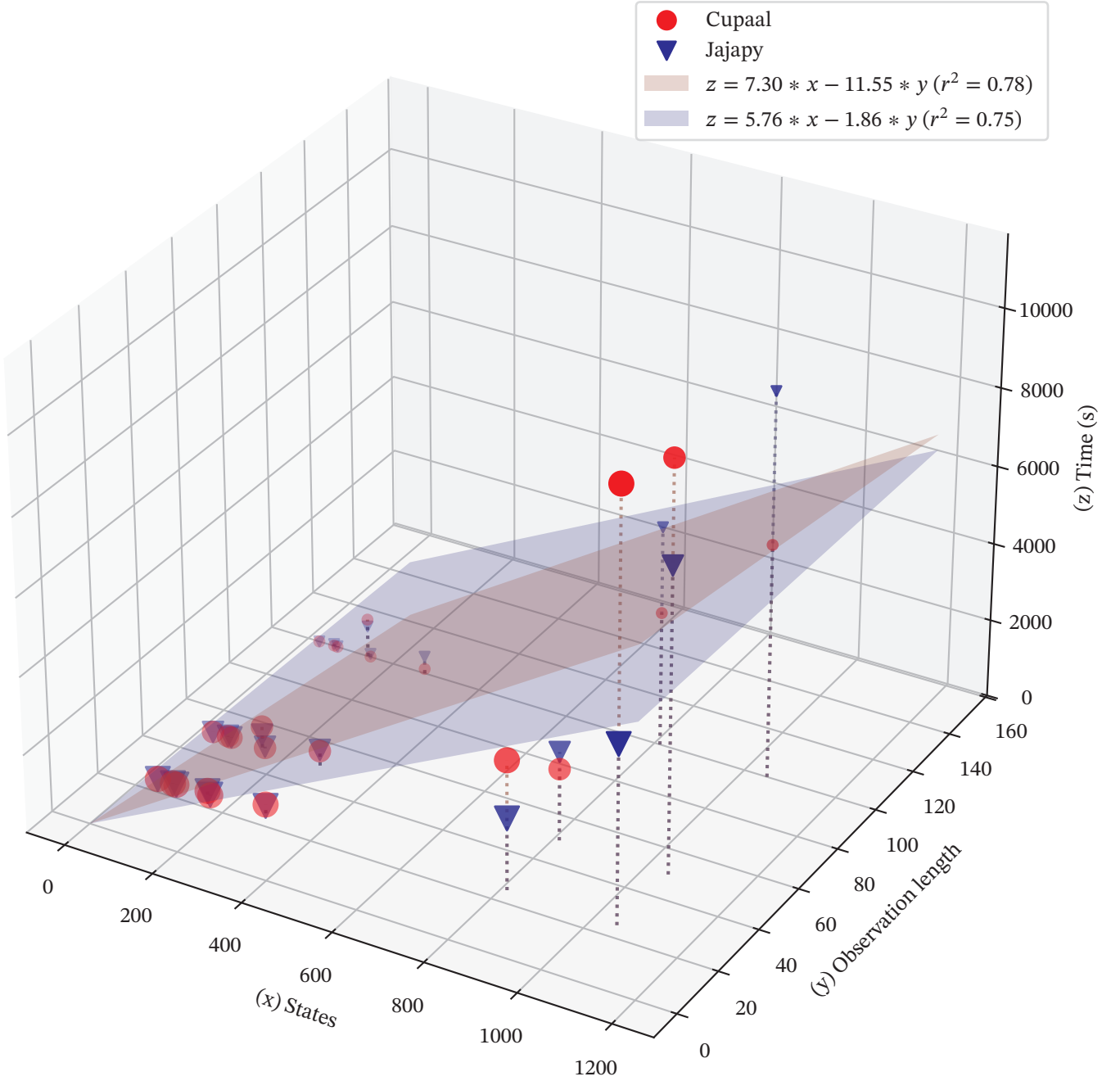


Fig. 3. Plot of the run time of JAJAPY and CUPAAL for the leader sync models, given the number of states and the length of the observations. The planes are linear regression fits to indicate the directions of the trends for the datapoints of similar color.

