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.

However, these methods can become computationally expensive, particularly for large state spaces and long observation sequences. The inherent redundancy in matrix-based representations leads to inefficiencies in both time and memory usage, limiting scalability in practical applications.

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 HMMs, significantly reducing memory consumption and accelerating computation.

This paper explores the integration of ADDs into the Baum-Welch algorithm, demonstrating how this approach enhances efficiency while preserving numerical accuracy.

The proposed method is implemented in the tool CuPAAL, which will be compared to the Python library Jajapy, an existing implementation that employs a recursive matrix-based approach. These comparisons will evaluate key factors such as scalability, runtime, number of iterations, and log-likelihood.

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.

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 two primary tools: 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.

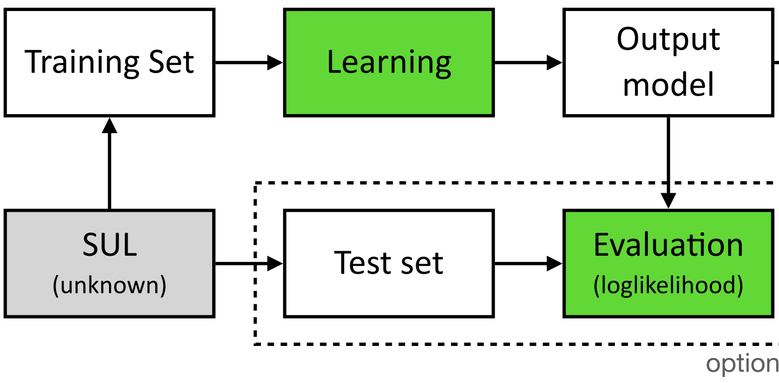

Fig. 1. Modeling and verification workflow using JAJAPY. Phases involving JAJAPY are highlighted in green, while the blue phase represents verification using STORM or PRISM.

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

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

At the core of JAJAPY's learning capabilities are several variants of the Baum-Welch (BW) algorithm [4, 5], adapted for discrete-time Markov chains (MCs), Markov decision processes (MDPs)[6], and continuous-time Markov chains (CTMCs)[7]. 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,

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

```python
from jajapy import BW
type(training_set) # list
output_model = BW().fit(training_set,
   ↪ nb_states=10) type(output_model) #
   ↪ stormpy.SparseDtmc
```

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

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.

Once a model has been learned, JAJAPY supports direct verification of properties using STORM, 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 certain components of the Baum-Welch algorithm were optimized 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. 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.

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 report. We begin by defining the key concepts of a Hidden Markov Model (HMM) and a Markov Decision Process (MDP), which are the two main models used in this report.

3.1 Hidden Markov Model

HMMs were introduced by Baum and Petrie in 1966 [8] and have since been widely used in various fields, such as speech recognition [1], bioinformatics [2], and finance [3].

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 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.

An example of an 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.

3.2 Continuous Time Hidden Markov Model

In the above definition, we have defined a discrete-time HMM, meaning that the system evolves in discrete time steps. In this report, we are interested in continuous-time systems, where the system evolves in continuous time. To model continuous-time systems, we use a Continuous Time Hidden Markov Model (CTHMM), which is an extension of the HMM to continuous time.

Definition 2 (Continuous Time Hidden Markov Model). *A Continuous Time Hidden Markov Model (CTHMM) is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi, \lambda)$, where S, $\mathcal{L}$, ℓ, τ, and π are as defined in the HMM definition.*

- *$\lambda : S \times S \to \mathbb{R}_{\geq 0}$ is the rate function.*

The rate function λ determines the transition rate between states, meaning that the time spent in a state before transitioning follows an exponential distribution.

3.3 Markov Decision Process

MDPs were introduced by Bellman in 1957 [9] and have since been widely used in various fields, such as robotics [10], finance [11], and healthcare [12].

Definition 3 (Markov Decision Process). *A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (S, A, \mathcal{L}, \ell, \tau, R, \gamma)$, where:*

- *S is a finite set of states.*
- *A is a finite set of actions.*
- *$\mathcal{L}$ is a finite set of labels.*

- *$\tau : S \times A \rightarrow D(S)$ is the transition function.*
- *$R : S \times A \rightarrow \mathbb{R}$ is the reward function.*
- *$\pi \in D(S)$ is the initial distribution.*

A MDP works by an agent interacting with an environment. Unlike an HMM, the agent takes actions in the environment and receives rewards. An agent is the decision-maker that interacts with the environment by taking actions. The environment is modeled as a MDP, which consists of a set of states S, a set of actions A, a set of labels $\mathcal{L}$. Each state is directly observable, meaning the agent always knows which state it is in. When the agent takes an action in a state, it transitions to a new state according to the transition function τ.

