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

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

Abstract—The Baum-Welch algorithm is a cornerstone of parameter estimation for hidden Markov Models (HMMs), which are widely used in fields such as speech recognition and bioinformatics. Traditional implementations rely on matrix operations, which scale poorly for large models due to quadratic space complexity. This paper introduces the framework for an entirely symbolic implementation of the Baum-Welch algorithm using Algebraic Decision Diagrams (ADDs). By employing ADDs throughout the algorithm — including forward-backward computations and parameter updates — we expect a significant potential reduction in memory requirements and improved scalability. This approach leverages symbolic computation to minimize overhead from explicit state space representations, paving the way for efficient parameter estimation in complex systems. Preliminary findings and challenges in integrating this symbolic framework with tools like the Storm model checker are discussed.

Index Terms—Formal Verification, Parameter Estimation, Decision Diagram, Hidden Markov Model

1 Introduction

Hidden Markov Models (HMMs) are probabilistic models widely used to model systems with hidden states. Applications span speech recognition, bioinformatics, and finance [1–3], where systems exhibit stochastic behavior. HMMs rely on the Markov property, which assumes that the system's future depends only on its current state [4]. This memory-less behavior simplifies analysis and makes HMMs particularly effective in modeling systems where the underlying states are not directly observable.

Parameter estimation is a critical task in analyzing HMMs, as the accuracy of model predictions depends heavily on the quality of the estimated parameters [5]. The Baum-Welch algorithm, a widely used Expectation-Maximization (EM) framework [6], iteratively refines HMM parameters to maximize the likelihood of observed data [7]. However, traditional algorithm implementations rely on matrix representations of the model, which suffer from quadratic space complexity, limiting scalability for large systems [8].

PRISM [9] and Storm [10] provide state-of-the-art model-checking capabilities for Markov models in probabilistic verification. These tools efficiently verify properties such as reachability and safety using symbolic representations, such as Algebraic Decision Diagrams (ADDs), to handle large state spaces. Despite their advanced capabilities, these tools lack native support for parameter estimation, which is crucial for learning models from data. Conversely, tools like Jajapy [11] and SUDD [12] focus on parameter estimation but lack integration with symbolic model-checking frameworks.

By leveraging ADDs in all algorithm steps for the Baum-Welch algorithm, we achieve significant improvements in runtime performance and scalability compared to recursive-based approaches. Our work builds on the symbolic techniques introduced in SUDD [12] but expands their application to the entire parameter estimation process for HMMs.

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

Our contribution are as follows:

- We largely symbolically implement the complete Baum-Welch algorithm, including forward-backward computations and parameter updates[1].
- We have found an error in [12] and corrected it.

1.1 Related Works

Several tools have been developed for parameter estimation in Markov models. Jajapy is a Python-based tool that uses the Baum-Welch algorithm to estimate parameters in various models, including HMMs [11]. It employs a matrix representation of the model and implements an iterative approach based on a recursive definition.

While accessible and straightforward, Jajapy is hindered by the space complexity inherent in its iterative-based calculation. This limitation makes it computationally expensive for large-scale models, as memory requirements grow quadratically with the number of states in the system. While versatile, Jajapy relies on matrix representations, resulting in quadratic space complexity [8].

SUDD builds upon Jajapy's limitations by introducing a symbolic representation for the forward-backward algorithm for calculating Parametric Continuous Time Markov Chain (pCTMC); it does not handle the update step in the Baum-Welch algorithm [12]. Specifically, it leverages ADDs to reduce memory consumption and improve the runtime performance of the Baum-Welch algorithm. By employing ADD-based computations, SUDD significantly improves scalability, making it feasible to handle larger models.

However, the update step in the Baum-Welch algorithm requires SUDD to give an explicit state space representation of the model, which limits the algorithm's scalability.

Our work extends the capabilities of SUDD by implementing the complete Baum-Welch algorithm for HMMs using symbolic representations. Here, we provide a method that estimates

1. https://github.com/AAU-Dat/CuPAAL

parameters efficiently and enables the accurate modeling of complex systems.

2 Preliminaries

This section introduces the concepts and definitions essential for understanding the subsequent sections. We begin by defining the key concepts of a HMM and then describe how it can be represented using matrices.

We then introduce the Baum-Welch algorithm, which estimates the parameters of a HMM from observed data. We describe all the steps involved in the Baum-Welch algorithm, namely the forward-backward algorithm and the parameter's update.

Finally, we describe how the Baum-Welch algorithm can be implemented using matrix operations..

2.1 Hidden Markov Models

Baum and Petrie introduced HMMs in 1966 [4]. HMMs are a class of probabilistic models widely used to describe sequences of observations dependent on some underlying hidden states. These models consist of two main components: observations and hidden states.

The observations are the visible data emitted by the model, while the hidden states represent the underlying process that generates these observations.

HMMs have applications in speech recognition[1], bioinformatics [13], and natural language processing [14]. HMM was chosen as the model of choice for this project due to its versatility and ability to model complex systems.

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 discrete probability distributions over a finite set. The model emits a label $l \in \mathcal{L}$ in state $s \in S$ with probability $\ell(s)(l)$, and transitions from state s to state s' with probability $\tau(s)(s')$, and starts in state s with probability $\pi(s)$.

2.2 Matrix Representation of HMMs

HMMs can be represented using various matrices. The emission function ℓ can be represented as a matrix $\boldsymbol{\omega}$ where $\boldsymbol{\omega}_{s,l} = \ell(s)(l)$. The matrix $\boldsymbol{\omega}$ has the size $|S| \times |\mathcal{L}|$. The sum of each row in the matrix $\boldsymbol{\omega}$ is equal to one, reflecting the total probability of emitting all labels from a given state.

$$\boldsymbol{\omega} = \begin{bmatrix} \ell(s_1)(l_1) & \cdots & \ell(s_1)(l_{|\mathcal{L}|}) \\ \vdots & \ddots & \vdots \\ \ell(s_{|S|})(l_1) & \cdots & \ell(s_{|S|})(l_{|\mathcal{L}|}) \end{bmatrix}$$

The transition function τ can be represented as a stochastic matrix $\boldsymbol{P}$ where $\boldsymbol{P}_{s,s'} = \tau(s)(s')$. The matrix $\boldsymbol{P}$ has the size $|S| \times |S|$. The sum of each row in $\boldsymbol{P}$ is equal to one, reflecting the total probability of transitioning from a given state to all other states.

