Summary

This thesis introduces a symbolic approach to the Baum-Welch (BW) algorithm using Algebraic Decision Diagrams (ADDs) for efficiently learning parameters of Hidden Markov Models (HMMs) and Markov Chains (MCs). Traditional recursive or matrix-based implementations of BW often suffer from high memory usage and limited scalability. To address this, we propose a novel symbolic implementation built into a C++ library named CuPAAL, which is integrated into the Python JaJaPy library as JaJaPy 2.

Tools like JaJaPy implement BW recursively or through matrix operations. However, these approaches become inefficient for large or repetitive models due to memory constraints and computational redundancy.

The motivation behind CuPAAL is to overcome these issues by leveraging ADDs, data structures that compactly represent numerical functions over discrete variables. ADDs generalize Binary Decision Diagrams (BDDs), allowing them to represent and manipulate probabilities and other numeric values symbolically. This symbolic computation reduces redundancy and memory use, making it feasible to train large-scale probabilistic models.

CuPAAL represents transition, emission, and initial state matrices as ADDs. Matrix operations, including the Kronecker and Hadamard products, are symbolically implemented. This structure is especially beneficial when matrix sparsity or redundancy exists. PRISM models are translated into JaJaPy models and then encoded as ADDs for use in CuPAAL. The main contributions of this work are:

1. Symbolic Reformulation of the BW Algorithm: Each step of the BW algorithm (forward-backward, parameter updates) is redefined in terms of ADD operations using the Colorado University Decision Diagram (CuDD) library.

2. Multi-Sequence Learning Support: The symbolic implementation handles multiple observation sequences for both MCs and HMMs.

3. Integration with JaJaPy: The symbolic implementation is integrated into the JaJaPy library, allowing users to switch between traditional and symbolic learning modes.

4. Empirical Evaluation: Using models from the QComp benchmark (specifically, the leader sync model), experiments demonstrate: Comparable or better runtime performance, especially for models with repeated structural patterns; Accuracy on par with traditional methods; Improved scalability, especially when models are initialized with repeated values (controlled initialization).

The symbolic implementation of the BW algorithm using ADDs proves to be a scalable and efficient alternative to traditional approaches. With JaJaPy 2, users now have access to a flexible tool for probabilistic model learning that maintains accuracy while improving performance for complex and large-scale models.

Baum-Welch Algorithm for Markov Models Using Algebraic Decision Diagrams

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

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 extend on 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.

Index Terms—Algebraic Decision Diagrams, Baum-Welch Algorithm, Hidden Markov Models, Markov Chains, Model Checking

1 Introduction

The Baum-Welch (BW) algorithm is a widely used method for training Markov models in various applications, including speech recognition, bioinformatics, and financial modeling [1–3].

Traditionally, the BW 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 BW 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 BW algorithm implementation has been a significant limitation due to the inherent redundancy in matrix-based representations, which leads to inefficiencies, particularly in terms of time and memory consumption, thereby restricting 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 HMM and MC models by leveraging ADDs:

First, we extend the BW algorithm for these models using symbolic computation, reformulating each algorithm step as operations on ADDs. We leverage the Colorado University Decision Diagram (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 BW 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.

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[1].

Finally, we integrate these CuPAAL Python bindings into JAJAPY as JAJAPY 2, allowing users to run symbolic probabilistic learning algorithms within JAJAPY seamlessly.

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.

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

1. Source code available at: https://github.com/AAU-Dat/CuPAAL

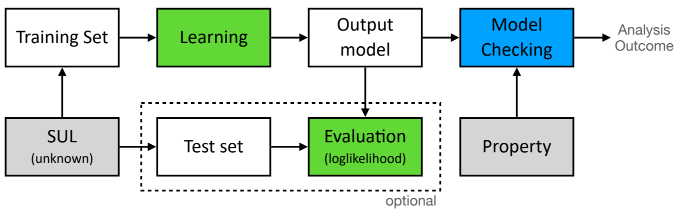

Fig. 1. Modeling and verification workflow using JAJAPY [6]. Phases involving JAJAPY are highlighted in green, while the blue phase represents verification using STORM or PRISM .

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 (MCs). 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 [7, 8], adapted for MCs, Markov Decision Processs (MDPs) [9], and Continuous Time Markov Chains (CTMCs) [10].

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 MC). It 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 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 MC from a training set, starting from a random initial hypothesis, is shown in Listing 1.

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

```python
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 MC from a training set.

2.2 CuPAAL

CUPAAL is a tool developed in C++ that extends the work done in JAJAPY by implementing the BW algorithm with an Algebraic Decision Diagram (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 E-step of the BW algorithm was 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 Hidden Markov Models (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 have implemented the BW algorithm for use with MCs. Given the similarities between these model types, we have 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 Hidden Markov Model (HMM) and a Markov Chain (MC), which are the two main models used in this report, then go on to introduce the Baum-Welch (BW) 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

HMMs were introduced by Baum and Petrie in 1966 [11] 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 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.*
- *$\omega : S \to D(L)$ 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 ω 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 MC, named after Andrei Markov, is a stochastic model widely used in different fields of study [11].

Definition 2 (Markov Chain)**.** *A 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 ω 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 are 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 $P[O \mid \mathcal{M}] = P[O \mid \mathcal{M}']$.

3.4 Baum-Welch Algorithm

The BW 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, ω, 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 via the forward-backward algorithm.
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 [8].

We can represent the parameters of 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 $\lvert S \rvert$.
$\boldsymbol{\tau}$	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 $\lvert S \rvert \times \lvert S \rvert$.
$\boldsymbol{\omega}$	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 $\lvert S \rvert \times \lvert L \rvert$.

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 [7, 12].

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:

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

The forward and backward algorithms are implemented using dynamic programming, as shown in Listing 2. For a

```
FORWARD-ALGORITHM
1  $\boldsymbol{\alpha}(1) = \boldsymbol{\omega}(1) \odot \boldsymbol{\pi}$
2  for $t = 2$ to $T$
3      $\boldsymbol{\alpha}(t) = \boldsymbol{\omega}(t) \odot \left(\boldsymbol{\tau}^\top \boldsymbol{\alpha}(t-1)\right)$

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

Listing 2. Computation of the forward and backward coefficients

given time step t, let $\boldsymbol{\omega}(t)$ be the column vector of emission probabilities for label o_t for each state, and $\odot$ the Hadamard (element-wise) product.