For model 3.2 i took on average 1.33 rounds, and for both CUPAAL and JAJAPY it took 1.28 rounds. All the models true values can be found in [20]. Both CUPAAL and JAJAPY have identical values for the rounds, hereby showing that they both are equally good at learning a model based on the training set. Importantly, the models learned by CUPAAL and JAJAPY are the same models.

The column  $\phi$  displays the relative error between CUPAAL and the true value. This value is calculated with  $\frac{r-e}{e}$  where  $r$  is the value from CUPAAL and  $e$  is the expected value from the true model. Across all the different models and their variations, both methods come to identical results.

From these results we can gather that CUPAAL is a successful implementation of the BW algorithm.

### 6.3 Extra Scalability

This section will cover the second experiment done for comparing CUPAAL and JAJAPY. This experiment explores the effect of random and semi-random initial model parameters, as we expect repeated values to be highly beneficial for CUPAAL's implementation.

Table 3 is the table that compares CUPAAL to JAJAPY, and the impact of random and semi-random initialization, in regards to the time needed to learn the model.

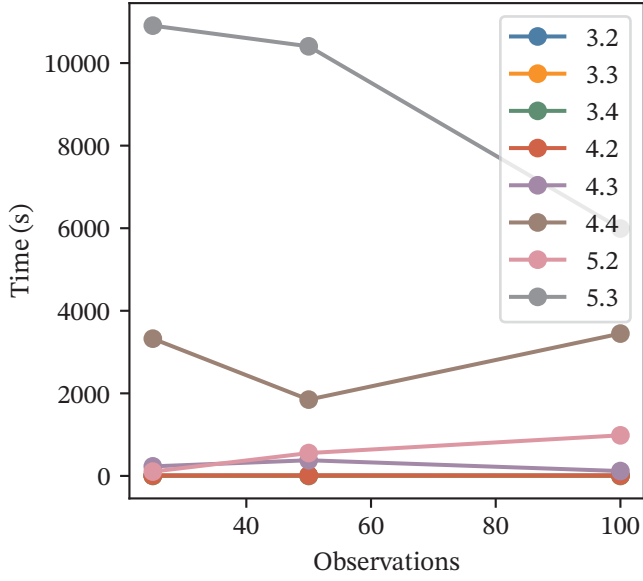


Fig. 4. CUPAAL runtimes with increasing observation length.

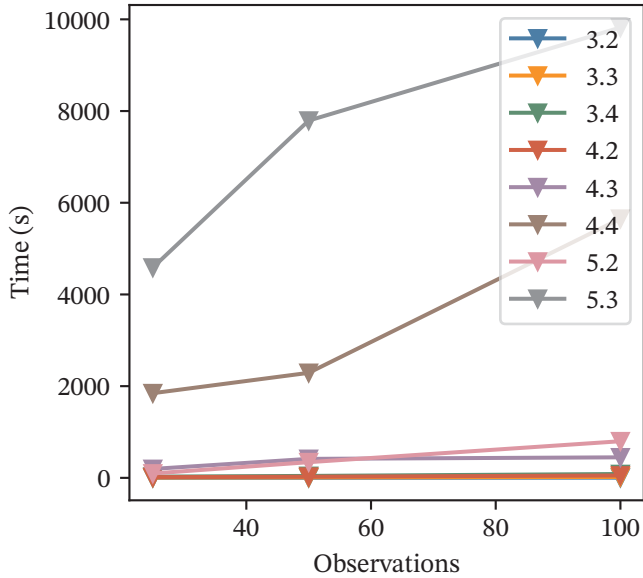


Fig. 5. JAJAPY runtimes with increasing observation length.

Looking at the columns `rand-ja` and `semi-ja` which show the impact of random and semi random initialization for JAJAPY, there is no major impact between the two. But when looking at the differences for CUPAAL it is clear that the semi-random approach is generally faster than the completely random, this becomes especially pronounced as the number of states for the model increase.

