Symbolic Parameter Estimation of Continuous-Time Markov Chains

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

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

Index Terms—Formal Verification, Parameter Estimation, Decision Diagram

1 Introduction

Markov models are a class of probabilistic models that are used to describe the evolution of a system over time. A Markov model has the Markov property, which states that the future behavior of the system depends only on its current state and not on its past history [1]. This property simplifies analysis by focusing only on the present state, making Markov models especially useful for systems where memory-less behavior is a reasonable assumption.

Markov models are widely used in various fields, such as biology, finance, and computer science, to model systems that exhibit stochastic behavior [2–5]. As such, their analysis has a wide range of applications.

An example of a Markov model, is a simple weather model, if today is sunny, there might be an 80% chance of sun tomorrow and a 20% chance of rain. Similarly, if today is rainy, there might be a 70% chance of rain tomorrow and a 30% chance of sun.

Model checking is a technique used to verify the correctness of Markov models by comparing the predictions of the model with observed data. Model checking is widely used in the verification of Markov models, where the model is analyzed to check if it satisfies certain properties [6]. It ensures reliability and correctness in critical systems, from traffic controls to industrial automation and communication protocols [6]. It is also used to check if the model satisfies certain properties, such as reachability, can we reach a desired state and safety properties, can we avoid going a specific sequence of states.

A real world example of model checking is the verification of a traffic light system, where the model is analyzed to check if the traffic lights are working correctly. For reachability, we can ask: *can a traffic light system always cycle back to green after being red?*. For safety properties we can ask, *can a traffic light system avoid having both lights green at the same time?*.

There exists several tools for model checking, such as PRISM [7] and Storm [8], which are widely used in the verification of Markov models. These tools use symbolic representations to represent the model and perform the operations required for model checking. The limitation of these tools is that they do not support parameter estimation, which makes them unsuitable for learning the parameters of the model from data.

Parameter estimation is a crucial step in the analysis of Markov models, as the analysis of the model depends on the accuracy of the estimated parameters, particularly when in a timing and probabilistic behaviour [9]. Parameter estimation is the process of estimating the parameters of the model from observed data, which is used to make predictions about the system's behavior.

These parameters are used to ensure that the model accurately represents the system's behavior and dynamics and to make accurate predictions about the system's future behavior. Accurate parameter estimation is essential for making reliable predictions and validating model behavior, with applications ranging from healthcare diagnostics to network security [9].

The Baum-Welch algorithm is a widely used method for estimating the parameters of Markov models [10]. The algorithm uses the Expectation-Maximization (EM) framework to iteratively update the parameters of the model until convergence [11]. The Baum-Welch algorithm is computationally expensive for large models, as it uses matrices to represent the model, which has a space complexity that grows quadratically with the number of states in the model. This makes the algorithm computationally expensive for large models, as the memory requirements grow rapidly with the size of the model [12].

Addressing these challenges requires innovative techniques, such as symbolic representations, which reduce memory consumption while preserving accuracy.

1.1 Related Works

Jajapy [13] is a Python-based tool designed for estimating parameters in parametric models using the Baum-Welch algorithm. It employs a matrix representation of the model and implements the necessary operations for parameter estimation through standard matrix computations.

While accessible and straightforward, Jajapy is hindered by the space complexity inherent in its recursive-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.

SUDD [14] builds upon the limitations of Jajapy by introducing a symbolic representation for the forward-backward algorithm. Specifically, it leverages Algebraic Decision Diagrams (ADDs) to reduce memory consumption and improve the

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

runtime performance of the Baum-Welch algorithm. By employing ADD-based computations, SUDD provides a significant improvement in scalability, making it feasible to handle larger models.

However, the implementation is limited to a subset of the Baum-Welch algorithm, focusing primarily on forward-backward computations without addressing the full parameter estimation process.

In this paper, we extend the work of SUDD by utilizing ADDs to represent the full Baum-Welch algorithm. Our approach not only inherits the scalability benefits of ADDs but also implements the complete parameter estimation process. Additionally, we compare our implementation with both the original Jajapy, SUDD and an extended version of SUDD using the log-semiring framework, which improves numerical stability in computations.