The procedures in Listing 2 compute the column vectors $\boldsymbol{\alpha}(t)$ and $\boldsymbol{\beta}(t) \in [0,1]^n$ for $t = 1 \dots T$ 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:

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

$$\boldsymbol{\xi}(t) = (P[O|\mathcal{M}]^{-1} \cdot \boldsymbol{\tau}) \odot \left(\boldsymbol{\alpha}(t) \otimes (\boldsymbol{\beta}(t+1) \odot \boldsymbol{\omega}(t+1))^\top\right) \tag{2}$$

Here, $\otimes$ is the Kronecker product and the probability $P[O|\mathcal{M}]$ to observe O in $\mathcal{M}$ is computed as $\mathbf{1}^\top\boldsymbol{\alpha}(T)$. 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:

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

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

$$\hat{\boldsymbol{\tau}} = (\mathbf{1} \oslash \boldsymbol{\gamma}) \bullet \boldsymbol{\xi} \tag{5}$$

Where $\bullet$ 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. The $\boldsymbol{\gamma}$ and $\boldsymbol{\xi}$ are defined as follows:

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

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

These update rules form the standard BW algorithm 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 $\boldsymbol{\alpha}$ probabilities in this way:

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

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 $\boldsymbol{\alpha}_i(t)$, $\boldsymbol{\beta}_i(t)$, $\boldsymbol{\gamma}_i(t)$, and $\boldsymbol{\xi}_i(t)$ values independently.

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

$$\boldsymbol{\gamma} = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T} \boldsymbol{\gamma}_i(t) \tag{9}$$

$$\boldsymbol{\xi} = \sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T-1} \boldsymbol{\xi}_i(t) \tag{10}$$

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:

$$\boldsymbol{\omega}_s(o) = (\mathbf{1} \oslash \boldsymbol{\gamma}) \bullet \left(\sum_{i=1}^{|\mathcal{O}|} \sum_{t=1}^{T} \boldsymbol{\gamma}_i(t) \otimes [[o_{it}]]\right) \tag{11}$$

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 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 $\boldsymbol{\tau}$ 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 BW 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 [13].

A 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 [13].

BDDs have been widely used in formal verification, model checking, and logic synthesis due to their ability to represent large Boolean functions efficiently while maintaining compact computational properties. However, in rare cases, BDDs can suffer from exponential blowup, which can occur particularly when dealing with functions that lack inherent structure or when representing numerical computations that go beyond Boolean logic.

3.8 Algebraic Decision Diagrams

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 the Boolean domain, ADDs can store arbitrary numerical values, making them useful for representing and manipulating functions over discrete domains [14]. This generalization enables the efficient representation of functions such as cost functions [15], probabilities [16], 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 [14].

4 IMPLEMENTATION

This section provides an overview of CuPAAL's implementation, including the Baum-Welch (BW) 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 BW algorithm for parameter estimation.

Specifically, we aim to improve the performance of the algorithm when handling large and complex models, and address the existing limitations of the BW algorithm in JAJAPY.

4.1.1 Recursive vs. Matrix vs. ADD-based Approaches

When working with the BW algorithm, different approaches can be taken to optimize computational efficiency. Three common strategies are recursive, matrix-based, and Algebraic Decision Diagram (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, allowing them to be accessed when needed [17, 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 for accessing stored results, leading to faster computations overall [17, 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 [14]. Compared to matrices, ADDs can offer a more space-efficient alternative for structured data while extending Binary Decision Diagram (BDD) techniques to handle both Boolean and numerical computations.

In this work, we investigate the advantages of ADD-based approaches for solving complex problems, with a focus on parameter estimation in Markov Chains (MCs) and Hidden Markov Models (HMMs). 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 BW algorithm for parameter estimation, which has evolved over time.

The initial version of CuPAAL was written in C and called *SUDD*, and was a partial implementation of the BW algorithm that utilized ADDs. This version was primarily focused on demonstrating 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 BW 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 MCs and can handle multiple observations. This version of CuPAAL is designed for standalone use, as well as in conjunction with JAJAPY, facilitating easy integration and application in parameter estimation problems.

The following sections provide an overview of what CuPAAL is and its capabilities.

4.2.1 What Does Cupaal Contain

Throughout all its iterations, CuPAAL has utilized the Colorado University Decision Diagram (CuDD) library - a library for implementing and manipulating BDDs and ADDs developed at the University of Colorado [18].

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 MCs.

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 directly, but we have added functions that wrap several functions of CuDD. 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 BW algorithm. The models are encoded from PRISM models to CUPAAL models, which is achieved by parsing the PRISM model into JAJAPY using STORMPY.

The JAJAPY model comprises a transition matrix, a label matrix, and an initial state vector. 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, because we want our matrices to be square whenever possible.

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 is provided 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.

Here we present a variation of the Kronecker product that only works between vectors - specifically one row and one column vector, as it relies on the structure of the vectors being expanded into square matrices.