We expected this to be the case, as with the number of observations meant more repeated values, the semi random initialization had a similar effect. This seems to partially alleviate the poor scaling CUPAAL suffers from as the number of model states increase.

Table 4, displays three models with a varying number of states and observation sequences. This table is used to highlight that the different methods of initialising model parameters, has an impact of the number of iterations needed,

TABLE 2  
Leader sync model comparison of the average number of rounds inside.

$\mathcal{M}$	$ O $	rounds	JAJAPY	CUPAAL	$\Delta$	$\phi$
3.2	25	1.33	1.28	1.28	0.00	0.04
3.2	75	1.33	1.30	1.30	0.00	0.02
3.2	100	1.33	1.28	1.28	0.00	0.04
3.3	25	1.12	1.23	1.23	0.00	0.09
3.3	50	1.12	1.11	1.11	0.00	0.01
3.3	100	1.12	1.13	1.13	0.00	0.00
3.4	25	1.07	1.08	1.08	0.00	0.01
3.4	50	1.07	1.09	1.09	0.00	0.02
3.4	100	1.07	1.11	1.11	0.00	0.04
4.2	25	2.00	2.11	2.12	-0.00	0.06
4.2	50	2.00	2.16	2.16	0.00	0.08
4.2	100	2.00	2.00	2.00	0.00	0.00
4.3	25	1.35	1.37	1.37	0.00	0.01
4.3	50	1.35	1.28	1.28	0.00	0.05
4.3	100	1.35	1.25	1.25	0.00	0.07
4.4	25	1.19	1.25	1.25	0.00	0.05
4.4	50	1.19	1.17	1.17	0.00	0.01
4.4	100	1.19	1.27	1.27	0.00	0.07
5.2	25	3.20	3.27	3.27	-0.00	0.02
5.2	50	3.20	3.06	3.06	-0.00	0.04
5.2	100	3.20	3.50	3.50	0.00	0.09
5.3	25	1.35	1.32	1.32	0.00	0.02
5.3	50	1.35	1.27	1.27	0.00	0.06
5.3	100	1.35	1.33	1.33	0.00	0.01

TABLE 3  
Leader Sync model variations in training time with random and semi-random initial values.

$\mathcal{M}$	$ S $	$ O $	rand-ja	rand-cup	semi-ja	semi-cup
4.2	61	25	15.68	11.12	13.26	7.87
4.2	61	50	24.87	13.56	28.32	12.54
4.2	61	75	36.02	9.72	38.62	10.54
4.2	61	100	52.11	11.24	53.75	10.95
4.3	274	25	194.88	231.28	190.65	194.04
4.3	274	50	414.30	379.21	414.95	308.81
4.3	274	75	476.68	232.21	486.04	206.67
4.3	274	100	447.83	117.78	441.40	118.17
4.4	812	25	1846.67	3324.83	1793.90	3087.66
4.4	812	50	2290.28	1848.44	2239.18	1526.81
4.4	812	75	3017.72	1597.78	3177.81	1512.33
4.4	812	100	5652.14	3447.56	5586.23	2959.75

but not showing a clear strength in either method.

This is to be expected, as depending on how close or far off, the values are to the learned model, has an impact on the number of iterations needed. But it also showcases that there is no impact on the loglikelihood, meaning that no matter the method used the model learned is still equally close to the correct model.

Figures Figure 6 Figure 7 Figure 8 give a visual representation of how CUPAAL compares to JAJAPY based on the observation counts while using both random and semi-random initialization.

These graphs showcase the general trend of CUPAAL performing slightly better with a semi-random initialization, where for JAJAPY we observe there is no clear tendency for what performs best.

Table 5 shows the seconds per iteration to compute, for CUPAAL and JAJAPY for both random and semi-random initialization. JAJAPY is not noticeably affected by either method, giving a gain of about 1% when using the semi-random initialization over the random approach, this can be explained

TABLE 4  
Leader Sync model variations in loglikelihood for random and semi-random initial values.