PRISM [7] is a widely used probabilistic model checker designed to verify the correctness of Markov models. It employs symbolic representations such as ADDs to efficiently represent and manipulate large-scale models, enabling the verification of properties like reachability and safety.

For example, PRISM can determine whether a traffic light system will always cycle back to green after being red or verify that conflicting light signals are avoided. However, PRISM does not support parameter estimation, limiting its use to model verification rather than learning the parameters required for accurate system predictions.

Storm [8] is another state-of-the-art probabilistic model checker that shares many similarities with PRISM. Like PRISM, it uses symbolic representations to handle large models efficiently and focuses on verifying properties of Markov models. Storm has been optimized for scalability and flexibility, supporting a wide range of model types and verification tasks. Despite these strengths, Storm also lacks support for parameter estimation, making it unsuitable for tasks requiring the inference of model parameters from observed data.

Our work bridges the gap between parameter estimation tools (e.g. Jajapy and SUDD) and model checking tools (e.g. PRISM and Storm). By integrating scalable symbolic representations into the full Baum-Welch algorithm, we provide a method that not only estimates parameters efficiently but also enables the accurate modeling of complex systems. This integration of parameter learning with symbolic computation addresses a critical limitation in the current landscape of tools for Markov models.

2 Preliminaries

We introduce some preliminary notions and notations, which will be used in the rest of the paper. The parameter estimation algorithm studied here focuses on Continuous Time Markov Chains (CTMCs). We will first introduce the definition of a CTMC and Parametric Continuous Time Markov Chain (pCTMC) then present the Baum-Welch algorithm, which is used to estimate the parameters of a CTMC.

2.1 Continuous-Time Markov Chains

In stochastic systems, state transitions are governed by two key aspects: timing and transition probabilities. In CTMCs, these aspects are explicitly modeled as separate but interrelated components:

1) **Dwell Time**: The time spent in a state before transitioning to another state. This is a random variable governed by an exponential distribution, characterized by the exit rate of the state.
2) **Transition Probability**: Once the dwell time elapses, the system transitions to a new state. The destination state is determined probabilistically based on the rates of outgoing transitions.

By decoupling these two aspects, CTMCs provide a flexible framework for modeling systems where the timing of transitions and the likelihood of transitioning to specific states are influenced by different factors.

Definition 1 (CTMC). A CTMC is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, R, \pi)$, where:

- S is a finite set of states.
- $\mathcal{L}$ is a finite set of labels.
- $\ell : S \to \mathcal{L}$ is a labeling function, which assigns a label $\mathcal{L}$ to each state.
- $R : S \times S \to \mathbb{R}_{\geq 0}$ is the rate function. The model transitions from state s to state s' with rate $R(s, s')$.
- π is the initial distribution, the model starts in state s with probability $\pi(s)$.

2.1.1 Key Properties

1) The time spent in s, known as the dwell time, is exponentially distributed with rate:

$$E(s) = \sum_{s' \in S} R(s, s') \tag{1}$$

2) The probability of transitioning to state s' is given by:

$$P(s' \mid s) = \frac{R(s, s')}{E(s)} \tag{2}$$

Transitions are independent of the time spent in the current state. If there are multiple possible transitions, a race condition occurs, and the first transition to complete determines the next state.

2.1.2 Discrete Time Markov Chains

If the dwell time is disregarded or assumed to be uniform across all states, the timing of transitions becomes irrelevant, and the CTMC simplifies into a Discrete Time Markov Chain (DTMC). In this case, transitions are described by the probabilities $P(s' \mid s)$ of moving to state s' from state s.

2.1.3 Observations in CTMCs

An execution of the CTMC is represented by a sequence of states and dwell times in **Paths** $\subseteq (S \times \mathbb{R}_{>0} \cup \{\emptyset\})^{\omega}$ where ω symbolizes infinite execution. Similarly, an Observation, also called a Trace, is represented by a sequence of labels and dwell times in **Observations** $\subseteq (\mathcal{L} \times \mathbb{R}_{>0} \cup \{\emptyset\})^{\omega}$.