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:

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

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

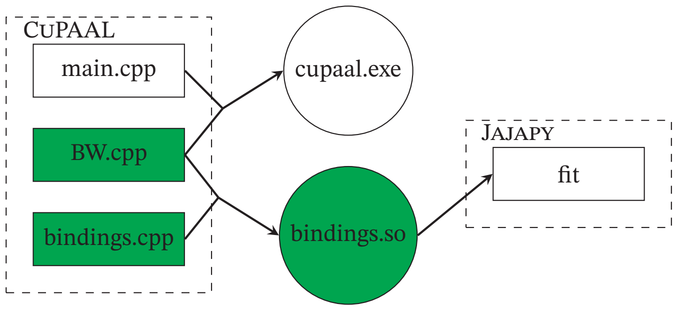

Fig. 2. Architecture of CUPAAL combined with JAJAPY.

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

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.

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

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:

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

Hereby showing that the Hadamard product can be used to compute the Kronecker product between two vectors, by using the row and column duplication method.

4.3 Implementation to Jajapy

This section provides an overview of how CUPAAL is implemented in Jajapy, utilizing bindings between C++ and Python. Figure 2 shows the overall architecture of the implementation.

CUPAAL consists of two primary components: the main function and the BW library. Both of these are compiled into an executable program called `cupaal.exe`, which can be used to run the BW 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 [19], which 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.

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 for use in CUPAAL. The transformation is done at line 3, where all the parameters are inputted to create a `Markov Model` object, which is then used to run the BW algorithm on line 6.

```
1  // Some parameters have been omitted for brevity
2  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){
3      MarkovModel model(states, labels,
        ↪  initial_distribution, transitions,
        ↪  emissions, observations);
4      cupaal_markov_model model_data;
5      chrono::seconds time = chrono::seconds(3600);
6      model.baum_welch_multiple_observations(
7          max_iterations, epsilon, time);
8
9      // output and result path omitted for brevity
10     model_data.initial_distribution =
        ↪  model.initial_distribution;
11     model_data.transitions = model.transitions;
12     model_data.emissions = model.emissions;
13
14     Cudd_Quit(model.manager);
15     return model_data;
16 }
```

Listing 3. C++ bindings file for CuPAAL

Each of the values relevant to the BW algorithm is then passed into the `model_data` object, which is an intermediate data object containing the learned model parameters returned to JAJAPY, as seen in lines 10 through 15. These are the initial distribution, the transitions and the emissions.

```
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 BW algorithm on multiple observations in CuPAAL.

The C++ bindings file is then compiled to a shared library, which can be imported into JAJAPY. JAJAPY can call the *cupaal_bw_symbolic* function, which will then call the CUPAAL implementation of the BW algorithm.

We create a new function in JAJAPY, called `_bw_symbolic`, which is used to call the CUPAAL implementation of the BW 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, returning the `cupaal_model` data object.

The values are then extracted from the `cupaal_model` data object and assigned to the JAJAPY model, as 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. When the parameter is true, the JAJAPY model will call the Listing 5 function, which will then call the CUPAAL implementation of the BW algorithm.

5 EXPERIMENTS

In this section, we present the experiments for evaluating and comparing the performance of two implementations of the Baum-Welch (BW) algorithm: the original version from JAJAPY and the new symbolic implementation introduced in CUPAAL.

For this comparison, we use a Markov Chain (MC) model, taken from the QComp benchmark set [5], which is a collection of models used for analysis of quantitative verification. The goal is to assess the 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 CUPAAL:

- **Accuracy** — Comparing the similarity of learned models in terms of log-likelihood and model checking.
- **Scalability** — Evaluating how the symbolic implementation scales with increasing model size and observation length.
- **Controlled initialization** — Evaluating the scalability of the symbolic implementation when initializing the model hypothesis with fewer unique values.

The experiments aim to answer the following research questions:

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

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

5.1 Model

The model used in the experiments is the leader sync model [20], a Discrete Time Markov Chain (DTMC) from the QComp benchmark set [5]. It simulates a group of processors selecting a leader.

The model sizes range from 26 to 1050 states, determined by the number of processors and the maximum number a leader can choose during election: $\{26, 69, 147, 61, 274, 812, 141, 1050\}$.

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.

We selected this model due to its scalability and interpretability. To make it suitable for learning, we extended it with additional labels.

The original model had only a single label, which is insufficient for meaningful training. We added two new labels: `reading` and `deciding` which correspond to key phases in the leader election process. The added labels and properties used for evaluation are shown in Listing 7.