$\mathcal{M}$	$ S $	$ O $	iter(rand)	iter(semi)	$\ell$ rand	$\ell$ semi
4.2	61	25	18	16	-140.41	-140.41
4.2	61	50	17	20	-149.13	-149.13
4.2	61	75	17	18	-117.80	-117.80
4.2	61	100	17	18	-138.63	-138.63
4.3	274	25	18	18	-79.92	-79.92
4.3	274	50	18	18	-67.24	-67.25
4.3	274	75	18	18	-65.71	-65.71
4.3	274	100	18	18	-62.55	-62.55
4.4	812	25	18	18	-62.55	-62.55
4.4	812	50	18	18	-48.49	-48.49
4.4	812	75	17	18	-52.26	-52.26
4.4	812	100	18	18	-65.71	-65.71

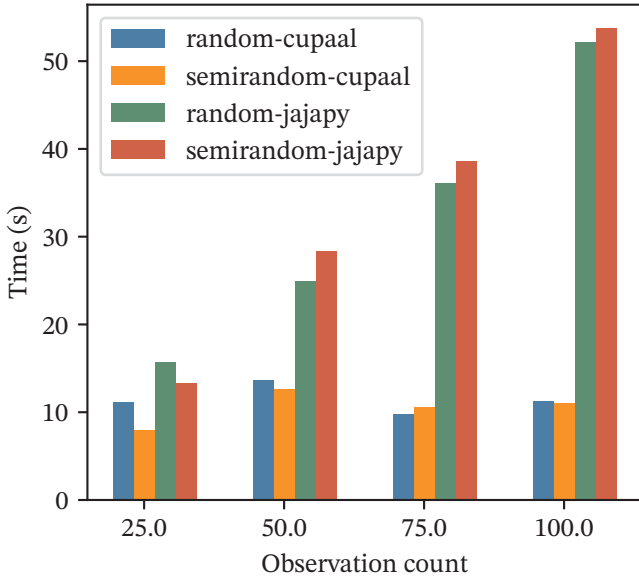


Fig. 6. Model 4.2 - Runtime for random and semi-random

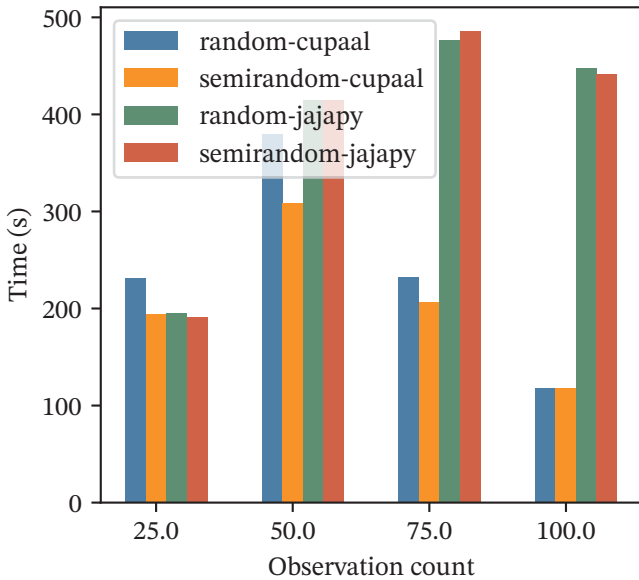


Fig. 7. Model 4.3 - Runtime for random and semi-random

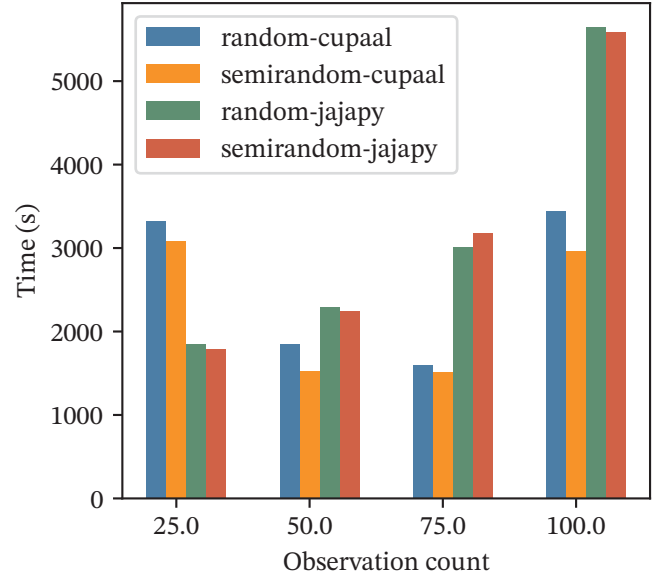


Fig. 8. Model 4.4 - Runtime for random and semi-random

by coincidence as the differences is negligible. CUPAAL does have a slight gain when using the semi-random approach, showcasing a gain of  $\sim 11\%$  less time needed pr. iteration compared to the completely random approach.

These results seem to indicate a gain for CUPAAL when there are repeated values for the initialization. This lines up well with our hypothesis that CUPAAL performs better when it can leverage repeated values for it's ADD structure.

## 7 DISCUSSION

This section discusses the results presented in section 6 and reflects on the performance of CUPAAL compared to JAJAPY.

The results from the section 5 show that CUPAAL outperforms JAJAPY in terms of run time as the observation length increases. For models with few states, the run time is similar, but as the number of states increases, the differences become more pronounced. This indicates that CUPAAL is a better choice for larger observation sequences, but JAJAPY remains a better choice for large models.

The experiment is conducted using a single model, `leader_sync`, which may not provide a comprehensive view of the performance across different scenarios. A more effective approach would have been to utilize multiple models with varying characteristics and compare their performance. This would allow for a more comprehensive overview of CUPAAL's performance compared to JAJAPY. Other models that were considered were the `NAND` and `BRP` models.