$$\boldsymbol{P} = \begin{bmatrix} \tau(s_1)(s_1) & \cdots & \tau(s_1)(s_{|S|}) \\ \vdots & \ddots & \vdots \\ \tau(s_{|S|})(s_1) & \cdots & \tau(s_{|S|})(s_{|S|}) \end{bmatrix}$$

The initial distribution π can be represented as a vector $\boldsymbol{\pi}$ where $\boldsymbol{\pi}_s = \pi(s)$. The vector $\boldsymbol{\pi}$ has the size $|S|$. The sum of all elements in $\boldsymbol{\pi}$ is equal to one, reflecting the fact that $\pi \in D(s)$.

$$\boldsymbol{\pi} = \begin{bmatrix} \pi(s_1) \\ \vdots \\ \pi(s_{|S|}) \end{bmatrix}$$

2.3 Observations and Hidden States

A HMM can be understood as a stochastic generator of sequences of hidden states, referred to as a run. Formally, an infinite run is denoted as $\rho = s_1, s_2, \cdots s_n \cdots \in S^\omega$, where S is the set of hidden states. The probability that the HMM, denoted as M, generates a finite prefix of the run $\rho_n = s_1, s_2, \cdots s_n$ is given by:

$$P(\rho_n) = \pi(s_1) \prod_{i=1}^{n-1} \tau(s_i)(s_{i+1}) \tag{1}$$

Where $\pi(s_1)$ is the initial probability of state s_1, and $\tau(s_i)(s_{i+1})$ represents the transition probability from state s_i to state s_{i+1}.

Although the run ρ is not directly observable, it emits a corresponding sequence of observable labels, forming an observation trace $O = l_1, l_2, \cdots l_n \cdots \in L^\omega$, where L is the set of observable labels.

The probability that the hidden state sequence $\rho_n = s_1, s_2, \cdots s_n$ produces the observed trace $O_n = l_1, l_2, \cdots l_n$ is given by:

$$P(O_n|\rho_n) = P(\rho_n) \cdot \prod_{i=1}^{n} \ell(s_i)(l_i) \tag{2}$$

Where $\ell(s_i)(l_i)$ is the emission probability of label l_i from state s_i.

In practical applications, rather than working with infinite sequences, we typically observe a multiset of finite observation sequences, denoted as $\mathcal{O} = \{O_1, O_2, ..., O_N\}$, where each O_i is an observation sequence of labels $\mathbf{o} = o_0, o_1, \cdots, o_{|\mathbf{O}|-1}$.

These observed sequences serve as the input for parameter estimation in the HMM, which involves determining the emission function ℓ, the transition function τ, and the initial distribution π using algorithms like the Baum-Welch algorithm.

2.4 The Baum-Welch Algorithm

The Baum-Welch algorithm is a fundamental method for estimating the parameters of a HMM given a sequence of observations.

These parameters include the emission matrix $\boldsymbol{\omega}$, the transition matrix $\boldsymbol{P}$, and the initial state distribution $\boldsymbol{\pi}$. The algorithm is widely recognized as the standard approach for training HMMs. It was chosen for this project because it can estimate these parameters without prior knowledge of the hidden states that generated the observations [7].

The Baum-Welch algorithm applies the Expectation-Maximization (EM) framework to iteratively improve the likelihood of the observed data under the current model

parameters. It does so until it reaches a set convergence value, which indicates how much the model improves after each iteration. It consists of the following steps:

1) **Initialization:** Begin with a given initial estimates for the HMM parameters $(\boldsymbol{\pi}, \boldsymbol{P}, \boldsymbol{\omega})$.
2) **Expectation Step (E-step):** Compute the expected counts of the latent variables, i.e., the hidden states, based on the observation sequence and the current model parameters. That is, we compute the probabilities of observing the sequence up to time t, given that the HMM is in state s at time t and the probabilities of observing the sequence from time $t+1$ to the end, given that the HMM is in state s at time t.
3) **Maximization Step (M-step):** Update the HMM parameters $(\boldsymbol{\pi}, \boldsymbol{P}, \boldsymbol{\omega})$ to maximize the likelihood of the observed data based on the expected counts computed in the E-step.
4) **Iteration:** Repeat the E-step and M-step until convergence, i.e., when the change in likelihood between iterations falls below a predefined threshold.

The Baum Welch algorithm seeks to estimate the parameters $\boldsymbol{\tau}$, $\boldsymbol{\pi}$, and $\boldsymbol{\omega}$ of a HMM model $\mathcal{M}$, so that it maximizes the likelihood function $\mathfrak{l}(\mathcal{M}|O)$. That is, the probability that the HMM $\mathcal{M}$ has independently generated each observation sequence $O_1, \cdots, O_N$.

Starting with initial hypothesis $\mathbf{x}_0 = (\boldsymbol{\pi}, \boldsymbol{P}, \boldsymbol{\omega})$, the algorithm produces a sequence of parameter estimates $\mathbf{x}_1, \mathbf{x}_2, \cdots$, where each new estimate improves upon the previous one. The process terminates when the improvement in likelihood is sufficiently small, satisfying the convergence criterion:

$$||\mathfrak{l}(x_n, o)|| < \epsilon$$

Where $\mathfrak{l}(x_n, o)$ is the likelihood of all the observations under the parameters x_n, and $\epsilon > 0$ is a small threshold.

The steps of the Baum-Welch algorithm are detailed in the following sections, including the initialization of the HMM parameters, the forward-backward algorithm, and the update step, see subsection 2.5, 2.6, and 2.7 respectively.

2.5 Initialization of HMM Parameters

Before starting the Baum-Welch algorithm, we need to initialize the model parameters: the emission matrix $\boldsymbol{\omega}$, the transition matrix $\boldsymbol{P}$, and the initial state distribution $\boldsymbol{\pi}$.

Since the algorithm is designed to converge iteratively towards a locally optimal parameter set, the initial estimates do not need to be exact. However, reasonable initialization can accelerate convergence and improve numerical stability [15].