```python
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 BW algorithm using CuPAAL.

```python
# 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 BW algorithm when symbolic is set to true.

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

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

Listing 7. Labels added to the leader sync model and properties checked.

5.2 Experimental Setup

All experiments were conducted on the same machine (see Appendix A for hardware and environment details). Because the CuPAAL implementation is not parallelized in anyway, for the most telling comparison we use only a single core for the experiments. Technically Jajapy uses a secondary core for printing to the console, but this does not factor in the numbers in any way.

The experimental steps are as follows:

1) Load the Prism model.
2) Generate 100 observation sequences of different lenghts. For experiment 1 and 2 we have $\{25, 50, 100\}$ and for experiment 3 we have $\{25, 50, 75, 100\}$.
3) Create a random initial MC using the function `jajapy.MC_random`.
4) Run the BW algorithm for up to 4 hours or until the change in log-likelihood is less than 0.01 (default stopping criterion in Jajapy).
5) Record runtime, number of iterations, log-likelihood, and save the resulting model.

We save both the initial models and observations in files to ensure both implementations use the same inputs. The generation of the training set and the randomization of the model are done using Jajapy, which provides a convenient way to generate random models and training sets.

The training set is generated by the System Under Learning (SUL), which is then used to train the randomized model. Each configuration (model size and dataset size) is repeated ten times to compute average results.

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: Accuracy

This experiment evaluates the accuracy of CuPAAL compared to the original Jajapy. The goal is to measure the similarity of the symbolic implementation in CuPAAL to the original recursive implementation in Jajapy accuracy-wise.

We compare the log-likelihood, which is a measure of how well the model fits the data in the symbolic implementation against the original recursive implementation.

We also measure the relative error of model checking properties to the correct model, which is the leader sync model from the QComp benchmark set. The properties can be seen in Listing 7, and the properties are taken from the QComp benchmark set.

The relative error is defined as:

$$\phi = \frac{|e - r|}{r} \tag{14}$$

where e is result of the verification of the learned model and r is the reference value from the original model.

5.4 Experiment 2: Scalability

This experiment evaluates the scalability of the symbolic implementation of the BW algorithm in CuPAAL. The goal is to measure the runtime performance of the symbolic implementation as the size of the model increases. Further we evaluate the runtime as the observation length increases as well.

The experiment measures the average runtime of the BW algorithm for each model size and number of observation sequences.

5.5 Experiment 3: Controlled initialization

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

This experiment aims to measure the scalability of CuPAAL under circumstances that are theoretically good for the symbolic implementation, i.e. the more repeated values the transition matrix contains, the sparser the Algebraic Decision Diagram (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. Specifically, we use the original first row of the original transition matrix, and shuffle the order for subsequent rows in the transition matrix for this experiment.

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 three main parts.

The first part focuses on the accuracy of JaJaPy and CuPAAL in terms of log-likelihood and model checking properties.

The second and third part evaluates the scalability of both tools, with differences as explained in section 5.

6.1 Accuracy

This experiment compares the accuracy of CuPAAL and JaJaPy in learning the leader sync model. Specifically, we model-check how many rounds it takes for each model to select a new leader, starting from the original model, JaJaPy, and CuPAAL, using properties from Listing 7.

Table 1 shows the results. The table includes the number of rounds for the original model (**rounds** column) and the learned models from JaJaPy and CuPAAL. The ϕ column shows the relative error between JaJaPy and the true value from the original model.