A broader variety of models would provide a clearer insight into CUPAAL and thereby display the strengths and weaknesses of CUPAAL.

The model could also have utilized a greater number of states and observations; currently, the largest model contains  $\sim 1,050$  states and observation sequences of length 10.

Larger models were considered, but it was determined that the largest model used was sufficient, given the time required for parameter estimation.

We ran the experiments in a Docker container, which has a maximum memory capacity of 16 GB. This limitation also



TABLE 5  
Experiment three results

$\mathcal{M}$	$ O $	i ran	i semi	s ran-ja	s ran-cup	s semi-ja	s semi-cup	$\ell$ ran	$\ell$ semi	s/i ran-ja	s/i semi ja	s/i ran-cup	s/i semi-cup	ja
4.2	25	18	16	15.68	11.12	13.26	7.87	-140.41	-140.41	0.87	0.83	0.62	0.49	
4.2	50	17	20	24.87	13.56	28.32	12.54	-149.13	-149.13	1.46	1.42	0.80	0.63	
4.2	75	17	18	36.02	9.72	38.62	10.54	-117.80	-117.80	2.12	2.15	0.57	0.59	
4.2	100	17	18	52.11	11.24	53.75	10.95	-138.63	-138.63	3.07	2.99	0.66	0.61	
4.3	25	18	18	194.88	231.28	190.65	194.04	-79.92	-79.92	10.83	10.59	12.85	10.78	
4.3	50	18	18	414.30	379.21	414.95	308.81	-67.24	-67.25	23.02	23.05	21.07	17.16	
4.3	75	18	18	476.68	232.21	486.04	206.67	-65.71	-65.71	26.48	27.00	12.90	11.48	
4.3	100	18	18	447.83	117.78	441.40	118.17	-62.55	-62.55	24.88	24.52	6.54	6.57	
4.4	25	18	18	1846.67	3324.83	1793.90	3087.66	-62.55	-62.55	102.59	99.66	184.71	171.54	
4.4	50	18	18	2290.28	1848.44	2239.18	1526.81	-48.49	-48.49	127.24	124.40	102.69	84.82	
4.4	75	17	18	3017.72	1597.78	3177.81	1512.33	-52.26	-52.26	177.51	176.54	93.99	84.02	
4.4	100	18	18	5652.14	3447.56	5586.23	2959.75	-65.71	-65.71	314.01	310.35	191.53	164.43	

affected the experiments, as we sometimes ran out of memory when experimenting with larger models.