As we work with models that we have no prior knowledge of, meaning we do not know the number of observations or what states generated the observations, we cannot initialize the model parameters based on domain knowledge. Therefore, we need to initialize the parameters based on some strategy. If we set a probability to zero, we see in parameter estimation (Equation 6) that the probability will remain zero. Therefore, if no domain knowledge is available, it is better to initialize the parameters with non-zero values. A common approach to initialize these parameters is as one of the following strategies:

1) Random initialization:
   - Assign random probabilities to for the initial state distribution π_s, such that $\sum_{s \in S} \pi_s = 1$.
   - Assign random probabilities to each transition $P_{s,s'}$, such that $\sum_{s' \in S} P_{s,s'} = 1$.
   - Assign random probabilities to each emission $\omega_{s,l}$, such that $\sum_{l \in \mathcal{L}} \omega_{s,l} = 1$ for all $s \in S$.
2) Uniform initialization:
   - Set $\pi_s = \frac{1}{|S|}$ for all $s \in S$.
   - Set $P_{s,s'} = \frac{1}{|S|}$ for all $s, s' \in S$.
   - Set $\omega_{s,l} = \frac{1}{|\mathcal{L}|}$ for all $s \in S, l \in \mathcal{L}$.

We initialize the parameters using random initialization, as it provides a diverse set of initial values that can help avoid local optima. The uniform initialization is the worst choice as it provides the least information. With uniform initialization, any transition is equally likely to be chosen to explain an observation, making the latent states indistinguishable. Therefore, it is not recommended for practical use.

These initialization strategies provide a starting point for the Baum-Welch algorithm, which iteratively refines the model parameters based on the observed data.

2.6 Forward-Backward Algorithm

For a given HMM $\mathcal{M}$, the forward-backward algorithm computes the forward and backward variables, $\alpha_s(t)$ and $\beta_s(t)$, for each observation sequence $o_0, o_1, \cdots, o_{|\mathbf{o}|-1} = \mathbf{o} \in \mathcal{O}$.

The forward variable $\alpha_s(t)$ represents the likelihood of observing the partial sequence $o_0, o_1, \cdots, o_t$ and being in state s at time t, given the model $\mathcal{M}$. The backward variable $\beta_s(t)$ represents the likelihood of observing the partial sequence $o_{t+1}, o_{t+2}, \cdots, o_{|\mathbf{o}|-1}$ given state s at time t and the model $\mathcal{M}$.

The forward variable $\alpha_s(t)$ and backward variable $\beta_s(t)$ can be computed recursively as follows:

$$\alpha_s(t) = \begin{cases} \boldsymbol{\omega}_{s,o_t} \boldsymbol{\pi}_s & \text{if } t = 0 \\ \boldsymbol{\omega}_{s,o_t} \sum_{s' \in S} \boldsymbol{P}_{s's} \alpha_{s'}(t-1) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{3}$$

$$\beta_s(t) = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \sum_{s' \in S} \boldsymbol{P}_{ss'} \boldsymbol{\omega}_{s'}(t+1) \beta_{s'}(t+1) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{4}$$

Here, $\boldsymbol{\omega}_{s,o_t}$ is the likelihood of observing o_t while being in state s at time t given the model $\mathcal{M}$. Formally $\boldsymbol{\omega}_{s,o_t} = \mathfrak{l}(o_t \mid s, \mathcal{M}) = \ell(s)(o_t)$.

The forward-backward algorithm computes the forward and backward variables for each state s and time t in the observation sequence $\mathbf{o}$, providing a comprehensive view of the likelihood of the observed data under the model.

In preparation for later discussions, we would like to draw attention to the fact that the above recurrences can be solved using dynamic programming, which requires $\Theta(|S| \times |\mathbf{o}|)$ space.

2.7 Update Step

The update step refines the parameter values of the HMM model based on the observed data and the forward and backward variables computed in the forward-backward algorithm. Given the forward and backward variables $\alpha_s(t)$ and $\beta_s(t)$, the update step adjusts the parameter values to maximize the likelihood of the observed data.

The update step iteratively refines the parameter values until convergence is reached.

2.7.1 Intermediate Variables

We need to calculate the intermediate variables $\gamma_s(t)$ and $\xi_{ss'}(t)$. $\gamma_s(t)$ represents the expected number of times the model is in state s at time t given that the sequence $\mathbf{o}$ was observed. $\xi_{ss'}(t)$ represents the expected number of transitions from state s to state s' at time t given that the sequence $\mathbf{o}$ was observed.

For a given HMM $\mathcal{M}$, the intermediate variables, $\gamma_s(t)$ and $\xi_{ss'}(t)$, are computed for each observation sequence $o_0, o_1, \dots, o_{|\mathbf{o}|-1} = \mathbf{o} \in \mathcal{O}$. These variables are computed as follows:

$$\gamma_s(t) = \frac{\alpha_s(t)\beta_s(t)}{\sum_{s' \in S} \alpha_{s'}(t)\beta_{s'}(t)} \tag{5}$$

In Equation 5, the numerator is the product of the forward variable $\alpha_s(t)$ and the backward variable $\beta_s(t)$, representing the joint probability of observing the entire sequence given that the model passed by state s at time t. The denominator represents the probability of the observation sequence.

$$\xi_{ss'}(t) = \frac{\alpha_s(t)\boldsymbol{P}_{ss'}\boldsymbol{\omega}_{s'}(t+1)\beta_{s'}(t+1)}{\sum_{s''} \alpha_{s''}(t)\beta_{s''}(t)} \tag{6}$$

In Equation 6, the numerator is the joint probability of observing the sequence given that the model transitions from state s to state s' at time t. The denominator represents the probability of the observation sequence.

The terms $\gamma_s(t)$ and $\xi_{ss'}(t)$ are normalized to ensure they represent probabilities. For $\gamma_s(t)$, this involves dividing by the total probability across all states at time t, while for $\xi_{ss'}(t)$, normalization occurs over all possible transitions at time t.

2.7.2 Parameter Update

The parameter update step refines the models parameter values based on the intermediate variables $\gamma_s(t)$ and $\xi_{ss'}(t)$. THe update step adjusts the parameter values to maximize the likelihood of the observed data given the model $\mathcal{M}$.

Once $\gamma_s(t)$ and $\xi_{ss'}(t)$ are computed for all states s, s' and all time steps t for every observation sequence, the model parameters can be updated to maximize the expected log-likelihood.

Transition Probabilities ($\boldsymbol{P}$): We update the transition probabilities based on the expected number of transitions between states:

$$\boldsymbol{P}_{ss'} = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \xi_{ss'}(t)}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{7}$$

The numerator sums the expected number of transitions from state s to state s' over all time steps. The denominator sums the expected number of times the model is in state s over all time steps, ensuring $P_{ss'}$ is normalized across all s'.

