Baum-Welch Algorithm for Markov Models Using Algebraic Decision Diagrams

Sebastian Aaholm[1], Lars Emanuel Hansen[1], and Daniel Runge Petersen[1][0009-0004-6529-4342]

Dept. of Computer Science, Aalborg University, Aalborg, Denmark

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 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.

2.1 Hidden Markov Model

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

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.

2.2 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.

2.3 Markov Decision Process

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

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.*
- $\tau : S \times A \to D(S)$ *is the transition function.*
- $R : S \times A \to \mathbb{R}$ *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 $\theta : S \to D(A)$ 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.,* $\theta : S \to D(A)$*.*

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

3 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.

3.1 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 [4].

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 [4].

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.

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 [5]. This generalization enables the efficient representation of functions such as cost functions [6], probabilities [7], 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 [5].

3.2 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 [8, 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 [8, 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 [5]. 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.

3.3 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 [9] 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 Experiments

In this section, we describe the experiments conducted to evaluate the performance of the symbolic implementation of the Baum-Welch algorithm in the CuPAAL library by comparing it to the recursive implementation in Jajapy. The evaluation is based on two key aspects: execution time and accuracy.

We conduct two experiments:

- **Performance Comparison** - Measuring runtime and accuracy across different models.
- **Scalability Analysis** - Evaluating performance as the number of states increases.

Through these experiments, we aim to answer the following research questions:

- **Question 1**: How does the symbolic implementation of the Baum-Welch algorithm in CuPAAL compare to the recursive implementation in Jajapy in terms of runtime and accuracy?

– **Question 2**: How does the performance of the CuPAAL implementation scale with the size of the model?

4.1 Experimental Setup

All experiments are conducted using a set of DTMCs and CTMCs obtained from publicly available benchmarks [10] [1] [10].

Each experiment is run ten times. We report the average runtime (full run and per iteration), the average number of iterations, log-likelihood per iteration, and the type of error based on the model type.

Experiments stop when reaching a convergence threshold of 0.05 (the Jajapy default) or a 4-hour runtime limit. The final iteration's results are recorded.

The training data is randomly generated based on these models, consisting of 30 observation sequences of length 10 for each model.

The implementations used are:

1. The original Jajapy implementation.
2. The symbolic CuPAAL implementation.

4.2 Experiment 1: Performance Comparison of Implementations

The first experiment is based on the ideas from the experiment conducted in [11].

We split this experiment into two separate analyses: one focusing on DTMCs and another on CTMCs. Since DTMCs estimate probabilities while CTMCs estimate rates, we use different error measures for accuracy evaluation.

The experiments evaluate the efficiency and accuracy of the symbolic approach (CuPAAL) versus the recursive approach (Jajapy). 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.

Log-likelihood: Measures how well a learned model explains observed data. For a given observation sequence O and model M, it is defined as:

$$\log P(O \mid M) = \sum_{t=1}^{T} \log P(O_t \mid M) \tag{1}$$

where $P(O_t|M)$ is the probability of observing O_t given the model.

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.

[1] The models are available at `https://qcomp.org/benchmarks/`. The models are Leader_sync, Brp, Crowds, Mapk, Cluster, and Embedded.

Experiment 1A: Performance Comparison for DTMCs For DTMCs, the models used are shown in Table 1. For DTMCs, we use the absolute error as the measure of accuracy, because in DTMCs we are estimating probabilities, and we are interested in the absolute difference between the real value and the estimated value.

The absolute error is calculated as follows:

$$\text{Absolute Error} = |r - e| \tag{2}$$

where r is the real value and e is the expected value.

The results are presented for DTMCs in Table 3, Table 4, and Table 5.

Experiment 1B: Performance Comparison for CTMCs For CTMCs, the models used are shown in Table 2.

For CTMCs, we use relative error as the measure of accuracy, since rates in CTMCs can vary significantly in magnitude, an absolute difference may not properly reflect the accuracy. For example, a difference of 0.1 in a transition rate of 0.2 is a large error, whereas the same difference in a rate of 10 is negligible. Using relative error ensures that errors are proportional to the expected value.

The relative error is calculated as follows:

$$\text{Relative Error} = \frac{|r - e|}{e} \tag{3}$$

where r is the real value and e is the expected value.

The results are shown in Table 6, Table 7, and Table 8.

Table 1. DTMC models

Name	Number of States
Leader_sync	274
Brp	886
Crowds	1145

Table 2. CTMC models

Name	Number of States
Mapk	118
Cluster	820
Embedded	3480

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

4.3 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 and CTMCs over the number of states. We measure:

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

For this experiment, we selected two models-*leader_sync* (DTMC) and *polling* (CTMC)—as they represent those used in the performance comparison experiment and scale well to large state spaces.

Experiment 2A: Scalability for DTMCs In this experiment, we evaluate the scalability of CuPAAL for DTMCs by measuring runtime efficiency as the number of states increases. 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. The results are shown in Figure 1. We expect CuPAAL to demonstrate lower runtime and improved scalability, while maintaining accuracy comparable to Jajapy.

missing picture

Fig. 1. Scalability results for the leader_sync model

Experiment 2B: Scalability for CTMCs In this experiment, we evaluate the scalability of CuPAAL for CTMCs by measuring runtime efficiency as the number of states increases. We use the *polling* model, scaling from 36 to 1334 states.

This experiment provides insights into how the symbolic approach scales as model complexity increases. The results are shown in Figure 2. We expect CuPAAL to demonstrate lower runtime and improved scalability, while maintaining accuracy comparable to Jajapy.

missing picture

Fig. 2. Scalability results for the polling model

5 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 4.

5.1 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.

5.2 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 4, 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.

5.3 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.

5.4 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.

6 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.

6.1 Jajapy

Jajapy is a Python library developed to implement the Baum-Welch algorithm for learning various types of Markov models. The library employs a recursive implementation of the Baum-Welch algorithm to infer the parameters of these models.

Jajapy was chosen for our research due to its established use of recursive methods in training Hidden Markov Models (HMMs) and other probabilistic models. Our objective was to explore the impact of using ADDs in the Baum-Welch algorithm, as opposed to conventional matrix-based or recursive methods. Jajapy served as an appropriate baseline for comparison, allowing us to evaluate the potential computational efficiency and accuracy improvements that ADDs might offer.

6.2 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.

Acronyms

AAU Aalborg University. 1
ADD Algebraic Decision Diagram. 1, 3–5, 10–12

BDD Binary Decision Diagram. 3–5

CTHMM Continuous Time Hidden Markov Model. 2
CTMC Continuous Time Markov Chain. 5–7, 9
CuDD Colorado University Decision Diagram. 3, 5

DTMC Discrete Time Markov Chain. 5–7, 9

HMM Hidden Markov Model. 1–3, 10, 11

MDP Markov Decision Process. 2–4

References

[1] R. S. Chavan and G. S. Sable, "An overview of speech recognition using hmm," *International Journal of Computer Science and Mobile Computing*, vol. 2, no. 6, pp. 233–238, 2013.

[2] F. Ciocchetta and J. Hillston, "Bio-pepa: A framework for the modelling and analysis of biological systems," *Theoretical Computer Science*, vol. 410, no. 33-34, pp. 3065–3084, 2009.

[3] R. S. Mamon and R. J. Elliott, *Hidden Markov models in finance*. Springer, 2007, vol. 4.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] F. Somenzi, "Cudd: Cu decision diagram package," *Public Software, University of Colorado*, 1997.

[10] 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.
[11] R. Reynouard et al., "On learning stochastic models: From theory to practice," 2024.

A Cheatsheet

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 $\mathcal{O}$