Some models contain a far greater amount of states, which could be further explored.

In subsection 5.5, the values for the initial model values are not entirely random. Instead, they are designed to have repeated values. This was done to display the known strengths of CUPAAL.

This skews the model to favor CUPAAL, as the ADDstructure benefits from repeated values and, therefore, will display results that indicate CUPAAL as the stronger implementation.

This was done purely to research what was believed to be a strength of CUPAAL and to further the discussion on when CUPAAL is a good option to use over other tools, such as JAJAPY.

## 7.1 Implementation Discussion

CUPAAL displays clear benefits when working with repeated values, but in general, it struggles to compete with JAJAPY. Previous work indicated that CUPAAL overall was a stronger implementation, but with an entirely symbolic implementation, some potential bottlenecks have been discovered.

Specifically in the update step of the BW algorithm, as when working with ADDs for just the  $\alpha$  and  $\beta$  steps, CUPAAL performed very well.

This suggests that there may be issues when updating the values when using ADDs. To further research this topic, a hybrid implementation could be provided. This implementation would utilize ADDs when calculating  $\alpha$  and  $\beta$  and then employ a recursive approach when updating values. An implementation like this would require much conversation between matrices and ADDs, but comparing a fully symbolic, a recursive, and a hybrid approach would give further insight into what CUPAAL struggles with.

For now, CUPAAL only measures the time taken to compute the BW algorithm, but an interesting metric to compare would be the memory used. If CUPAAL was discovered to require less memory than JAJAPY, even with more time needed for larger models, it could be a better choice. However, without a memory metric to compare, the decision can only be made based on the time required for computing BW.

To allow CUPAAL and JAJAPY to work together, Pybind11 was used to create bindings between the two. This decision was made without exploring other options. Therefore, it might not

be the best-suited tool. For now, bindings that worked were sufficient for the current iteration of CUPAAL, but further consideration should be given to whether a better tool exists.

The library used to manipulate ADDs was CuDD, as it was what previous work had built upon. A discussion at the time was also whether this is the best tool for the job, as other tools such as Sylvan exist. This discussion remains relevant and worth exploring.

CUPAAL is designed for the BW algorithm, but it is worth exploring other algorithms that could benefit from a symbolic implementation. An algorithm that could be explored would be the Viterbi algorithm. By exploring other algorithms, the general benefits of using a symbolic approach can be better understood.

## 7.2 Future Work

This section will discuss areas that might be worth exploring in future work.

CUPAAL only utilizes a single core. This is not an issue when comparing it to JAJAPY, as it can be limited to using only a single core. However, improving CUPAAL to support multiple cores could be a worthwhile direction to explore, as it could provide a significant performance increase.

In an observation sequence, observations are grouped if they are identical, meaning that when computing these observations, we can factor in the number of identical observations and only compute one of them.

Expanding upon this idea involves making use of prefixes and suffixes to enhance observations. Many observations may not be entirely identical, but they could share a significant number of labels. To leverage this, prefixes and suffixes of observations could be considered and grouped as is done currently. Given that an observation sequence contains many observations that share prefixes and suffixes of labels, the gain could prove to be significant.

In the update step of BW in CUPAAL, consideration is not made to check if the model worked on is an MC. This might be worth adding, as in these cases, unnecessary computation is made, as MCs do not require the Emission function to be updated. This is a minor consideration, as these values are ignored after they are computed, but it could be worth implementing.

## 8 CONCLUSION

In this work, we present a symbolic implementation of the Baum-Welch algorithm for both HMMs and MCs, leveraging ADDs to replace traditional matrix and recursive representations. By reformulating the BW algorithm using compact and canonical ADD structures, our approach efficiently handles both the stochastic emissions of HMMs and the deterministic emissions of MCs, enabling scalable parameter learning across model types.

We extend the BW algorithm to support learning from multiple observation sequences. Based on a matrix-derived aggregation of expectations, we implement the corresponding update steps symbolically using ADDs, eliminating the need for recursive or dense matrix computations while retaining the theoretical correctness of the original BW method.