The results show that both CuPAAL and JaJaPy implementations learned the same model, and they both closely match the true model. For example, in the row for model 3.2 with 25 observations, the original model takes 1.33 rounds, and both CuPAAL and JaJaPy predict 1.28 rounds.

The relative error ϕ indicates a minimal deviation from the true value, but to evaluate if this is overfitting or not, additional test set log-likelihoods should have been calculated.

In conclusion, these results demonstrate that CuPAAL is a reliable and accurate implementation of the Baum-Welch (BW) algorithm, achieving performance comparable to that of JaJaPy in learning the model.

TABLE 1
Leader sync learning tool model comparison of CuPAAL, JaJaPy, and the reference model. R represents the reward model from Listing 7. ϕ represents the relative difference to the reference model.

$\mathcal{M}$	T	R	R_{ja}	R_{cup}	ϕ	ℓ cup	ℓ ja
3.2	25	1.33	1.28	1.28	0.04	-98.72	-98.72
3.2	75	1.33	1.30	1.30	0.02	-70.23	-70.23
3.2	100	1.33	1.28	1.28	0.04	-67.24	-67.24
3.3	25	1.12	1.23	1.23	0.09	-59.27	-59.27
3.3	50	1.12	1.11	1.11	0.01	-35.87	-35.87
3.3	100	1.12	1.13	1.13	0.00	-40.33	-40.33
3.4	25	1.07	1.08	1.08	0.01	-28.52	-28.52
3.4	50	1.07	1.09	1.09	0.02	-31.07	-31.07
3.4	100	1.07	1.11	1.11	0.04	-35.87	-35.87
4.2	25	2.00	2.11	2.12	0.06	-140.41	-140.41
4.2	50	2.00	2.16	2.16	0.08	-149.13	-149.13
4.2	100	2.00	2.00	2.00	0.00	-138.63	-138.63
4.3	25	1.35	1.37	1.37	0.01	-79.92	-79.92
4.3	50	1.35	1.28	1.28	0.05	-67.24	-67.24
4.3	100	1.35	1.25	1.25	0.07	-62.55	-62.55
4.4	25	1.19	1.25	1.25	0.05	-62.55	-62.55
4.4	50	1.19	1.17	1.17	0.01	-48.50	-48.50
4.4	100	1.19	1.27	1.27	0.07	-65.71	-65.71
5.2	25	3.20	3.27	3.27	0.02	-155.11	-155.11
5.2	50	3.20	3.06	3.06	0.04	-185.72	-185.72
5.2	100	3.20	3.50	3.50	0.09	-208.39	-208.39
5.3	25	1.35	1.32	1.32	0.02	-73.11	-73.11
5.3	50	1.35	1.27	1.27	0.06	-65.71	-65.71
5.3	100	1.35	1.33	1.33	0.01	-74.52	-74.52

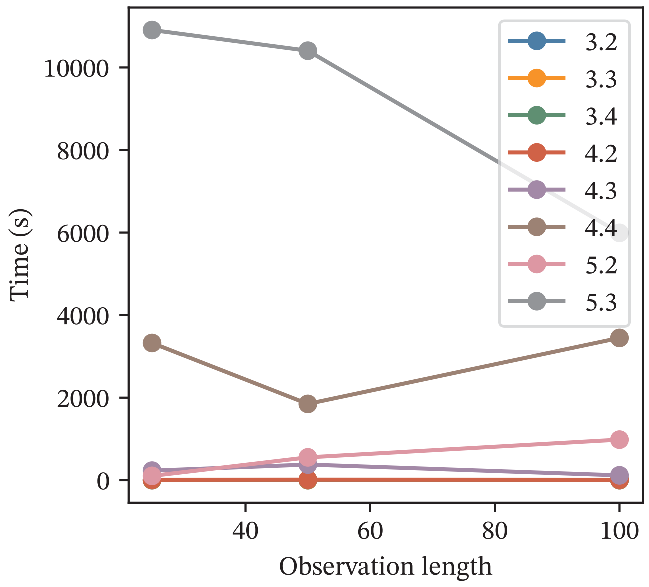

Fig. 3. CuPAAL runtimes with increasing observation length.

TABLE 2
Leader sync model variations in training time in seconds.

$\mathcal{M}$	$\|S\|$	T	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

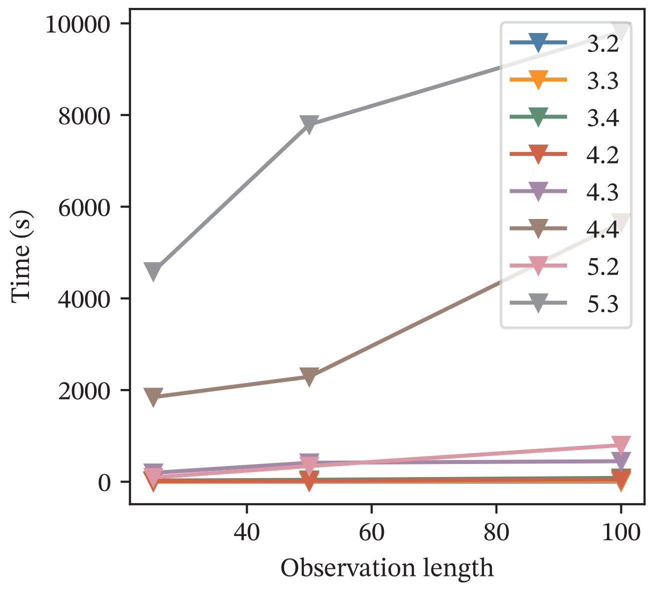

Fig. 4. JAJAPY runtimes with increasing observation length.

6.2 Scalability

These results represent the time taken to train a model based on two parameters: the number of states and the length of the observations in the training set.

The results for the leader sync model are presented in Table 2 and Figure 5, showing the time required to train a model based on 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.

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 Markov Chains (MCs).

For very small models, the running time does not matter too much; however, we observe an initial overhead in JAJAPY compared to CUPAAL. This is likely related to the consensus that Python is generally slower than C.

More states mean longer running time, but interestingly, variations with a similar number of states may have very different training times. The most obvious example is the 3.4 and 5.2 models, which have 147 and 141 states, respectively. The 5.2 model is significantly slower, especially in CUPAAL, exhibiting a nearly 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 significantly better than CUPAAL.

To explore this behavior, we extended the experiment to include data for observations of different lengths, and our observations are now 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.

Looking at Figures 3 and 4, the CUPAAL runtime decreases as the observation length increases. This is contrary to what one might intuitively expect, as one would assume that with more data, the calculations either increase the running time or have no effect on it.

Our hypotheses as to why this is the case for Figure 3 are due to the model this experiment is based on. Since leader sync usually ends its observations with the label `elected`, as the observation length increases, more of this label appears in sequence.

This leads the BW algorithm to learn the final part of the model more quickly and, at a certain point, reach a constant value that it can reuse in future computations, thereby accelerating the learning of part of the model.

6.3 Controlled Initialization

This section will cover the third experiment, which compares CUPAAL and JAJAPY. This experiment explores the effect of random and controlled initial model parameters, as we expect repeated values to be highly beneficial for CUPAAL's implementation.