Emission Probabilities ($\boldsymbol{\omega}$): We update the emission probabilities based on the expected emissions of state s and the corresponding observations, meaning the probability of observing the specific label o in state s. This is given by:

$$\boldsymbol{\omega}_{s,l} = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)[\![o_t = l]\!]}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{8}$$

$[\![o_t = l]\!]$ is an indicator function that returns 1 if the observation at time t is label l and 0 otherwise. The numerator sums $\gamma_s(t)$ for all time steps t where the observed value $o_t = l.$, meaning the model is in state s and emits the label l.

The denominator sums $\gamma_s(t)$ for all time steps t where the model is in a given state s.

Initial Probabilities ($\boldsymbol{\pi}$): We update the initial probabilities based on the expected number of times we start in state s.

$$\pi_s = \gamma_s(1) \tag{9}$$

We can then update the parameters $(\boldsymbol{\pi}, \boldsymbol{P}, \boldsymbol{\omega})$ given the model $\mathcal{M}$ by maximizing the expected log-likelihood of the observed data under the model. The update step iteratively refines the parameter values until convergence is reached.

2.8 Matrix Operations

The Baum-Welch algorithm can be implemented using matrix operations to efficiently compute the forward and backward variables, intermediate variables, and parameter updates.

Given a HMM $\mathcal{M}$ with parameters $\boldsymbol{\omega}$, $\boldsymbol{P}$ and $\boldsymbol{\pi}$, and an observation sequence $\mathbf{o}$, the forward and backward variables $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ can be computed using matrix operations as follows:

$$\boldsymbol{\alpha}_t = \begin{cases} \boldsymbol{\omega}_0 \circ \boldsymbol{\pi} & \text{if } t = 0 \\ \boldsymbol{\omega}_t \circ \left(\boldsymbol{P}^\top \boldsymbol{\alpha}_{t-1}\right) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{10}$$

$$\boldsymbol{\beta}_t = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \boldsymbol{P}\left(\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1}\right) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{11}$$

Here $\circ$ represents the Hadamard (point-wise) matrix multiplication, $\boldsymbol{P}^\top$ denotes the transpose of the matrix $\boldsymbol{P}$, and $\mathbb{1}$ is a column vector of ones, and $\boldsymbol{\omega}_t$ is the column vector that represents the label we are observing at time t of matrix $\boldsymbol{\omega}$. The resulting vectors $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ for each time step t are then related to $\alpha_s(t)$ and $\beta_s(t)$ for some s by:

$$\boldsymbol{\alpha}_t = \begin{bmatrix} \alpha_{s_0}(t) \\ \vdots \\ \alpha_{s_{|S|-1}}(t) \end{bmatrix}, \ \boldsymbol{\beta}_t = \begin{bmatrix} \beta_{s_0}(t) \\ \vdots \\ \beta_{s_{|S|-1}}(t) \end{bmatrix} \tag{12}$$

We denote $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, as the matrices gathering the columns α_t and β_t for $t = 1 \cdots |\mathbf{o}| - 1$

$$\boldsymbol{\alpha} = \begin{bmatrix} \alpha_{s_0 t_0} & \cdots & \alpha_{s_{|S|-1} t_{|\mathbf{o}|-1}} \\ \vdots & \ddots & \vdots \\ \alpha_{s_0 t_0} & \cdots & \alpha_{s_{|S|-1} t_{|\mathbf{o}|-1}} \end{bmatrix} \text{ and } \boldsymbol{\beta} = \begin{bmatrix} \beta_{s_0 t_0} & \cdots & \beta_{s_{|S|-1} t_{|\mathbf{o}|-1}} \\ \vdots & \ddots & \vdots \\ \beta_{s_0 t_0} & \cdots & \beta_{s_{|S|-1} t_{|\mathbf{o}|-1}} \end{bmatrix} \tag{13}$$

γ and ξ can be expressed in terms of matrix operations as follows:

$$\boldsymbol{\gamma}_t = \left(\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti})\right)^{-1} \cdot \boldsymbol{\alpha}_t \circ \boldsymbol{\beta}_t \tag{14}$$

$$\boldsymbol{\xi}_t = \left(\left(\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti})\right)^{-1} \cdot \boldsymbol{P}\right) \circ \left(\boldsymbol{\alpha}_t \otimes \left(\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1}\right)\right) \tag{15}$$

Here $\otimes$ represents the Kronecker (block) matrix multiplication, $\cdot$ denotes the scalar product, and $^{-1}$ denotes the elementwise inverse of a matrix.

We can simplify $\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\boldsymbol{\beta}_{ti})$ as, the sum does not depend on t:

$$\sum_{i=1}^{|\mathbf{o}|-1} (\boldsymbol{\alpha}_{ti}\, \boldsymbol{\beta}_{ti}) = \sum_{i=1}^{|\mathbf{o}|-1} \boldsymbol{\alpha}_{|\mathbf{o}|-1i} \tag{16}$$

$$= \mathbb{1}^T\, \boldsymbol{\alpha}_{|\mathbf{o}|-1} \tag{17}$$

Here $\mathbb{1}^T$ is a row vector of ones, and $\boldsymbol{\alpha}_{|\mathbf{o}|-1}$ is the last column of the matrix $\boldsymbol{\alpha}$. From this we get:

$$\boldsymbol{\gamma}_t = (\mathbb{1}^T\, \boldsymbol{\alpha}_{|\mathbf{o}|-1})^{-1} \cdot \boldsymbol{\alpha}_t \circ \boldsymbol{\beta}_t \tag{18}$$

$$\boldsymbol{\xi}_t = ((\mathbb{1}^T\, \boldsymbol{\alpha}_{|\mathbf{o}|-1})^{-1} \cdot\ \boldsymbol{P}) \circ (\boldsymbol{\alpha}_t \otimes (\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1})) \tag{19}$$

The resulting vectors $\boldsymbol{\gamma}_t$ and $\boldsymbol{\xi}_t$ for each time step t are then related to $\gamma_s(t)$ and $\xi_{ss'}(t)$ for some s, s' by:

$$\boldsymbol{\gamma}_t = \begin{bmatrix} \gamma_{s_0}(t) \\ \vdots \\ \gamma_{s_{|S|-1}}(t) \end{bmatrix},\ \boldsymbol{\xi}_t = \begin{bmatrix} \xi_{s_0 s_0}(t) & \cdots & \xi_{s_0 s_{|S|-1}}(t) \\ \vdots & \ddots & \vdots \\ \xi_{s_{|S|-1} s_0}(t) & \cdots & \xi_{s_{|S|-1} s_{|S|-1}}(t) \end{bmatrix} \tag{20}$$

We can then update the parameters with matrix operations as follows:

$$\boldsymbol{P} = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet \boldsymbol{\xi} \tag{21}$$

$$\boldsymbol{\omega}_s(o) = (\mathbb{1} \oslash \boldsymbol{\gamma}) \bullet (\sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\gamma}_t \otimes \mathbb{1}_{yt}^{|\mathbf{o}|-1}) \tag{22}$$

$$\boldsymbol{\pi} = \boldsymbol{\gamma}_1 \tag{23}$$

Here $\oslash$ denotes Hadamard division (elementwise division) and $\bullet$ denotes the Katri-Rao product (column-wise Kronecker product). In the formulas above, $\mathbb{1}$ denotes a column vector of ones, $\mathbb{1}_{yt}$ denotes a column vector with $|\mathcal{L}|$ rows, with all elements set to zero except for the element at the index where $o_t = l$ which is set to one.