To make this approach practical, we integrate the symbolic implementation into the JAJAPY library, resulting in JAJAPY 2. Through Pybind11 bindings, the C++ backend of CuPAAL is exposed to Python, allowing users to switch seamlessly between traditional and symbolic learning modes without disrupting existing workflows.

Our experimental evaluation using the "leader\_sync" model from the QComp benchmark demonstrates that the symbolic implementation in CuPAAL achieves significant runtime improvements over JAJAPY's original recursive method, especially in scenarios involving long observation sequences or models with structural redundancy. Accuracy remains unaffected, with both implementations converging to equivalent log-likelihoods and parameter estimates.

These findings underscore the potential of symbolic methods based on ADDs for large-scale probabilistic model learning. By exploiting structure and sparsity, symbolic techniques enable efficient manipulation of high-dimensional models, offering promising applications in domains such as formal verification, machine learning, and systems biology. Future work may explore a hybrid implementation, parallelization, and extensions of CuPAAL to other model types such as Markov Decision Processes (MDPs) and CTMCs.

## ACRONYMS

ADD	Algebraic Decision Diagram. 1, 2, 5, 6, 9, 12–14
BDD	Binary Decision Diagram. 4, 5
BW	Baum-Welch. 1–4, 6–10, 13, 14
CTMC	Continuous Time Markov Chain. 5, 14
CuDD	Colorado University Decision Diagram. 5, 13
DTMC	Discrete Time Markov Chain. 5, 7, 9
HMM	Hidden Markov Model. 1–5, 7, 14
i.i.d.	independently identically distributed. 4
MC	Markov Chain. 1–4, 7, 8, 13, 14
MDP	Markov Decision Process. 14

## REFERENCES

- [1] R. S. Chavan and G. S. Sable, "An overview of speech recognition using hmm," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 6, pp. 233–238, 2013.
- [2] F. Ciocchetta and J. Hillston, "Bio-pepa: A framework for the modelling and analysis of biological systems," *Theoretical Computer Science*, vol. 410, no. 33–34, pp. 3065–3084, 2009.
- [3] R. S. Mamon and R. J. Elliott, *Hidden Markov models in finance*. Springer, 2007, vol. 4.
- [4] R. Reynouard, A. Ingólfssdóttir, and G. Bacci, "Jajapy: A Learning Library for Stochastic Models," in *Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 20–22, 2023, Proceedings*, N. Jansen and M. Tribastone, Eds., ser. Lecture Notes in Computer Science, vol. 14287, Springer, 2023, pp. 30–46. DOI: 10.1007/978-3-031-43835-6\\_3.
- [5] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruiters, "The quantitative verification benchmark set," in *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Proceedings, Part I*, T. Vojnar and L. Zhang, Eds., ser. Lecture Notes in Computer Science, vol. 11427, Springer, 2019, pp. 344–350. DOI: 10.1007/978-3-030-17462-0\\_20.
- [6] L. E. Baum, T. Petrie, G. Soules, and N. Weiss, "A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains," *The Annals of Mathematical Statistics*, vol. 41, no. 1, pp. 164–171, 1970. DOI: 10.1214/aoms/1177697196.
- [7] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989, ISSN: 1558-2256. DOI: 10.1109/5.18626.
- [8] G. Bacci, A. Ingólfssdóttir, K. G. Larsen, and R. Reynouard, "Active learning of markov decision processes using baum welch algorithm," in *20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, Pasadena, CA, USA, December 13–16, 2021*, M. A. Wani, I. K. Sethi, W. Shi, G. Qu, D. S. Raicu, and R. Jin, Eds., IEEE, 2021, pp. 1203–1208. DOI: 10.1109/ICMLA52953.2021.00195.
- [9] G. Bacci, A. Ingólfssdóttir, K. G. Larsen, and R. Reynouard, "An MM algorithm to estimate parameters in continuous-time markov chains," in *Quantitative Evaluation of Systems - 20th International Conference, QEST 2023, Antwerp, Belgium, September 20–22, 2023, Proceedings*, N. Jansen and M. Tribastone, Eds., ser. Lecture Notes in Computer Science, vol. 14287, Springer, 2023, pp. 82–100. DOI: 10.1007/978-3-031-43835-6\\_6.
- [10] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite state markov chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–1563, 1966, ISSN: 00034851.
- [11] R. Reynouard et al., "On learning stochastic models: From theory to practice," 2024.
- [12] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