Table 3 presents three variations of the leader sync model with varying numbers of states and observation sequences. We notice that the initialization method can have an impact on the number of iterations it takes to learn the model.

This is to be expected, as the initialization of a Markov model is known to impact the BW algorithm [8]. We are therefore interested in the time per iteration as a metric for runtime comparison.

Table 3 compares CUPAAL to JAJAPY and the impact of random and controlled initialization on the time needed to learn the model.

Examining the columns `rand-ja` and `control-ja`, which show the impact of random and controlled initialization for JAJAPY, reveals no significant difference between the two.

However, when examining the differences for CUPAAL it is clear that the controlled approach is generally faster than the completely random one; this becomes especially pronounced as the number of states in the model increases.

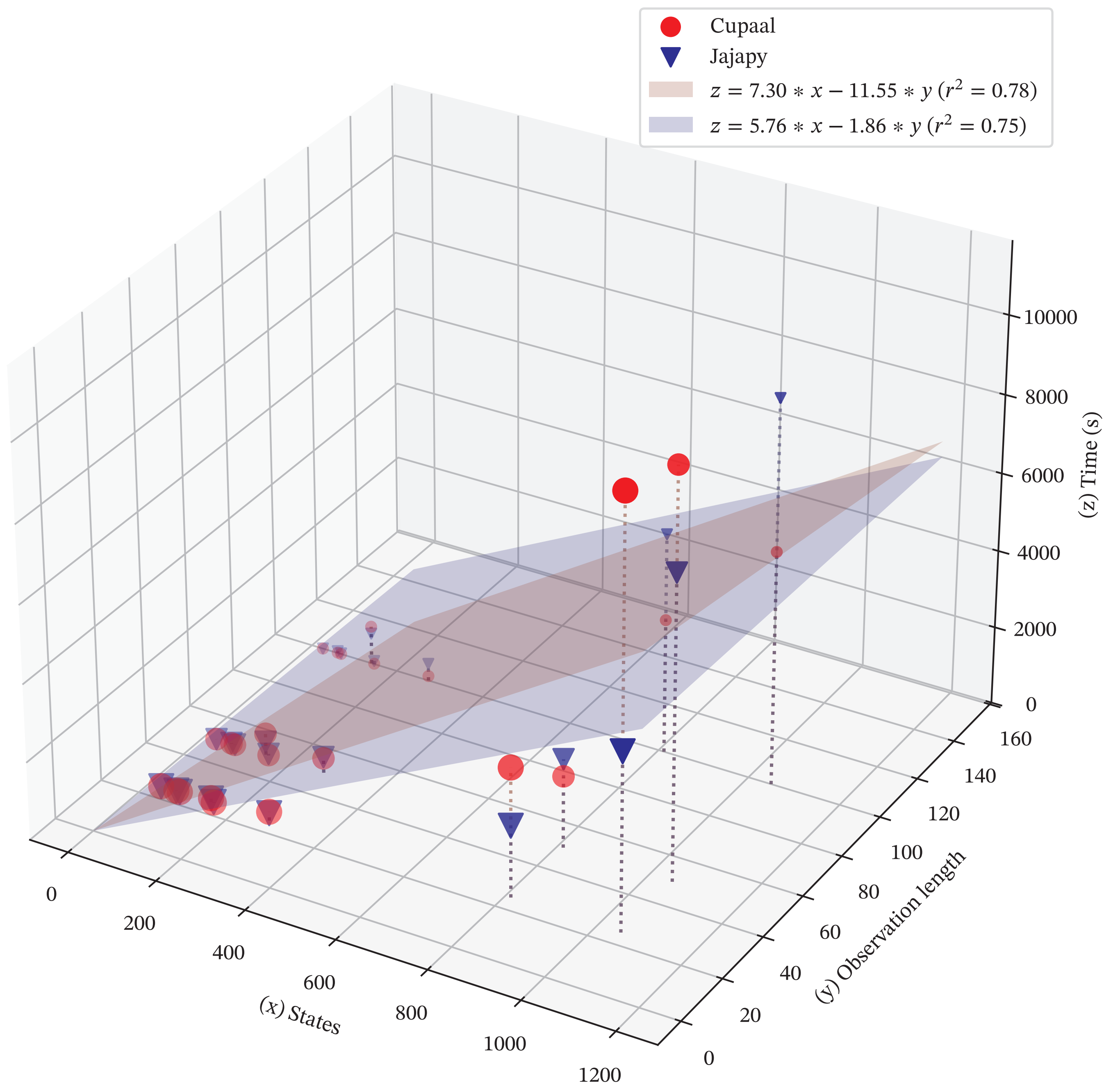

Fig. 5. Plot of the run time of JAJAPY and CUPAAL for the leader sync models, given the number of states and the length of the observations. The planes are linear regression fits to indicate the directions of the trends for the datapoints of similar color. This is not an attempt to make any definitive statements about the degrees of scaling but rather to illustrate the generally observable trend.

The Δ columns for CUPAAL (cup) and JAJAPY (ja) show this relative difference between the different initializations. The mean difference for CUPAAL is $\sim -11\%$, while it is $\sim -1\%$ for JAJAPY.

We expected this to be the case, as the number of observations resulted in more repeated values, and the controlled initialization had a similar effect. This seems to partially alleviate the poorer scaling CUPAAL suffers from as the number of model states increases at low observation lengths.

Both the random and controlled initialization resulted in identical log-likelihoods, meaning that regardless of the method used, the learned model is still equally close to the correct model, which is still in line with our results from experiment 1.

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

These graphs illustrate the general trend of CUPAAL performing slightly better with controlled initialization, whereas for JAJAPY, we observe no clear tendency for what performs best.

These results indicate a gain for CUPAAL when there are repeated values for the initialization.

TABLE 3
Leader sync model variations in training time with random (ran) and controlled (con) initial values. i is the number of iterations, s represents seconds s/i represents seconds per iteration, and Δ represents the relative difference between random and controlled initialization.

$\mathcal{M}$	T	i ran	i con	s ran cup	s con cup	s ran ja	s con ja	s/i ran cup	s/i con cup	s/i ran ja	s/i con ja	Δ cup	Δ ja
4.2	25	18	16	11.12	7.87	15.68	13.26	0.62	0.49	0.87	0.83	-20.34	-4.87
4.2	50	17	20	13.56	12.54	24.87	28.32	0.80	0.63	1.46	1.42	-21.41	-3.21
4.2	75	17	18	9.72	10.54	36.02	38.62	0.57	0.59	2.12	2.15	2.41	1.27
4.2	100	17	18	11.24	10.95	52.11	53.75	0.66	0.61	3.07	2.99	-8.04	-2.58
4.3	25	18	18	231.28	194.04	194.88	190.65	12.85	10.78	10.83	10.59	-16.10	-2.17
4.3	50	18	18	379.21	308.81	414.30	414.95	21.07	17.16	23.02	23.05	-18.56	0.16
4.3	75	18	18	232.21	206.67	476.68	486.04	12.90	11.48	26.48	27.00	-11.00	1.96
4.3	100	18	18	117.78	118.17	447.83	441.40	6.54	6.57	24.88	24.52	0.33	-1.44
4.4	25	18	18	3324.83	3087.66	1846.67	1793.90	184.71	171.54	102.59	99.66	-7.13	-2.86
4.4	50	18	18	1848.44	1526.81	2290.28	2239.18	102.69	84.82	127.24	124.40	-17.40	-2.23
4.4	75	17	18	1597.78	1512.33	3017.72	3177.81	93.99	84.02	177.51	176.54	-10.61	-0.55
4.4	100	18	18	3447.56	2959.75	5652.14	5586.23	191.53	164.43	314.01	310.35	-14.15	-1.17

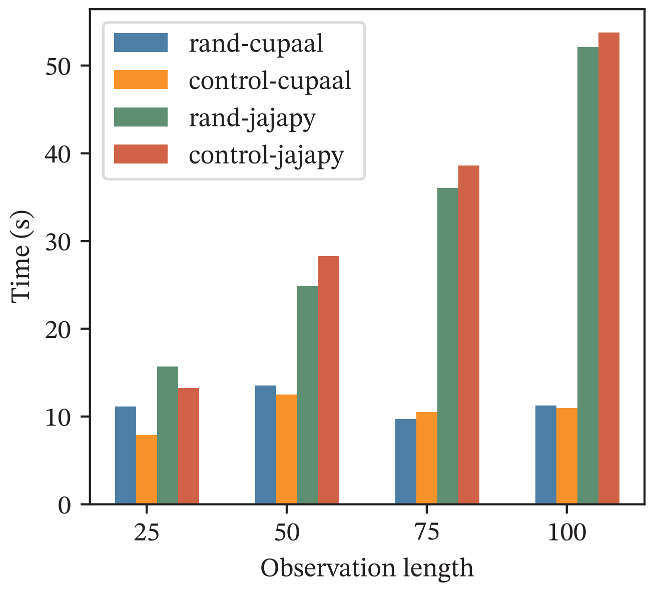

Fig. 6. Model 4.2 - Runtime for random and controlled initialization

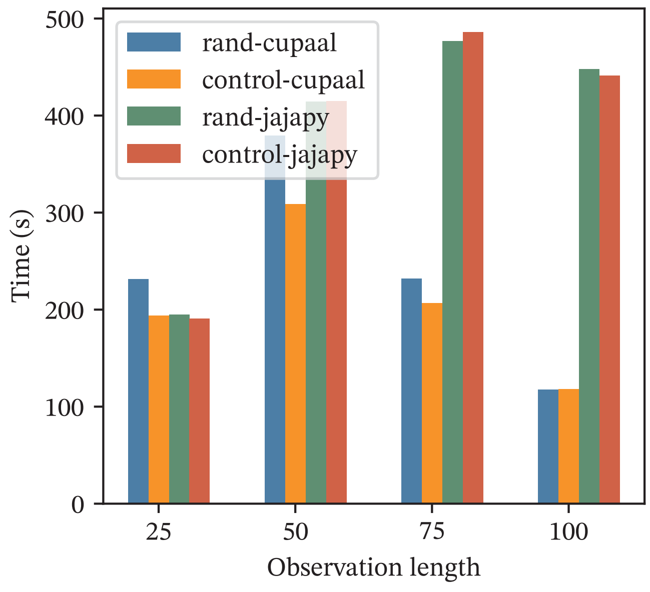

Fig. 7. Model 4.3 - Runtime for random and controlled initializationm

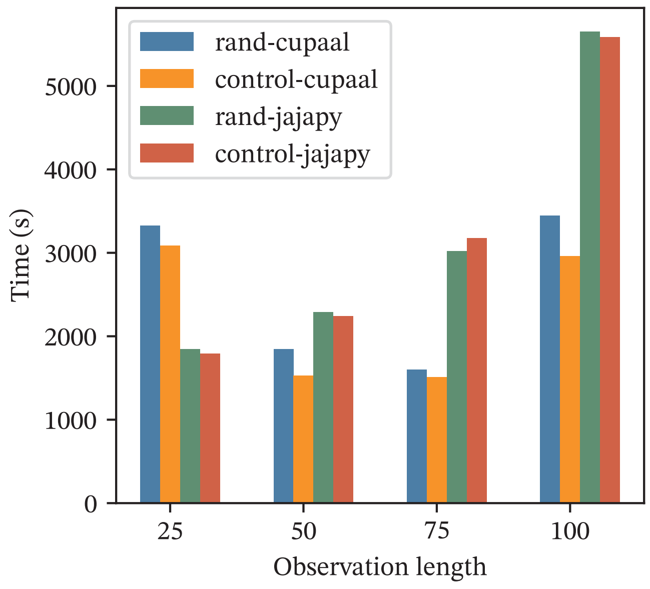

Fig. 8. Model 4.4 - Runtime for random and controlled initialization

This lines up well with our hypothesis that CUPAAL performs better when it can leverage repeated values for its Algebraic Decision Diagram (ADD) structure.

7 Discussion

In this section, we discuss the results presented in section 6 and reflect on the performance of CUPAAL compared to JAJAPY.