γ and ξ are the sum of the respective vectors over all time steps t:

$$\boldsymbol{\gamma} = \sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\gamma}_t \text{ and } \boldsymbol{\xi} = \sum_{t=1}^{|\mathbf{o}|-1} \boldsymbol{\xi}_t \tag{24}$$

3 Implementation

In this section, we will discuss the implementation of the project. We will start by discussing ADDs and their advantages and disadvantages. We will discuss how we transition from vectors and matrices to ADDs. Then we will then discuss the Colorado University Decision Diagram (CuDD) library and the Storm model checker. Finally, we will discuss the implementation of the matrix operations using ADDs.

3.1 Algebraic Decision Diagrams

Algebraic Decision Diagrams (ADDs) generalize Binary Decision Diagrams (BDDs) [16]. Unlike BDDs, which are limited to binary values ({0,1}), ADDs extend this data structure by allowing terminal nodes to store arbitrary values, such as integers or real numbers. This flexibility makes them well-suited for applications like probabilistic model checking and optimization and for representing matrices.

In an ADD, each path from the root to a terminal node represents a unique variable assignment, with terminal nodes storing the associated function values.

Advantages of ADDs:

- **Compact Representation:** ADDs exploit data redundancies, providing a compact representation for many functions. Shared subgraphs reduce memory usage, especially when storing large, structured data, such as matrices.
- **Canonical Form:** For a fixed variable ordering, ADDs are canonical. This property simplifies equivalence checking and ensures consistency in symbolic manipulations.
- **Efficient Operations:** Operations such as addition and multiplication can be performed directly on ADDs, often avoiding explicit enumeration of the underlying data.

Disadvantages of ADDs:

- **Variable Ordering Sensitivity:** The size and efficiency of ADDs depend heavily on the chosen variable ordering. A suboptimal ordering can lead to exponentially larger diagrams, reducing performance.
- **Overhead in Construction:** Constructing an ADD involves significant overhead compared to simpler data structures like arrays or hash tables, particularly for small datasets, where the benefits of compactness may not be realized.
- **Complexity for Certain Functions:** While ADDs work well for structured data, highly unstructured or random data can result in large, inefficient diagrams. In such cases, other data structures may be more suitable.

ADDs can symbolically encode row and column indices as binary variables, enabling efficient storage and manipulation of matrix data. This approach is particularly advantageous for sparse or structured matrices, where patterns and redundancies can be exploited [16].

In this work, ADDs provide a foundation for symbolic matrix operations like addition, multiplication, and Kronecker product. Their compact representation and efficiency make them a powerful tool for handling complex parameterized computations.

3.2 Transition to ADDs

The first step in the implementation is to transition from vectors and matrices to ADDs. This conversion leverages the compact and efficient representation of ADDs to perform operations symbolically.

To convert a vector or matrix into an ADD, we encode the indices as binary variables and the values as terminal nodes. It is important to note that all the variables in the ADD have to

be consistent, meaning a variable in the ADD has to be in a consistent position in all the paths to the terminal nodes.

When a matrix is represented as an ADD, it has to be square to satisfy what is known as the power of 2 rule [16]. This is due to the binary nature of the diagram and the interleaving of the row and column variables. This scheme for representing matrices as ADDs is called a partitioning of the matrix [16]. For a complete discussion on how to pad matrices to obtain a square version see [16].

3.2.1 Vector to ADD

We will illustrate the conversion from a vector to an ADD using an example. Consider the following vector:

$$V = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

In an ADD, each layer corresponds to one binary variable (or bit) in the encoding of an index. For a vector of size n, where $n = 2^k$, the binary representation of the column indices requires k bits each. In the case of the vector V, the vector has 4 elements, so it requires $4 = 2^2$ bits to represent the indices.

We can create the binary representation of the indices by using the binary representation of the numbers from 0 to k.

The binary representation of the vector V entries is shown in Table 1.

TABLE 1
Binary encoding of a vector V of size 4

Vector Index	Value	Binary Encoding
0	1	00
1	2	01
2	3	10
3	4	11

The binary encodings determine the structure of the decision diagram, where each entry in the vector is stored as a terminal node. The binary representation of their indices dictates the paths to these nodes. The root node represents the first bit of the binary encoding, with subsequent layers corresponding to the remaining bits.

The ADD representation of the vector V is shown in Figure 1. The structure of matrices represented as ADDs is similar to vectors but has two sets of binary variables for row and column indices.

The terminal nodes store the matrix entries, with paths determined by the binary representation of the row and column indices. The ADD variables are interleaved, with the row and column indices alternating in the path from the root to the terminal nodes.

3.3 CUDD

The CuDD library [17] is a powerful tool for implementing and manipulating decision diagrams, including BDDs and ADDs.

In this project, the CuDD library stores ADDs and performs operations on them. Its optimized algorithms and efficient memory management allow us to handle large and complex matrices symbolically, leading to significant performance improvements over traditional methods.

The CuDD library is implemented in C, ensuring high-performance execution. Therefore, it can be used in C++ programs, which we use for this paper.

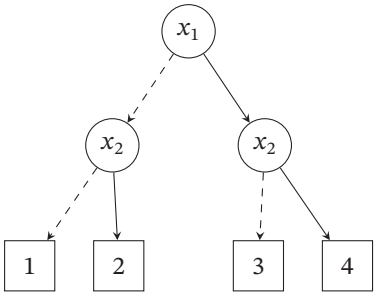

Fig. 1. ADD representation of a vector V of size 4

3.4 Storm

Storm is a versatile, open-source probabilistic model checking tool designed to verify the correctness and properties of stochastic models [10]. It supports a wide range of probabilistic models, including Markov Chains (MCs) and Markov Decision Processs (MDPs).