For a finite observation $\mathbf{o} = o_0, o_1, \dots, o_{|\mathbf{O}|-1} = (l_0, \tau_0), (l_1, \tau_1), \dots, (l_{|0|-1}), \emptyset \in$ **Observations**:

- $l_t \in \mathcal{L}$ is the label observed during the t-th transition.
- $\tau_t \in \mathbb{R}_{>0} \cup \{\emptyset\}$ is the dwell time observed during the t-th transition.

if $\tau_t = \emptyset$ for all transitions in a sequence, the sequence is untimed, effectively ignoring dwell times.

Intuitively, observations link the observed labels (l_t) and dwell times (τ_t) to the underlying states of the CTMC. This connection is captured through the likelihood function $\omega_s(t)$, which combines label matching with timing information.

The likelihood of observing an observation $o_t = (l_t, \tau_t)$ in state s is given by the function $\omega_s(i)$, which links the observation to the model's dynamics. This function ensures that transitions are appropriately weighted by the likelihood of the observed data.

2.2 Parametric Continuous Time Markov Chains

In practice, the rate function R in a CTMC is often unknown and must be estimated from observed data. In systems with complex or uncertain dynamics, pCTMCs extend CTMCs by introducing parameters into the model's rate functions. These parameters allow for the representation of families of CTMCs rather than a single, fixed model. Like CTMCs, pCTMCs are governed by two key aspects:

- **Dwell Time**: The time spent in a state before transitioning. This is determined by the parametric exit rate, which depends on the specific values of the parameters.
- **Transition Probability**: After the dwell time elapses, the system transitions to a new state. The probability of transitioning to a particular state is derived from the parametric rates of all outgoing transitions from the current state.

For a full description of pCTMC we refer [9].

Definition 2 (pCTMC). A pCTMC is a tuple $\mathcal{P} = (S, \mathcal{L}, \ell, R, \pi)$, where:

- $S, \mathcal{L}, \ell, \pi$ are defined as for CTMCs.
- $R : S \times S \to (\mathbb{R}^n_{\geq 0} \rightharpoonup \mathbb{R}_{\geq 0})$ is a parametric transition rate function that maps transitions to polynomial expressions over a vector of parameters $\mathbf{x} = (x_1, \dots, x_n)$.

In this definition, the rate function under $\mathbf{x}$, $R(s, s'; \mathbf{x})$ determines the rate at which the system transitions from state s to state s' dependent on the parameter values. Note that the partial function $(\mathbb{R}^n_{\geq 0} \rightharpoonup \mathbb{R}_{\geq 0})(\mathbf{x})$ is the actual evaluation of the rate.

2.2.1 Key Properties

- For a given state s, the parametric dwell time is exponentially distributed with rate:

$$E(s; \mathbf{x}) = \sum_{s' \in S} R(s, s'; \mathbf{x}), \tag{3}$$

where the sum depends on the current values of the parameters $\mathbf{x}$.

- The parametric transition probability is given by:

$$P(s' \mid s; \mathbf{x}) = \frac{R(s, s'; \mathbf{x})}{E(s; \mathbf{x})}. \tag{4}$$

These parametric formulations allow a single pCTMC to represent a broad class of CTMCs, where the specific model instance is determined by fixing the parameter values.

2.3 Baum-Welch Algorithm

The Baum-Welch algorithm is an iterative method used to estimate the parameters of a pCTMC from observed data. GThe algorithm aims to find the parameter values that maximize the likelihood of the observed data under the model. This process is based on the Expectation-Maximization (EM) framework and involves two main steps:

1) **Expectation Step (E-step)**: Compute the expected values of the latent variables, which are the unobserved state sequences corresponding to the observations.
2) **Maximization Step (M-step)**: Update the parameter values to maximize the likelihood of the observed data, using the expected latent variables computed in the E-step.
3) Repeat the E-step and M-step until convergence.

