Baum-Welch Algorithm for Markov Models Using Algebraic Decision Diagrams

Sebastian Aaholm*, Lars Emanuel Hansen†, Daniel Runge Petersen‡,

Abstract—This is a placeholder abstract. The whole template is used in semester projects at Aalborg University (AAU).

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 Algebraic Decision Diagrams (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 Hidden Markov Models (HMMs) and Markov Chains (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 Baum-Welch algorithm (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 learning 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 theses 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 observed traces. Once learned, these models can be directly exported for formal analysis in tools such as Storm and 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.

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

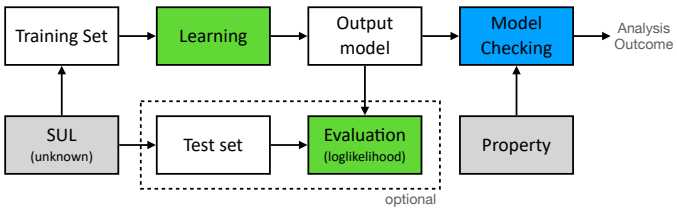

Fig. 1. Modeling and verification workflow using JAJAPY. Phases involving JAJAPY are highlighted in green, while the blue phase represents verification using STORM or PRISM.

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 depend 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.

```
import jajapy
training_set = jajapy.loadSet("Path/to/data")
type(training_set) # list

learned_model = jajapy.BW().fit(training_set,
    nb_states=10)
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.

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 Baum-Welch algorithm, 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 $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi)$, where:*

- *S is a finite set of states.*
- *$\mathcal{L}$ is a finite set of labels.*
- *$\ell : S \to D(\mathcal{L})$ is the emission function.*
- *$\tau : S \to D(S)$ 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 ℓ describes the probability of emitting a label given a state. The transition function τ describes the probability of transitioning from one state to another. The initial distribution π 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* $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi)$ *identical to the HMM structure above except that the emission function is* deterministic*: for every* $s \in S$ *there is a single label* $l = \ell(s)$ *emitted with probability 1.*

In other words, the emission function ℓ 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* $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi)$*, there exists a trace-equivalent HMM* $\mathcal{M}' = (S', \mathcal{L}', \ell', \tau', \pi')$*, where:*

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

The only difference in this case is the structure of the emission functions.

3.4 Baum-Welch Algorithm

The Baum-Welch algorithm 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 π, ℓ, and τ 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 a HMM as matrices for computational efficiency.

They are defined as follows:

- $\boldsymbol{\pi}$ 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 $|S|$.
- $\boldsymbol{P}$ is the transition matrix, where $P_{ij} = \tau(s_i)(s_j)$ is the probability of transitioning from state s_i to state s_j, this is a square matrix of size $|S| \times |S|$.
- $\boldsymbol{\omega}$ is the emission matrix, where $\omega_{ij} = \ell(s_i)(l_j)$ is the probability of emitting label l_j given state s_i, this is a matrix of size $|S| \times |\mathcal{L}|$.

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 $\mathbf{o} = o_1 \ldots o_T$ be a sequence of observations, where each $o_t \in \mathcal{L}$, the observation sequence has the length T. Suppose $\mathcal{M}$ has n states, i.e., $S = s_1, \ldots, s_n$, with parameters represented as follows:

- $\boldsymbol{\pi} \in [0,1]^n$ is the initial state distribution column vector.
- $\boldsymbol{P} \in [0,1]^{n \times n}$ is the transition probability matrix.
- $\boldsymbol{\omega} \in [0,1]^{n \times |\mathcal{L}|}$ 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 $\boldsymbol{\omega}(t)$ be the column vector of emission probabilities for label o_t for each state, and $\bigcirc$ the Hadamard (element-wise) product.