It does not include HMMs in its supported models, but a HMM can be encoded as a MC [18]. We can also implement a MC as a HMM, where the emission matrix is treated as a constant and each row represents a Dirac distribution (a.k.a. indicator vector) pointed at the state label.

Storm allows users to analyze models efficiently by computing various quantitative properties, such as probabilities, expected rewards, or long-run averages.

The Storm model checker is widely recognized as a state-of-the-art verification tool for probabilistic models. It is known for its efficiency and scalability. Storm uses the CuDD and Sylvan libraries to represent models symbolically, which users can choose when running the tool.

3.5 Matrix Operations Using ADDs

The matrix operations implemented are matrix transpose, matrix addition, matrix multiplication, Hadamard product, Hadamard division, Kronecker product, and Khatri-Rao product. To show examples for each operation in the following subsections, we have two matrices A and B, shown here:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

In CuDD a *DdManager* manages the ADDs and their operations. A *DdNode* is used to represent the entire ADD and the variables of the ADD, which are the row and column indices in the ADD.

3.5.1 Matrix Transpose

The matrix transpose operation is implemented by swapping the row and column variables in the ADD. Specifically, for each path in the ADD representing an entry (i, j), the roles of the row index i and column index j are exchanged. The terminal nodes (values of the matrix entries) remain unchanged. The transpose of matrix A is:

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

In CuDD, the function we use for transposing an ADD is implemented as *Cudd_addSwapVariables(DdManager * dd,*

*DdNode * f, DdNode ** x, DdNode ** y, int n)*, where f is the ADD to be transposed, x and y are the set variables to be swapped and n is the size of the variables to be swapped.

3.5.2 Matrix Addition

The matrix addition operation is implemented by adding the terminal nodes of two ADDs while keeping the structure of the row and column indices consistent. This process involves:

1) Traversing the paths of both ADDs simultaneously.
2) Summing the values at the terminal nodes where the row and column indices match.

The resulting ADD represents the element-wise sum of the two matrices. The sum of matrices A and B is:

$$A + B = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

In CuDD, the function we use for adding two ADDs is implemented as *Cudd_addApply(DdManager * dd, Cudd_addPlus(), DdNode * f, DdNode * g)*, where f and g are the two ADDs to be added and, *Cudd_addPlus()* is the function that is used to add the two ADDs.

3.5.3 Scalar Product

The scalar product operation is implemented by multiplying each element in a matrix by a scalar value. The scalar product of matrix A by a scalar 2 is:

$$2 \times A = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

In CuDD, the function we use for scalar multiplication is implemented as *Cudd_addApply(DdManager * dd, Cudd_addTimes(), DdNode * f, DdNode * g)*, where f is the ADD to be multiplied by the scalar, and g is an ADD representing the scalar value, and *Cudd_addTimes()* is the function that is used to multiply the ADD by the scalar.

3.5.4 Matrix Multiplication

The matrix multiplication operation is implemented symbolically using the dot product of the row and column indices. For an ADD, the process is as follows:

1) For each pair of rows in the first matrix and columns in the second matrix, the corresponding elements are multiplied.
2) The products are summed along the shared index, combining them into the final terminal nodes of the resulting ADD.

The hierarchical structure of the ADD ensures that only relevant paths are explored. The product of matrices A and B is

$$A \times B = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

We use the function *Cudd_addMatrixMultiply(DdManager dd, DdNode A, DdNode B, DdNode **z, int nz)* in CuDD to multiply two ADDs. The function takes two ADDs A and B. z is the set of variables that are dependent on the columns in A and the rows in B.

A is assumed to have the same number of columns as B has rows, nz is the number of z variables. We must rename the variables in the ADDs to do the matrix multiplication, in this case, matrix A's column variables and matrix B's row variables.

3.5.5 Hadamard Product

The Hadamard product operation is implemented by pairwise multiplication on corresponding terminal nodes of two ADDs for each matching row-column index pair (i, j). This process involves:

1) The values from both ADDs are multiplied.
2) The resulting product is stored in the terminal node of the new ADD.

The structure of the indices remains unchanged. The Hadamard product of matrices A and B is:

$$A \circ B = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

In CuDD, the function used for Hadamard product is implemented as *Cudd_addApply(DdManager * dd, Cudd_addTimes(), DdNode * f, DdNode * g)*, where f and g are two ADDs to be multiplied and *Cudd_addTimes()* is the function that is used to apply elementwise multiplication.

3.5.6 Hadamard Division

The Hadamard division operation is implemented similar to the Hadamard product, but with division instead of multiplication. See subsubsection 3.5.5 for more details. The Hadamard division of matrices A and B is:

$$A \oslash B = \begin{bmatrix} 0.2 & 0.3333 \\ 0.4286 & 0.5 \end{bmatrix}$$

The Hadamard division is implemented in CuDD by *Cudd_addApply(DdManager * dd, Cudd_addDivide(), DdNode * f, DdNode * g)*, where f and g are two ADDs to be divided and *Cudd_addDivide()* is the function used to apply elementwise division.

3.5.7 Kronecker Product

The Kronecker product operation is implemented by symbolically expanding the indices and terminal nodes of two matrices. This operation results in a new ADD with dimensions equal to the product of the dimensions of the input matrices.

The Kronecker product generalizes the outer product, multiplying each element of the first matrix by the entire second matrix.

For a specific entry in the first matrix A at row i and column j, with value a_{ij}, the second matrix B is scaled by a_{ij}, and its indices are adjusted as follows:

1) Multiply the values of B by a_{ij}.
2) The row indices of B are shifted by $i \times \text{rows}(B)$ in the new ADD.
3) The column indices of B are shifted by $j \times \text{columns}(B)$ in the new ADD.
4) The scaled values are stored in the new ADD.

Here rows(B) and columns(B) represents the number of rows and columns in matrix B, respectively.

This operation is not natively supported in the CuDD library, we have not implemented it as a custom function.

The Kronecker product of matrices A and B is:

$$A \otimes B = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

3.5.8 Khatri-Rao Product

The Khatri-Rao product operation is implemented by combining rows of the first matrix with the corresponding rows of the second matrix. For each row index i:

1) The elements of row i in the first matrix are multiplied element-wise with the entire row i in the second matrix.
2) The resulting row is constructed symbolically within the ADD.