The Baum-Welch algorithm is particularly useful for estimating the parameters of a pCTMC when the underlying state sequence is unknown or partially observed.

Given a multiset of observations $\mathcal{O}$ and initial parameters $\mathbf{x}_0$, the Baum-Welch algorithm estimates the parameters of a pCTMC $\mathcal{P}$ by iteratively improving the current hypothesis $\mathbf{x}_n$ using the previous estimate $\mathbf{x}_{n-1}$ until a convergence criterion is met. A hypothesis refers to a specific set of values for the parameters $\mathbf{x}$.

Each iteration of the algorithm produces a new hypothesis, denoted as $\mathbf{x}_n$, which is the algorithm's current best guess for the parameter values based on the observed data. The algorithm consists of three main steps: the forward-backward procedure, the update step, and the convergence criterion. The Baum-Welch algorithm iteratively refines the parameters until the improvement between successive iterations falls below a predefined threshold. This is typically evaluated using a convergence criterion such as:

$$||\mathbf{x}_n - \mathbf{x}_{n-1}|| < \epsilon \tag{5}$$

where $\epsilon > 0$ is a small threshold, and $\mathbf{x}_n$ denotes the parameter values at the n-th iteration.

The algorithm stops when the change in parameters is sufficiently small, indicating that the model has converged to a local maximum of the likelihood function. The parameter estimation procedure is outlined in Algorithm 1.

```
ESTIMATE-PARAMETERS($\mathcal{P}, \mathbf{x}_0, \mathcal{O}$)
$\mathbf{x} \leftarrow \mathbf{x}0$
while $\neg$CRITERION($\mathbf{x}n - 1, \mathbf{x}n$)
    $\mathbf{x}_{n-1} \leftarrow \mathbf{x}_n$
    $(\alpha, \beta)$ = FORWARD-BACKWARD($\mathcal{P}(\mathbf{x}_n), \mathcal{O}$)
    $\mathbf{x}_n$ = UPDATE($\mathcal{P}(\mathbf{x}_n), \mathcal{O}, \alpha, \beta$)
return $\mathbf{x}_n$
```

Algorithm 1. Parameter estimation procedure [14].

Starting with initial parameters $\mathbf{x}_0$, the parameter estimation procedure iteratively improves the current hypothesis $\mathbf{x}_n$ using the previous estimate $\mathbf{x}_{n-1}$ until a specified criterion for convergence is met. The specifics of the FORWARD-BACKWARD and UPDATE procedures are detailed in subsection 2.4 and subsection 2.5 from [14].

2.4 The Forward-Backward Algorithm

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

The forward variable $\alpha_s(t)$ represents the likelihood of observing the partial sequence $o_0, o_1, \dots, o_t$ and being in state s at time t, given the model $\mathcal{M}$:

$$\alpha_s(t) = l(o_0, o_1, \dots, o_t, S_t = s \mid \mathcal{M}) \tag{6}$$

The backward variable $\beta_s(t)$ represents the likelihood of observing the partial sequence $o_{t+1}, o_{t+2}, \dots, o_{|\mathbf{o}|-1}$ given state s at time t and the model $\mathcal{M}$:

$$\beta_s(t) = l(o_{t+1}, o_{t+2}, \dots, o_{|\mathbf{o}|-1} \mid S_t = s, \mathcal{M}) \tag{7}$$

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

$$\alpha_s(t) = \begin{cases} \omega_s(0)\ \pi_s & \text{if } t = 0 \\ \omega_s(t) \sum_{s' \in S} P_{s's} \alpha_s(t-1) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{8}$$

$$\beta_s(t) = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \sum_{s' \in S} P_{ss'} \omega_{s'}(t+1) \beta_{s'}(t+1) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{9}$$

Here, $\omega_s(t)$ is the likelihood of observing o_t given that the current state at time t is s and the model $\mathcal{M}$, expressed as $\omega_s(t) = l(o_t \mid S_t = s, \mathcal{M})$. The function $\omega_s(t)$ incorporates observed labels and dwell times into the likelihood computation, linking the data to the model's dynamics. It ensures the forward-backward algorithm appropriately weights transitions based on their likelihood given the observations.