```
FORWARD-ALGORITHM
1  α(1) = ω(1) ○ π
2  for t = 2 to T
3        α(t) = ω(t) ○ (P^⊤ α(t − 1))

BACKWARD-ALGORITHM
1  β(T) = 1
2  for t = T − 1 to 1
3        β(t) = P (β(t + 1) ○ ω(t + 1))
```

Listing 2. Computation of the forward and backward coefficients

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

$$\boldsymbol{\gamma}(t) = \boldsymbol{\alpha}(t) \bigcirc \boldsymbol{\beta}(t)) / P[\mathbf{o}|\mathcal{M}]$$
$$\boldsymbol{\xi}(t) = (P[\mathbf{o}|\mathcal{M}] \cdot P) \bigcirc \left(\boldsymbol{\alpha}(t) \otimes (\boldsymbol{\beta}(t+1) \bigcirc \boldsymbol{\omega}(t+1))^\top\right)$$

Here, $\otimes$ is the Kronecker product and the probability $P[\mathbf{o}|\mathcal{M}]$ to observe $\mathbf{o}$ in $\mathcal{M}$ is computed as $\mathbf{1}^\top \boldsymbol{\alpha}(T)$, Here $\top$ describes the transposed vector, i.e., $\mathbf{1}^\top$ is a row vector of ones. We calculate $\boldsymbol{\gamma}(t)$ from $t = 1$ to T and $\boldsymbol{\xi}(t)$ from $t = 1$ to $T - 1$.

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

$$\hat{\boldsymbol{\pi}} = \boldsymbol{\gamma}(1) \tag{1}$$

$$\hat{\boldsymbol{\omega}} = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet (\sum_{t=1}^{T} \boldsymbol{\gamma}_t \otimes \mathbb{1}^T_{[[o_t]]}) \tag{2}$$

$$\hat{P} = (\mathbf{1} \oslash \boldsymbol{\gamma}) \bullet \boldsymbol{\xi} \tag{3}$$

Where $\bullet$ is the transposed Khatri-Rao product (i.e., row-by-row Kronecker product), and $[[o_t]] = ([[o_t = l]])_{l \in \mathcal{L}}$ is the one-hot encoding of the observation o_t, meaning that it is a vector of size $|\mathcal{L}|$ 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 $\boldsymbol{\gamma}$ and $\boldsymbol{\xi}$ are defined as follows:

$$\boldsymbol{\gamma} = \sum_{t=1}^{T} \boldsymbol{\gamma}(t) \text{ and } \boldsymbol{\xi} = \sum_{t=1}^{T-1} \boldsymbol{\xi}(t) \tag{4}$$

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

3.5 Multiple Observation Sequences

Suppose we are given a multiset of observation sequences $\mathcal{O} = O_1, O_2, \ldots, O_{|\mathcal{O}|}$, where each $O_i = (o_{i1}, o_{i2}, \ldots, o_{iT})$ is of length T. The E-step remains unchanged: for each sequence, we compute the corresponding $\boldsymbol{\alpha}(t)$, $\boldsymbol{\beta}(t)$, $\boldsymbol{\gamma}(t)$, and $\boldsymbol{\xi}(t)$ values independently.

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

$$\boldsymbol{\gamma} = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T} \boldsymbol{\gamma}_t \text{ and } \boldsymbol{\xi} = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T-1} \boldsymbol{\xi}_t \tag{5}$$

With these aggregate quantities, the update rules for the initial distribution (see Equation 1) and transition matrix (see Equation 3) 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:

$$\boldsymbol{\omega}_s(o) = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet (\sum_{i=1}^{N} \sum_{t=1}^{T} \boldsymbol{\gamma}_{it} \otimes \mathbb{1}^T_{[[o_t]]}) \tag{6}$$

This mirrors the single-sequence case (see Equation 2) 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 Baum-Welch algorithm 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 $\boldsymbol{\gamma}(t)$ and $\boldsymbol{\xi}(t)$ remain structurally the same, though emission terms are omitted due to determinism.
- The **M-step** updates for the initial distribution $\boldsymbol{\pi}$ and the transition matrix $\boldsymbol{P}$ are unchanged.
- The emission matrix $\boldsymbol{\omega}$ 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 Baum-Welch algorithm 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 applications like dynamic programming, Markov Decision Processs (MDPs), and linear algebraic computations [13].

4 Implementation

This section will provide an overview of different types of Decision Diagrams, how they each are structured, their differences and how they can be converted from one to another.