This operation is not natively supported in the CuDD library, we have not implemented it as a custom function. The Khatri-Rao product of matrices A and B is:

$$A \bullet B = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

The resulting ADD has the dimensions equal to the sum of the dimensions of the two matrices, as in the Kronecker product, see subsubsection 3.5.7.

4 Experiments

This section describes the experiment that would evaluate the proposed symbolic implementation of the Baum-Welch algorithm in CuPAAL. The experiment was designed to compare CuPAAL with another implementation: Jajapy, which uses a recursive approach to the Baum-Welch algorithm. The experiment would be conducted to answer the following research question:

- **Question:** How does the scalability of CuPAALs symbolic implementation compare to Jajapy as the number of model states increases?

This question would evaluate the computational efficiency of the symbolic implementation in CuPAAL. The insights gained would highlight the strengths and potential weaknesses of the symbolic approach to the Baum-Welch algorithm.

4.1 Experimental Setup

The experiment would evaluate how each implementation scales with increasing model states. We would conduct a single experiment to evaluate the proposed method. The experiment was intended to be run on the machine with the specifications shown in Table 2.

TABLE 2
Machine Specifications from AAU Strato

Component	Specification
CPU	AMD EPYC 16-core
RAM	64 GB
DISK	50 GB

The following implementations would used for comparison.

- **Jajapy**: A recursive implementation of the Baum-Welch algorithm [11].
- **CuPAAL**: The fully symbolic implementation proposed in this work.

The chosen implementations are either fully recursive to fully symbolic approaches, providing a comprehensive comparison of methodologies.

4.2 Experiment Dataset

The experiment would use synthetic datasets; The dataset would have five different labels {"A", "B", "C", "D", "E"} and the same observation sequence for all runs of the experiment. We would use one observation sequence of length 20, as our implementation is not set up to handle multiple observation sequences. The number of states in the model would be varied from 20 to 1020, with increments of 100 states.

The initial model would be generated with random parameters, and the observation sequence would be the same for all experiment runs. The model would start in the same initial state, generating the transition and emission matrices randomly. The emission matrix would be generated so that one state can emit a maximum of three different labels.

All implementations would tested on the same dataset to ensure fair comparison.

4.3 Experimental Procedure

The experiment would assess how the runtime of each implementation changes as the number of states in a synthetic model incrementally increases from 20 to 1020. The process would be done as follows:

1) Load the initial model and observation sequence.
2) Use each implementation of the Baum-Welch algorithm to estimate model parameters and record the runtime.
3) Increase the number of states in the model and load new initial parameters.
4) Repeat steps 2-3 for each implementation until the maximum number of states is reached.

A link to the specific experiment can be found in[2].

Scalability would be evaluated by examining runtime trends as model size increases. This would provide insight into how each implementation scales with larger models and is critical for understanding the practical applicability of symbolic versus recursive-based implementations.

Results would be visualized using plots to illustrate trends and highlight differences in scalability between implementations.

4.4 Discussion of Metrics and Methods

The runtime would be the primary evaluation metric, as it directly reflects the computational efficiency of each implementation when handling larger models. By plotting runtime alongside the number of states, we would aim to identify patterns and scalability limits for each approach.

We wanted to track the implementations' memory consumption, as this is a critical factor in the Baum-Welch algorithm's scalability, and symbolic approaches are expected to be more memory-efficient. However, tracking memory consumption is challenging and is therefore left for future work. Instead, we focus on runtime as the primary metric, as it provides a clear and comparable measure of computational efficiency.

2. https://github.com/AAU-Dat/P7-sudd/blob/P9-experiments/scaling_experiment.py

5 Discussion

This section will cover the paper's discussion. This paper aimed to study how implementing a symbolic representation of the Baum-Welch algorithm could impact the expected runtime and model accuracy.

This work extends the study conducted in [12] by developing an entirely symbolic representation using ADDs at every point of the Baum-Welch algorithm where matrices are traditionally used.

Unfortunately, no significant findings were achieved in our study. Several factors contributed to this outcome. One such factor was the issue of extending Storms implementation of CuDD to handle operations, which required computing the Baum-Welch algorithm using ADDs.

Initially, we aimed to read PRISM files representing our models using Storm. However, this process proved more difficult than expected. The Storm library proved particularly challenging to navigate. Although its extensive documentation was initially considered beneficial, it often led to confusion and hindered progress.

As a result, initial models were instead written manually, and random values were used as data for creating these models and observed data. If we had used manual initialization of models earlier, further testing and experimentation could have been conducted, leading to a more complete implementation.

Furthermore, the library choice for manipulating ADDs was not thoroughly considered, as we intended to extend the work in [12]. Efforts should have been dedicated to evaluating alternative libraries that work with ADDs. Examples of such libraries include Sylvan, also written in C and provides parallel execution capabilities. While CuDD worked effectively in [12], further consideration of alternative options could have yielded better results.

While working on our implementation, we studied the symbolic implementation of the Forward-Backward algorithm in [12]. Here, we identified an error in the implementation that suggested significant runtime improvements. This error involved a miscalculation caused by misaligned ADD variables in the matrix multiplication of the forward step.

Although the results and values remained correct, the computation time increased unnecessarily. The paper [12] had already demonstrated significant runtime improvements compared to the non-symbolic comparison implementations. Removing this unnecessary calculation could further enhance runtime performance.

Additionally, the implementation in [12] was only partially symbolic. While the Forward and Backward calculations used symbolic representations, other components relied on traditional matrices. This resulted in frequent conversions between ADDs and matrices, increasing computational overhead. An entirely symbolic implementation would likely reduce this overhead and improve efficiency.

These observations motivate the continuation of studying symbolic representations of the Baum-Welch algorithm. They also highlight the potential for further performance improvements through error correction and adopting an entirely symbolic approach.

6 Conclusion

In this paper, we have attempted to develop and test a symbolic implementation of the Baum-Welch algorithm.

Although the implementation is almost complete, experiments have not yet been conducted. As such, it is not possible to draw any conclusions based on results. However, an error has been found in prior work [12] in the forward step, indicating that a symbolic implementation has even better runtime than initially thought.

We have made small steps toward an entirely symbolic implementation of the Baum-Welch algorithm for HMMs. We have also described the necessary matrix calculation representations, most of which are remarkably similar to Markov Chains.

We know that the Kronecker product is implementable in CuDD from [19], and by extension, the Khatri-Rao product should be equally possible. Unfortunately, our attempts so far have been unsuccessful.