The results presented in subsection 6.1 confirm that CUPAAL and JAJAPY learn identical models, which is expected as the Baum-Welch (BW) algorithm should be the same despite the different technical implementations. Both reach the same model in the same number of iterations, with identical log-likelihood and accuracy, as confirmed by model checking with properties, which establishes their computational equivalence and provides a solid foundation for further work with symbolic implementations.

Unfortunately, the scalability results from Sections 6.2 and 6.3 do not show CUPAAL to be as clear a winner as we had expected. There are benefits to a symbolic calculation, but they are not as pronounced as previous work suggested in work done for implementations targeting only the forward-backward procedure.

The theoretical advantage of matrices with a lot of repeated values represented as Algebraic Decision Diagrams (ADDs) is confirmed, as seen by the results in subsection 6.3, and there may be even more performance to be gained with different symbolic implementations. Currently, CuPAAL cleans up after each iteration, which may generally improve or worsen performance; however, we expect it to worsen performance in cases with a high number of repeat values.

The experiment is conducted using a single model, leader sync, which may not provide a comprehensive view of performance across different scenarios. A more effective approach would have been to utilize multiple models with varying characteristics and compare their performance, which would allow for a more comprehensive overview of CuPAAL's scalability compared to Jajapy.

Other models that were considered were the NAND multiplexing (NAND) and Bounded Retransmission Protocol (BRP) models, also from the qcomp benchmark set [5]. A wider variety of models would provide more and clearer insight into CuPAAL, thereby displaying its strengths and weaknesses more completely.

The model could also have utilized a greater number of states and observations; currently, the largest model contains 1,050 states and observation sequences of length 100.

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

We ran the experiments in a Docker container, which introduces some level of overhead. We have not investigated whether this overhead introduces any bias towards Jajapy or CuPAAL. While we do not expect it to do so, we also cannot say that we fully understand the overhead of Docker. For completeness, this might have been better explored.

In subsection 5.5, the values for the initial model values are not entirely random. Instead, they are designed to have repeated values, which was done to display the theoretical strengths of CuPAAL. This method skews the model to favor CuPAAL, as the ADD structure 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. It would be interesting to explore the limits of this initialization strategy for the BW algorithm in general, as initialization of the System Under Learning (SUL) is a point of research on its own.

7.1 Implementation Discussion

The integration of CuPAAL into Jajapy has been successful, allowing us to leverage the BW algorithm for parameter estimation for Hidden Markov Models (HMMs) and Markov Chains (MCs).

The decision to use pybind11 for creating bindings between C++ and Python has proven effective, as it allows us to easily call C++ functions from Python.

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

CuPAAL displays clear benefits when working with repeated values; however, it did not compare as favorably to Jajapy as we expected.

Previous work indicated that CuPAAL overall was a stronger implementation, but with an entirely symbolic implementation, some potential bottlenecks have been observed.

Specifically in the update step of the BW algorithm, as when working with ADDs for just the α and β steps, CuPAAL performed very well - much better than what is indicated for the complete BW algorithm implemented here.

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 α and β and then employ a recursive approach when updating values.

An implementation like this would require much conversion 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 in situations where memory was a constraint. However, without a memory metric to compare, the decision can only be made based on the time required for computing BW.