The different approaches that can be taken for the Baum-Welch algorithm will also be discussed, including the recursive, matrix-based, and ADD-based approaches. The advantages and limitations of each approach will be highlighted.

Finally, the Colorado University Decision Diagram (CuDD) library will be introduced, which is a library for implementing and manipulating BDDs and ADDs.

4.1 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 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 and CTMCs.

In this project, we use the CuDD library 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.

4.3 From Prism to CuPAAL

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 $f: 0,1^n \times 0,1^n \to R$.

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 subsection 4.4.

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

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

4.4 Kronecker Product Implementation

This section will provide an overview of how the Kronecker product is implemented in CuPAAL, using the row and column duplication method mentioned in subsection 4.3.

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 $\hat{B}$

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

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

$$\hat{A} \otimes \hat{B} = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}. \tag{7}$$

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.

Let $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 2 \end{bmatrix}$ 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:

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

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.

4.5 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.

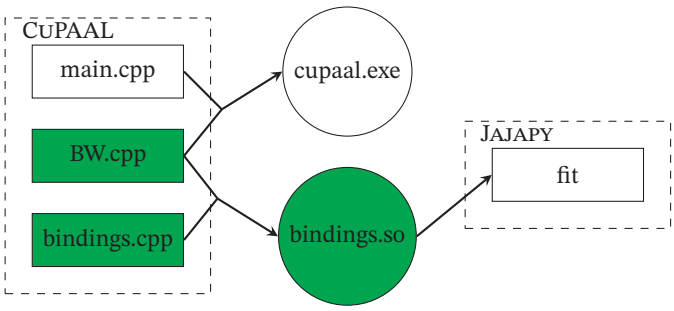

Fig. 2. Architecture of CuPAAL combined with Jajapy.

CuPAAL consists of two main componenets, 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.

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.

We create a C++ bindings file, that uses the BW library from CuPAAL and defines 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.

Listing 3 calculates the Baum-Welch algorithm using Listing 4 from CuPAAL, and returns the results to Jajapy, as a tuple with the relevant values. These being the initial distribution, the transitions and the emissions.

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.

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.

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.

5 Experiments

In this section, we present an empirical evaluation comparing the performance of two implementations of the Baum-Welch algorithm: the original version from Jajapy and the new symbolic implementation introduced in Jajapy 2. For this comparison, we use a selection of discrete-time Markov chains taken from the QComp benchmark set [19].

Our evaluation focuses on two primary criteria: execution time and estimation accuracy.

We designed two experiments to address these aspects:

- **Performance Comparison** — Assessing runtime and accuracy across a variety of models.
- **Scalability Analysis** — Examining how performance is affected as the model size, specifically the number of states, increases.

The goal of these experiments is to answer the following research questions:

- **Question 1**: How does the symbolic implementation of the Baum-Welch algorithm in CuPAAL perform in terms of runtime and accuracy compared to the recursive implementation in Jajapy?
- **Question 2**: How does the runtime performance of CuPAAL scale as the size of the model increases?

5.1 Experimental Setup

All experiments were conducted on a machine equipped with a Ryzen 5 3600 processor, 64 GB of RAM, running Ubuntu Linux.

For each model from the benchmark set, we selected a subset of observable atomic propositions[1] and generated a

1. link to models with description of the chosen observable atomic propositions