7 Future Works

This section outlines potential future works and extensions of the symbolic Baum-Welch algorithm.

7.1 Issues With the Implementation

Our first step would be implementing the Kronecker and Katri-Rao products using a symbolic approach. These operations are essential for the Baum-Welch algorithm, as they are used to update the transition and emission matrices. Implementing these operations would complete the symbolic approach, enabling the entire Baum-Welch algorithm to be executed symbolically.

Following this, since no experiments were conducted on the implementation, this would be a good starting point for future work, as we could compare the performance of the symbolic approach to the recursive Jajapy implementation. This comparison would assess the scalability of the symbolic implementation as the number of model states increases.

Runtime and memory usage would be core parameters to evaluate, providing insights into the computational efficiency of the symbolic approach in the context of the Baum-Welch algorithm. Additionally, we intend to test the symbolic approach on concrete models of varying sizes and complexities.

To deepen our analysis, we would compare the symbolic implementation with a C++-based recursive implementation of the Baum-Welch algorithm. In [12], the authors implemented the forward-backward algorithm in C and a symbolic approach in C, which showed that the symbolic approach was only slightly faster than the C implementation. Therefore, comparing the complete symbolic approach to a recursive approach in C++ would be interesting. We could better understand the symbolic approach's performance by benchmarking both implementations within the same programming language.

Numerical stability is a significant concern in implementations of the Baum-Welch algorithm, and steps must be taken to mitigate this. To do this, we suggest implementing the algorithm in the log semiring.

One final improvement to the algorithm would be to extend it to handle multiple observation sequences, as is done in [11].

7.2 Storm

A key aspect of our project was integrating the symbolic Baum-Welch algorithm into the Storm model checker. We planned to leverage the Storm Parser Library to enable the parsing of models and observation sequences for the Baum-Welch algorithm. This integration would expand the algorithm's accessibility and applicability, as Storm is widely used in model-checking research and practice.

Another area of improvement would be implementing the Baum-Welch algorithm using sparse matrices since we saw these in our work with Storm. We would evaluate their respective runtime and memory efficiency by comparing this traditional approach to the symbolic implementation.

7.3 Modelling

We would also explore extending the symbolic approach to MDPs. Given the close relationship between MDPs and HMMs, this extension would allow the symbolic Baum-Welch algorithm to estimate MDP parameters. Such advancements would broaden its utility in diverse applications.

Finally, we would investigate alternative symbolic representations beyond ADDs. We would look into further options, such as Multi-valued Decision Diagrams (MDDs), which support multiple edges per node. Comparative experiments will assess these representations' runtime and memory usage against ADDs, offering further insights into their relative strengths and weaknesses.

8 Acknowledgements

We want to express our sincere gratitude to Giovanni Bacci for his invaluable supervision and guidance throughout this project. We also extend this gratitude to our co-supervisor Raphaël Reynouard, the developer of Jajapy, whose knowledge of Markov Models and the Baum-Welch algorithm has been immensely helpful.

Additionally, we acknowledge that AI tools, namely ChatGPT and Grammarly, have been used to aid the production of this paper.

Acronyms

ADD Algebraic Decision Diagram. 1, 5–10

BDD Binary Decision Diagram. 5, 6

CuDD Colorado University Decision Diagram. 5–9

HMM Hidden Markov Model. 1–4, 6, 9, 10

MC Markov Chain. 6

MDP Markov Decision Process. 6, 10

pCTMC Parametric Continuous Time Markov Chain. 1

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 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.

[5] G. Bacci, A. Ingólfsdóttir, K. G. Larsen, and R. Reynouard, "Mm algorithms to estimate parameters in continuous-time markov chains," 2023. arXiv: 2302.08588 [cs.LG].

[6] P. Kenny, T. Stafylakis, P. Ouellet, V. Gupta, and M. J. Alam, "Deep neural networks for extracting baum-welch statistics for speaker recognition.," in *Odyssey*, vol. 2014, 2014, pp. 293–298.

[7] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi, "An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition," *Bell System Technical Journal*, vol. 62, no. 4, pp. 1035–1074, 1983.

[8] R. I. Davis and B. C. Lovell, "Comparing and evaluating hmm ensemble training algorithms using train and test and condition number criteria," *Formal Pattern Analysis & Applications*, vol. 6, pp. 327–335, 2004.

[9] M. Kwiatkowska, G. Norman, and D. Parker, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 585–591.

[10] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk, "The probabilistic model checker storm," *International Journal on Software Tools for Technology Transfer*, pp. 1–22,

[11] R. Reynouard, A. Ingólfsdóttir, and G. Bacci, "Jajapy: A learning library for stochastic models," in *International Conference on Quantitative Evaluation of Systems*, Springer, 2023, pp. 30–46.

[12] L. E. Hansen, C. Ståhl, D. R. Petersen, S. Aaholm, and A. M. Jakobsen, "Symbolic parameter estimation of continuous-time markov chains," *AAU Student Projects*, 2023, https : / / github . com / AAU - Dat / P7 - scalable - parameter-estimation-for-markov-models. eprint: https: / / kbdk - aub . primo . exlibrisgroup . com / permalink / 45KBDK_AUB/a7me0f/alma9921651457905762.

[13] V. De Fonzo, F. Aluffi-Pentini, and V. Parisi, "Hidden markov models in bioinformatics," *Current Bioinformatics*, vol. 2, no. 1, pp. 49–61, 2007.

[14] H. Murveit and R. Moore, "Integrating natural language constraints into hmm-based speech recognition," in *International Conference on Acoustics, Speech, and Signal Processing*, IEEE, 1990, pp. 573–576.

[15] B. Benyacoub, I. ElMoudden, S. ElBernoussi, A. Zoglat, and M. Ouzineb, "Initial model selection for the baum-

welch algorithm applied to credit scoring," in *Modelling, Computation and Optimization in Information Systems and Management Sciences: Proceedings of the 3rd International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences-MCO 2015-Part II*, Springer, 2015, pp. 359–368.

[16] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," *Formal methods in system design*, vol. 10, pp. 171–206, 1997.

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

[18] 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.

[19] D. K. Houngninou and M. A. Thornton, "Implementation of switching circuit models as transfer functions," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2016, pp. 2162–2165. DOI: 10.1109/ISCAS.2016.7539009.

Appendix A
Compiling in draft

You can also compile the document in draft mode. This shows todos, and increases the space between lines to make space for your supervisors feedback.