The reward function R describes the reward received when taking an action in a state. The goal of an MDP is to find a policy $\theta : S \rightarrow D(A)$ that maximizes the expected cumulative reward. The policy θ describes the probability of taking an action given a state.

Definition 4 (Policy). *A policy θ is a function that maps states to actions, i.e., $\theta : S \rightarrow D(A)$.*

An example of an MDP is a robot navigating a grid, where it can choose actions (move up, down, left, or right) and receives rewards based on reaching certain goal locations

4 METHODOLOGY

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 Binary Decision Diagrams (BDDs) and ADDs.

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.

4.1 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 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.

4.1.1 From BDDs to 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 [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 applications like dynamic programming, MDPs, and linear algebraic computations [14].

4.2 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. Previouse 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 [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 of accessing the stored results leading the 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 sub-structures, they improve efficiency and reduce memory overhead [14]. 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.

4.3 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 [18] is a powerful tool for implementing and manipulating 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 this paper. 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.

4.4 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.

The Transition matrix is a $S \times S$ matrix, where $S = States$, and is encoded to an ADD, by assigning each row and column with a binary value. This value is determined based on the size of the matrix, $|binaryValue| = \lceil log_2(S) \rceil$. Meaning for a 2×2 matrix, a single binary value for each row and column, will suffice. For this case, the first row will be assigned the binary value 0, and the second row will be assigned 1, and vice versa for the columns.

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.

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. However, we adapted how we represent vectors to optimize our symbolic computations.

This is due to 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 $\hat{B} = \begin{bmatrix} 3 & 4 \end{bmatrix}$.

$\hat{A}$ 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 $\mathbf{B} = \begin{bmatrix} 3 & 4 \\ 3 & 4 \end{bmatrix}$.

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 JAJAPYand 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 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

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

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 [20]. 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	$\lvert S\rvert$	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:

- **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.

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

6 RESULTS

In this section, we present the results of our experiments comparing the performance of Jajapy and CuPAAL, using the Baum-Welch algorithm. The metrics used for comparison are the time taken to train the model, the number of iterations needed, the average error, and the log-likelihood of the model.

The experiments were conducted on the same machine, the specifications of which are provided in Table.

For both experiments on DTMCs and CTMCs, we expect the CuPAAL implementation to be faster than the Jajapy implementation due to the symbolic approach, which can avoid redundant calculations. We also expect the accuracy to be consistent whether the type of model is a DTMC or a CTMC between the two implementations since the Baum-Welch algorithm is deterministic.

The results will be presented in tables for each model, showing the average time per run for each implementation, average number of iterations and log-likelihood and error for each model.

6.1 Performance Results

The first results presented are the performance results of the experiments. These results are the time taken to train the model, the number of iterations needed to train the model, and the log-likelihood of the model.

The results for DTMCs are displayed in tables Figure 2, Table 4, and Table 5. The same is done for CTMCs in tables Table 6, Table 7, and Table 8.

The results show a clear difference in the time taken to train the model between Jajapy and CuPAAL, for both DTMCs and CTMCs. CuPAAL is significantly faster than Jajapy, across all models used in the experiments. The difference between the two tools, in terms of time taken to train the model, is especially pronounced for the larger models used in the experiment.

The results also show that the number of iterations needed to train the models are similar for Jajapy and CuPAAL, across all models used in the experiment, both DTMCs and CTMCs. This shows that the reduction of time needed to train the modle is not due to a reduction in the number of iterations needed to train the model, and that the difference in time taken to train the model is due to the symbolic approach used in CuPAAL.

The log-likelihood of the models are also similar for Jajapy and CuPAAL, across all models used in the experiment. The log-likelihood of the model is a measure of how well the model fits the data, and the lower the value of the log-likelihood, the better the model fits the data. With the log-likelihood being similar for Jajapy and CuPAAL, this shows that the difference in time taken to train the model between Jajapy and CuPAAL does not come at the cost of accuracy.