```cpp
// Some parameters have been removed for brevity
cupaal_markov_model
    cupaal_bw_symbolic(vector<string>& states,
    vector<string>& labels,
    vector<vector<string>>& observations,
    vector<double>& initial_distribution,
    vector<double>& transitions, vector<double>&
    emissions, int max_iterations = 100, double
    epsilon = 1e-2){
    MarkovModel model(states, labels,
        initial_distribution, transitions,
        emissions, observations);
    cupaal_markov_model model_data;
    chrono::seconds time = chrono::seconds(3600);
    model.baum_welch_multiple_observations(
        max_iterations, epsilon, time);

    // Removed output and result path for brevity
    model_data.initial_distribution =
        model.initial_distribution;
    model_data.transitions = model.transitions;
    model_data.emissions = model.emissions;

    Cudd_Quit(model.manager);
    return model_data;
}
```

Listing 3. C++ bindings file for CuPAAL

```cpp
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.

```
def _bw_symbolic(self, max_iteration = 100, epsilon = 1e-2, outputPath = "", resultPath = ""):
    try:
        import libcupaal_bindings
    except ModuleNotFoundError:
        print("Cannot find module")

    states = [str(i) for i in range (self.h.nb_states)]
    labels = list(set(self.h.labelling))
    observations = []
    for times, sequences in zip(self.training_set.times, self.training_set.sequences):
        for i in range(times):
            observations.append(list(sequences))
    initial_state = self.h.initial_state.tolist()
    transitions = self.h.matrix.flatten().tolist()
    emissions = zeros((len(labels), self.h.nb_states))
    for row in range(len(labels)):
        for col in range(self.h.nb_states):
            if self.h.labelling[col] == labels[row]:
                emissions[row][col] = 1
    emissions = emissions.flatten().tolist()

    cupaal_model = libcupaal_bindings.cupaal_bw_symbolic( states, labels, observations, initial_state,
     ↪  transitions, emissions, max_iteration, epsilon, outputPath, resultPath)
    self.h.initial_state = array(cupaal_model.initial_distribution)
    self.h.matrix = array(cupaal_model.transitions).reshape( self.h.nb_states, self.h.nb_states)
    self.h.emissions = array(cupaal_model.emissions).reshape( len(labels), self.h.nb_states)
    return self.h
```