For pCTMCs, $\omega_s(t)$ is given for some observation $o_t = (l_t, \tau_t)$ by[1]:

$$\omega_s(t) = \begin{cases} [\![\ell(s) = l_t]\!] E(s)\ e^{-E_s \tau_t} & \text{if } \tau_t \neq \emptyset \\ [\![\ell(s) = l_t]\!] & \text{if } \tau_t = \emptyset \end{cases} \tag{10}$$

Here:

- $[\![\ell(s) = l_t]\!]$ is an indicator function, equal to 1 if the label $\ell(s)$ of state s matches the observed label l_t, and 0 otherwise.
- $E(s) = \sum_{s' \in S} R(s, s')$ is the total exit rate for state s.

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.

2.5 The Update Algorithm

The update algorithm refines the parameter values of a pCTMC based on the observed data and the forward and backward variables computed in the forward-backward procedure. Given the forward and backward variables $\alpha_s(t)$ and $\beta_s(t)$, the update algorithm aims to maximize the likelihood of the observed data by adjusting the parameter values.

The update algorithm iteratively refines the parameter values $\mathbf{x}$ by maximizing the expected log-likelihood of the observed data under the model. The update step is based on the expected sufficient statistics of the latent variables, which are the unobserved state sequences corresponding to the observations.

2.5.1 Intermediate Variables

We need to intermediate variables $\gamma_s(t)$ and $\xi_{ss'}(t)$, $\gamma_s(t)$ represent the expected number of times the model is in state s at time t and $\xi_{ss'}(t)$ represent the expected number of transitions from state s to state s' at time t. These variables are computed as follows:

$$\gamma_s(t) = \frac{\alpha_s(t)\beta_s(t)}{\sum_{s' \in S} (\alpha_{s'}(t)\beta_{s'}(t))} \tag{11}$$

In Equation 11, 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 sequence up to time t and the model being in state s at time t. The denominator normalizes the probabilities across all states $s' \in S$ to ensure that the sum of $\gamma_s(t)$ over all s equals 1.

$$\xi_{ss'}(t) = \frac{\alpha_s(t) P_{ss'} \omega_{s'}(t+1) \beta_{s'}(t+1)}{\sum_{s'' \in S} \left(\sum_{s''' \in S} (\alpha_{s''}(t)\ P_{s''s'''}\ \omega_{s'''}(t+1)\ \beta_{s'''}(t+1))\right)} \tag{12}$$

In Equation 12, the numerator is the joint probability of observing the sequence up to time t and the model transitioning from state s to state s' at time t. The denominator normalizes the probabilities across all states $s'' \in S$ to ensure that the sum of $\xi_{ss'}(t)$ over all s' equals 1.

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 likelihood across all states at time t, while for $\xi_{ss'}(t)$, normalization occurs over all possible transitions at time t.

2.5.2 Parameter Update

The parameter update step refines the parameter values $\mathbf{x}$ based on the expected sufficient statistics of the latent variables. The update algorithm aims to maximize the expected log-likelihood of the observed data under the model by adjusting the parameter values.

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 (P): We update the transition probabilities based on the expected number of transitions between states:

$$P_{s \to s'} = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \xi_{ss'}(t)}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{13}$$

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'.

Observation Probabilities (ω): We update the observation probabilities based on the expected occupancy of state s and the corresponding observations, meaning the likelihood of observing a specific value o in state s. It is important to note in forward-backwards we use $\omega_s(t)$ to compute $\alpha_s(t)$ and $\beta_s(t)$, we look at all the properties we can see at time t:

$$\omega_s(o) = \frac{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t) [\![o_t = o]\!]}{\sum_{t=1}^{|\mathbf{o}|-1} \gamma_s(t)} \tag{14}$$

[1] Note that $o_{|\mathbf{o}|-1} = (l_{|\mathbf{o}|-1}, \emptyset)$ is always true.