With the results showing that CuPAAL is significantly faster than Jajapy, and that the number of iterations needed to train the model is similar for both tools, the implementation of the Baum-Welch algorithm in CuPAAL is an efficient implementation in terms of performance.

6.2 Scalability Results

The second results presented are the scalability results of the experiments. These results are the time taken to train a model, with the number of states in the model increasing.

TABLE 3
Leader_sync results

Implementation	Iter	Time(s)	Avg δ	Log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

TABLE 4
Brp results

implementation	Iter	Time(s)	avg δ	log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

TABLE 5
Crowds results

implementation	Iter	Time(s)	avg δ	log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

TABLE 6
Mapk results

implementation	Iter	Time(s)	avg δ	log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

TABLE 7
Cluster results

implementation	Iter	Time(s)	avg δ	log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

TABLE 8
Embedded results

implementation	Iter	Time(s)	avg δ	log-likelihood
Jajapy	0	0	0	0
CuPAAL	0	0	0	0

The results for DTMCs are displayed in figure Figure 2, and the same is done for CTMCs in figure Figure 3.

The results show a clear difference in the time taken to train the model between Jajapy and CuPAAL, for both DTMCs and CTMCs. CuPAAL is significantly faster than Jajapy, across both models used in the experiments. This is especially noticable as the models grow larger throughout the experiment.

At the early stages of the experiment where the amount of states are low, the difference in time taken to train the model is not as pronounced, and at times Jajapy is faster than CuPAAL. This is likely due to the smaller size of the model, not having as much redundant calculations to avoid, which is the main advantage of the symbolic approach used in CuPAAL.

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.

missing picture

Fig. 2. Scalability results for the leader_sync model

missing picture

Fig. 3. Scalability results for the polling model

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.

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.

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.

Acronyms

AAU	Aalborg University. 1
ADD	Algebraic Decision Diagram. 1–4, 6, 7
BDD	Binary Decision Diagram. 3, 4
CTHMM	Continuous Time Hidden Markov Model. 2
CTMC	Continuous Time Markov Chain. 4–6
CuDD	Colorado University Decision Diagram. 3, 4
DTMC	Discrete Time Markov Chain. 4–6
HMM	Hidden Markov Model. 2, 3, 6, 7
MDP	Markov Decision Process. 2, 3

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] R. BELLMAN, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957, ISSN: 00959057, 19435274.

[10] H. Ali, D. Gong, M. Wang, and X. Dai, "Path planning of mobile robot with improved ant colony algorithm and mdp to produce smooth trajectory in grid-based environment," *Frontiers in Neurorobotics*, vol. 14, 2020, ISSN: 1662-5218. DOI: 10.3389/fnbot.2020.00044.

[11] B. Hambly, R. Xu, and H. Yang, "Recent advances in reinforcement learning in finance," *Mathematical Finance*, vol. 33, no. 3, pp. 437–503, 2023. DOI: https://doi.org/10.1111/mafi.12382. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/mafi.12382.

[12] C. Yu, J. Liu, S. Nemati, and G. Yin, "Reinforcement learning in healthcare: A survey," *ACM Comput. Surv.*, vol. 55, no. 1, Nov. 2021, ISSN: 0360-0300. DOI: 10.1145/3477600.

[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] A. Hartmanns, M. Klauck, D. Parker, T. Quatmann, and E. Ruijters, "The quantitative verification benchmark set," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 344–350.

[20] R. Reynouard et al., "On learning stochastic models: From theory to practice," 2024.

Appendix

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

Symbol	**Meaning**
$\mathbb{R}$	Real numbers
$\mathbb{Q}$	Rational numbers
$\mathbb{N}$	Natural numbers
$s \in S$	States
$l \in L$	Labels
$a \in A$	Actions
$\mathcal{M}$	Markov Model
$\mathcal{H}$	Hypothesis
$o \in O \in \mathcal{O}$	Observations
π	Initial distribution
τ	Transition function
ι or ω	Emission function
α	Forward probabilities
β	Backward probabilities
γ	State probabilities
ξ	Transition probabilities
$\lambda = (\pi, \tau, \omega)$	Model Parameters
ϕ or ψ	Scheduler
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 $\mathcal{O}$