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

```
# Some parameters have been removed for brevity
def fit(self, output_file: str, output_file_prism:
 ↪  str, epsilon: float, max_it: int, symbolic:
 ↪  bool):
    # Removed preparation and settings number of
     ↪  processes, for brevity
    if symbolic :
        return self._bw_symbolic(max_it, epsilon,
         ↪  output_file, output_file_prism)
    else:
        return self._bw(max_it, pp, epsilon,
         ↪  output_file, output_file_prism,
         ↪  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.

training dataset consisting of 30 observation sequences, each of length 10.

In each case, the output model was configured to match the state size of the original benchmark, and all transition probabilities were treated as parameters to be estimated.

Each training session was allowed to run until either the default convergence threshold of 0.01 was reached or a maximum runtime of 4 hours elapsed. Every experiment was repeated 10 times. During each run, we recorded the runtime, the absolute error ϵ_i for each estimated parameter x_i[2], and the log-likelihood value achieved in the final iteration.

5.2 Experiment 1: Performance Comparison of Implementations

The first experiment is based on the ideas from the experiment conducted in [11]. The models used are shown in Table 1. The experiment evaluate the efficiency and accuracy of the symbolic approach versus the recursive approach. We measure:

- **Runtime Efficiency** - The average time per run.
- **Convergence Speed** - The average number of iterations required.
- **Accuracy** - Measured using log-likelihood and an average error.

TABLE 1
DTMC models

Name	Number of States
Leader_sync	274
Brp	886
Crowds	1145

Table 2 reports the aggregated results of the experiments. The column $|S|$ provides the number of states of the model; the columns "time" and "iter" respectively report the average running time and number of iterations; and the column "ϵ" and "$\log \mathcal{L}$" respectively report the average error of the estimated transition probabilities and the average log-likelihood valued measured w.r.t. the training set.

Model	$\|S\|$	JAJAPY				JAJAPY 2			
		iter	time	$\log \mathcal{L}$	ϵ	iter	time	$\log \mathcal{L}$	ϵ
Leader sync	274	15.6	35.84	-0.00165602	0.35	15.7	24.02	-5.357103	

TABLE 2
Experimental comparison between the original and symbolic implementation of the BW algorithm in JAJAPY.

5.3 Scalability Experiment

The primary objective of this experiment is to evaluate the scalability of the proposed symbolic implementation of the Baum-Welch algorithm in comparison to the recursive implementation in Jajapy. Specifically, we aim to measure the time required to learn DTMCs over the number of states. We measure:

2. The absolute error is defined as $|e - r|$, where e is the estimated value and r is the real value.

- **Runtime efficiency** - The average time per run.

We use the *leader_sync* model, scaling from 26 to 1050 states. This experiment provides insights into how the symbolic approach scales as model complexity increases.

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.

TABLE 3
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

The results for the leader sync model are displayed in Table 3 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 3. 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 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 ~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. Jajapygets 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.