- [13] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, “Algebraic decision diagrams and their applications,” *Formal methods in system design*, vol. 10, pp. 171–206, 1997.
- [14] M. Kwiatkowska, G. Norman, and D. Parker, “Probabilistic symbolic model checking with prism: A hybrid approach,” *International journal on software tools for technology transfer*, vol. 6, no. 2, pp. 128–142, 2004.
- [15] C. Baier, E. M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, “Symbolic model checking for probabilistic processes,” in *Automata, Languages and Programming: 24th International Colloquium, ICALP’97 Bologna, Italy, July 7–11, 1997 Proceedings 24*, Springer, 1997, pp. 430–440.
- [16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.
- [17] F. Somenzi, “Cudd: Cu decision diagram package,” *Public Software, University of Colorado*, 1997.
- [18] W. Jakob, *Pybind11*, 2025.
- [19] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters, “The quantitative verification benchmark set,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 344–350.
- [20] M. Kwiatkowska, G. Norman, and D. Parker, “The prism benchmark suite,” in *2012 Ninth International Conference on Quantitative Evaluation of Systems*, 2012, pp. 203–204. DOI: 10.1109/QEST.2012.14.

TABLE 7  
Python environment

Requirement	Version
Python	3.12.3
Jajapy	0.10.8
CuPAAL	0.1.0
numpy	1.26.0
pandas	2.2.3
scipy	1.11.2
sympy	1.12.0
matplotlib	3.8.1
alive-progress	3.1.4
pybind11 global	2.13.6

## APPENDIX A MACHINE SPECIFICATIONS

The specifications of the machine used for the experiments are listed in Table 6. The image used for the experiments is based on the movesrwth/stormpy image (version 1.9.0), which is a Docker image that contains the necessary dependencies for running JAJAPY<sup>1</sup>. We add the dependencies for CuPAAL to the image, which are listed in Table 7. For full details see the github repository for CuPAAL.

TABLE 6  
Machine specifications

Specification	Value
CPU	AMD Ryzen 5 3600
RAM	64 GB DDR4
OS	Windows 11 Pro
Docker	4.40.0

### A.1 Python Environment

This section describes the Python environment used for the experiments. The Python version and the versions of the libraries used are listed in Table 7.

## APPENDIX B CHEATSHEET

If something is represented with a greek letter, it is something we calculate.

1. <https://hub.docker.com/r/movesrwth/stormpy>

TABLE 8  
Symbol table.

Symbol	Meaning
$\mathbb{R}$	Real numbers
$\mathbb{B}$	Boolean domain
$\mathbb{Q}$	Rational numbers
$\mathbb{N}$	Natural numbers
$\mathcal{M}$	Markov Model
$s \in S$	States
$l \in L$	Labels
$o \in O \in \mathcal{O}$	Observations
$t \in T$	Time steps
$\mathbf{1}$	Column vector of ones
$\pi$	Initial distribution
$\tau$	Transition function
$\omega$	Emission function
$\alpha$	Forward probabilities
$\beta$	Backward probabilities
$\gamma$	State probabilities given O
$\xi$	Transition probabilities given O
$\lambda = (\pi, \tau, \omega)$	Model Parameters
$\mu$	Mean
$\sigma$	Standard deviation
$\theta = (\mu, \sigma^2)$	Parameters of a distribution
$P(\mathcal{O}; \lambda)$	Probability of $\mathcal{O}$ given $\lambda$
$\ell(\lambda; \mathcal{O})$	Log likelihood of $\lambda$ under $\mathcal{O}$
$\cdot$	Scalar product
$\odot$	Hadamard product
$\otimes$	Kronecker product
$\oslash$	Hadamard division
$\bullet$	Transposed Khatri-Rao product