The library used to manipulate ADDs was Colorado University Decision Diagram (CuDD), as it was what previous work had built upon. A discussion at the time also raised the question of whether this is the best tool for the job, as other tools, such as Sylvan [21], exist. This discussion remains relevant and worth exploring, especially in the context of parallelization, which is not possible in the current implementation of CuPAAL.

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 could 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 as well. However, improving CuPAAL to support multiple cores could be a worthwhile direction to explore, as it could provide a significant performance increase.

In a multiset of observation sequences, sequences are grouped if they are identical, meaning that when computing these observation sequences, we can factor in the number of identical sequences and only compute the α and β values once for the same sequence.

Expanding upon this idea involves utilizing 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 whole sequences are currently.

Given that an observation sequence contains many observations that share prefixes and suffixes of labels, effectively building a tree structure of prefixes/suffixes. 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 a MC. This might be worth adding, as in these cases, unnecessary computation is made, as MCs do not require the ω 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 Hidden Markov Models (HMMs) and Markov Chains (MCs), leveraging Algebraic Decision Diagrams (ADDs) to replace traditional matrix and recursive representations.

By reformulating the Baum-Welch (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 hybrid implementations, parallelization, and extensions of CuPAAL to other model types, such as Markov Decision Processs (MDPs) and Continuous Time Markov Chains (CTMCs).

9 Acknowledgements

We would like to thank our two supervisors, Giovanni Bacci and Raphaël Reynouard for their guidance, supervision, and support throughout this project.

Giovanni provided valuable insights into the theoretical aspects of the Baum-Welch (BW) algorithm and its symbolic implementation, while Raphaël, the creator of Jajapy, offered practical advice on integrating our work into the existing Jajapy library and knowledge of the BW algorithm.

Additionally, we would like to acknowledge that AI tools, ChatGPT and Grammarly were used to assist in the writing and editing of this paper. ChatGPT was used in the phrasing of several paragraphs of text, and Grammarly has been used to proofread all the body text.

Acronyms

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

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 *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 344–350.

[6] R. Reynouard, A. Ingólfsdóttir, and G. Bacci, "Jajapy: A learning library for stochastic models," in *Quantitative Evaluation of Systems*, N. Jansen and M. Tribastone, Eds., Cham: Springer Nature Switzerland, 2023, pp. 30–46, ISBN: 978-3-031-43835-6.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] R. Reynouard et al., “On learning stochastic models: From theory to practice,” 2024.

[13] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[14] 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.

[15] 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.

[16] 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.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[18] F. Somenzi, “Cudd: Cu decision diagram package,” *Public Software, University of Colorado*, 1997.

[19] W. Jakob, *Pybind11*, 2025.

[20] A. Itai and M. Rodeh, “Symmetry breaking in distributed networks,” *Information and Computation*, vol. 88, no. 1, 1990.

[21] T. Van Dijk and J. Van de Pol, “Sylvan: Multi-core framework for decision diagrams,” *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 675–696, 2017.

Appendix A
Machine Specifications

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

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

TABLE 6
Symbol table.

Symbol	**Meaning**
$\mathbb{R}$	Real numbers
$\mathbb{B}$	Boolean domain
$\mathbb{N}$	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
π	Initial distribution
τ	Transition function
ω	Emission function
α	Forward probabilities
β	Backward probabilities
γ	State probabilities given O
ξ	Transition probabilities given O
$\lambda = (\pi, \tau, \omega)$	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 $\mathcal{O}$
$\cdot$	Scalar product
$\odot$	Hadamard product
$\otimes$	Kronecker product
$\oslash$	Hadamard division
$\bullet$	Transposed Khatri-Rao product