From Figure 3

6.2 Accuracy

7 Improvements

This section outlines the improvements gained by transitioning from the recursive implementation in Jajapy to the symbolic approach in CuPAAL.

As discussed in (Ref to previous section talking about Jajapy), Jajapy uses a recursive implementation of the Baum-Welch algorithm to learn HMMs. In contrast, CuPAAL implements the Baum-Welch algorithm using ADDs. By leveraging ADDs, CuPAAL demonstrates significant improvements over the recursive approach used in Jajapy. The discussion of these improvements is based on the experimental results presented in Section 5.

7.1 Run Time

One of the most notable advantages of CuPAAL over Jajapy is the reduction in run time, particularly for models with a large number of states.

The use of ADDs minimizes redundant computations by merging identical values within the structure. This optimization significantly reduces the computational needs, compared to a recursive implementation. As a result, the run time gap between CuPAAL and Jajapy increases as the number of states grows, making CuPAAL a more scalable solution for large HMMs.

The number of iterations of CuPAAL for each model is also slightly reduced compared to Jajapy.

This advantage is particularly beneficial in scenarios where handling large probability matrices would otherwise lead to excessive computational costs.

7.2 Accuracy

While run time is a key advantage of CuPAAL, it is equally important to assess whether these performance gains come at the cost of accuracy. Since both approaches implement the Baum-Welch algorithm, they are expected to converge to similar model parameters when learning HMMs.

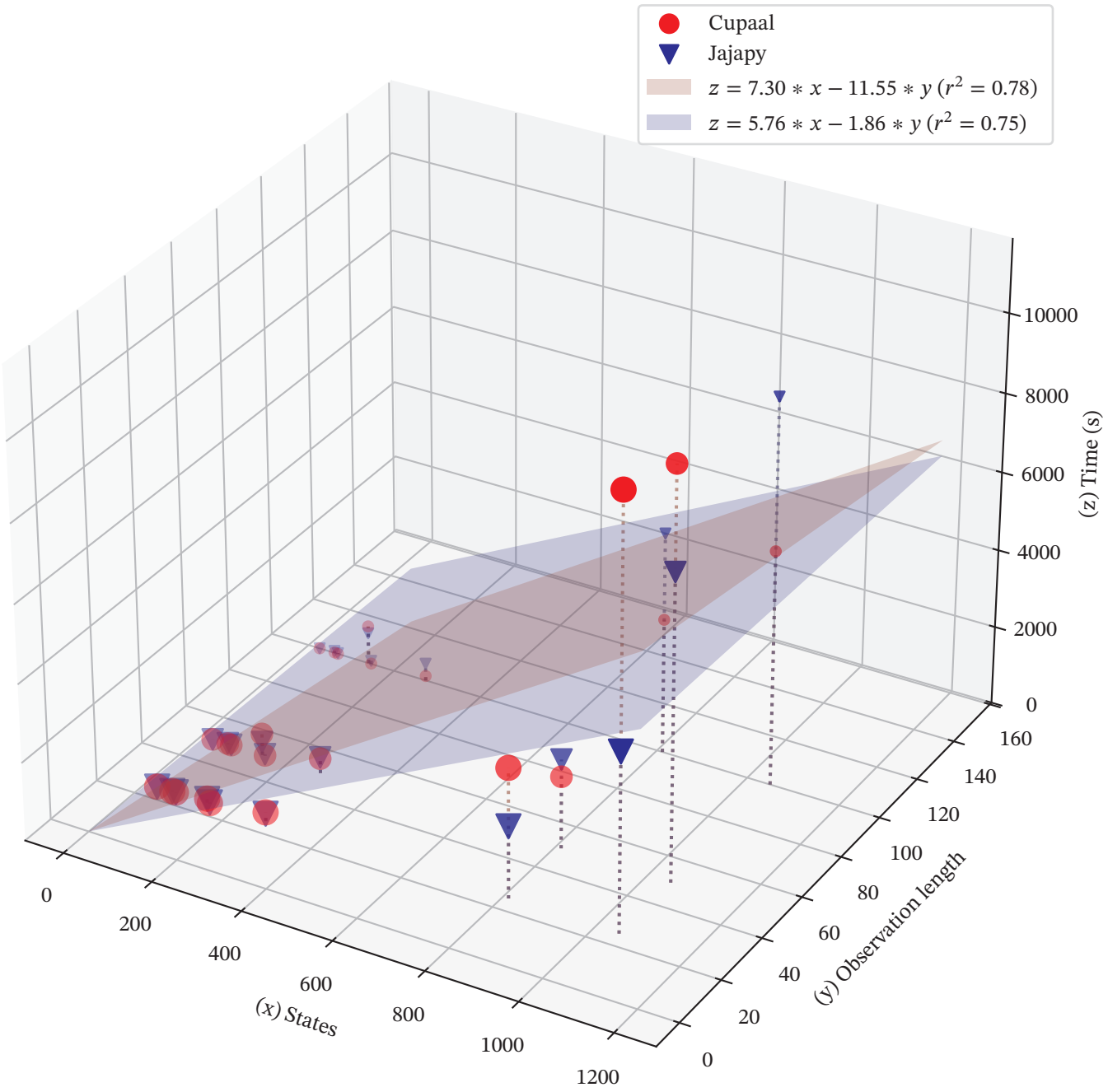

Fig. 3. Plot of the run time of JAJAPY and CUPAAL for the leader sync models, given the number of states and the length of the observations. The planes are linear regression fits to indicate the directions of the trends for the datapoints of similar color.

As shown in Section 5, CuPAAL achieves accuracy comparable to Jajapy across various models. These results can be seen in the values of avg delta and the log-likelihood, where the closer the value is to 0 the better. Displaying that a symbolic implementation does not introduce significant numerical errors, ensuring that the learned transition and emission probabilities remain consistent with those obtained using the recursive approach.

Furthermore, by eliminating redundant calculations, CuPAAL may reduce floating-point errors that typically accumulate in recursive implementations. Importantly, CuPAAL maintains accuracy even as the number of states increases, showcasing that its efficiency improvements do not compromise learning quality. This makes it particularly well-suited for handling large-scale HMMs.

7.3 Implementation

The implementation of CuPAAL has been done in c++, compared to Jajapy which is implemented in Python. This could also be a factor aiding the performance improvement of CuPAAL, as C++ is generally faster at computation compared to Python. This choice of implementation not only improves speed but also ensures that CuPAAL can efficiently handle large models that would be infeasible in Python.

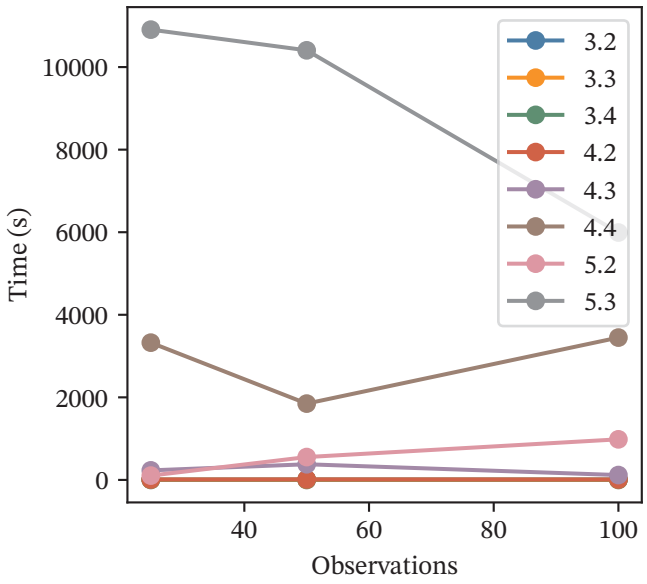

Fig. 4. CuPAAL runtimes with increasing observation length.

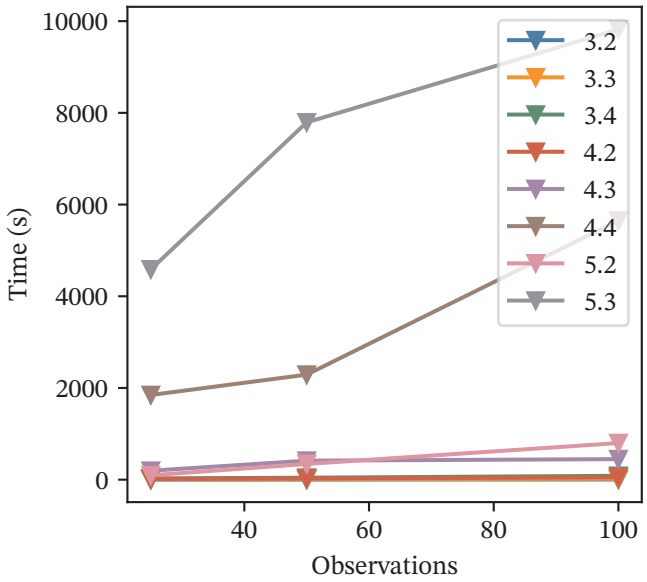

Fig. 5. JAJAPY runtimes with increasing observation length.

7.4 Final improvement overview

The improvements introduced by CuPAAL stem from multiple factors: the adoption of ADDs, optimized run time and a high-performance C++ implementation. These enhancements make CuPAAL a powerful alternative to recursive approaches like Jajapy, particularly when working with large, redundant, and complex HMMs.

8 Discussion

The accuracy of the models learned with the BW algorithm strongly depends on selecting an appropriate size for the output model. However, increasing this size substantially raises the computational cost of each update iteration, both in terms of time and space complexity.

This is because each iteration requires running the forward-backward algorithm on every trace in the training set. In the original implementation, this step was performed using Jajapy models, incurring a cost of $O(n^2 \cdot K)$ in time and $O(n \cdot K)$ in space per iteration, where n is the number of states in the output model and K is the total number of label occurrences in the training set. Moreover, computing the updated transition probabilities from the forward and backward coefficients added an extra $O(n^2 \cdot K)$ overhead in both time and space.

Unsurprisingly, this had a significant impact on the performance of the BW algorithm as the number of states increased.

To address this limitation, CuPAAL introduces a symbolic engine that efficiently handles both the forward-backward computation and the parameter updates.

Acronyms

AAU Aalborg University. 1
ADD Algebraic Decision Diagram. 1, 2, 4, 5, 8, 10

BDD Binary Decision Diagram. 4, 5
BW Baum-Welch algorithm. 1, 2, 6

CTMC Continuous Time Markov Chain. 5
CuDD Colorado University Decision Diagram. 5

DTMC Discrete Time Markov Chain. 5, 8

HMM Hidden Markov Model. 1–4, 8–10

MC Markov Chain. 1–4
MDP Markov Decision Process. 4

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. Ingólfsdóttir, 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. Ruijters, "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. Ingólfsdóttir, 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. Ingólfsdóttir, 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, "Algebric 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.

APPENDIX A
MACHINE SPECIFICATIONS

The specifications of the machine used for the experiments are listed in Table 4. 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[3]. We add the dependencies for CUPAAL to the image, which are listed in Table 5. For full details see the github repository for CUPAAL.

TABLE 4
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 5.

TABLE 5
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 B
CHEATSHEET

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

3. https://hub.docker.com/r/movesrwth/stormpy

TABLE 6
Symbol table.

Symbol	Meaning
$\mathbb{R}$	Real numbers
$\mathbb{Q}$	Rational numbers
$\mathbb{N}$	Natural numbers
$s \in S$	States
$l \in L$	Labels
$\mathcal{M}$	Markov Model
$o \in O \in \mathcal{O}$	Observations
$t \in T$	Time steps
$\top$	Transpose operator
π	Initial distribution
τ	Transition function
ω	Emission function
α	Forward probabilities
β	Backward probabilities
γ	State probabilities
ξ	Transition probabilities
$\lambda = (\pi, \tau, \omega)$	Model Parameters
μ	Mean
σ	Standard deviation
$\theta = (\mu, \sigma^2)$	Parameters of a distribution
$P(\mathcal{O}; \lambda)$	Probability of $\mathcal{O}$ given λ
$\ell(\lambda; \mathcal{O})$	Log likelihood of λ under $\mathcal{O}$