The numerator sums $\gamma_s(t)$ for all time steps t where the observed value $o_t = o$. The denominator ensures the observation probabilities are normalized for state s.

Initial Probabilities (π): We update the initial probabilities based on the expected occupancy of state s at $t = 1$:

$$\pi_s = \gamma_s(1) \tag{15}$$

We can then update the parameters $\mathbf{x}$ by maximizing the expected log-likelihood of the observed data under the model. The update algorithm iteratively refines the parameter values until convergence is reached.

2.6 Matrix Representation

For an arbitrary total ordering $s_0 \leq s_1 \leq \cdots \leq s_{|S|-1}$ of the states in S let:

$$\boldsymbol{P} = \begin{bmatrix} P_{s_0 s_0} & \cdots & P_{s_0 s_{|S|-1}} \\ \vdots & \ddots & \vdots \\ P_{s_{|S|-1} s_0} & \cdots & P_{s_{|S|-1} s_{|S|-1}} \end{bmatrix} \tag{16}$$

$$\boldsymbol{\omega}_t = \begin{bmatrix} \omega_{s_0}(o_t) \\ \vdots \\ \omega_{s_{|S|-1}}(o_t) \end{bmatrix}, \ \boldsymbol{\pi} = \begin{bmatrix} \pi_{s_0} \\ \vdots \\ \pi_{s_{|S|-1}} \end{bmatrix} \tag{17}$$

Then α and β can be described in terms of matrix operations as follows:

$$\boldsymbol{\alpha}_t = \begin{cases} \boldsymbol{\omega}_0 \circ \boldsymbol{\pi} & \text{if } t = 0 \\ \boldsymbol{\omega}_t \circ \left(\boldsymbol{P}^\top \boldsymbol{\alpha}_{t-1}\right) & \text{if } 0 < t \leq |\mathbf{o}| - 1 \end{cases} \tag{18}$$

$$\boldsymbol{\beta}_t = \begin{cases} \mathbb{1} & \text{if } t = |\mathbf{o}| - 1 \\ \boldsymbol{P}\,(\boldsymbol{\beta}_{t+1} \circ \boldsymbol{\omega}_{t+1}) & \text{if } 0 \leq t < |\mathbf{o}| - 1 \end{cases} \tag{19}$$

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. The resulting vectors $\boldsymbol{\alpha}_t$ and $\boldsymbol{\beta}_t$ for each moment t are then related to $\alpha_s(t)$ and $\beta_s(t)$ for some s by:

$$\boldsymbol{\alpha}_t = \begin{bmatrix} \alpha_{s_0}(t) \\ \vdots \\ \alpha_{s_{|S|-1}}(t) \end{bmatrix}, \ \boldsymbol{\beta}_t = \begin{bmatrix} \beta_{s_0}(t) \\ \vdots \\ \beta_{s_{|S|-1}}(t) \end{bmatrix} \tag{20}$$

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

$$\boldsymbol{\gamma}_t = (\sum_{i=1}^{|\mathbf{o}|-1} (\alpha_{ti}\ \beta_{ti}))^{-1} \cdot \alpha_t \circ \beta_t \tag{21}$$

$$\boldsymbol{\xi}_t = ((\sum_{i=1}^{|\mathbf{o}|-1} (\alpha_{ti}\ \beta_{ti}))^{-1} \cdot\ \boldsymbol{P}) \circ (\alpha_t \otimes (\beta_{t+1} \circ \boldsymbol{\omega}_{t+1})) \tag{22}$$

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

We can rewrite $\sum_{i=1}^{|\mathbf{o}|-1}(\alpha_{ti}\beta_{ti})$ as:

$$\sum_{i=1}^{|\mathbf{o}|-1} (\alpha_{ti}\ \beta_{ti}) = \sum_{i=1}^{|\mathbf{o}|-1} \alpha_{|\mathbf{o}|-1i} \tag{23}$$

$$= \mathbb{1}^T\ \alpha_{|\mathbf{o}|-1} \tag{24}$$

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

So we get:

$$\boldsymbol{\gamma}_t = (\mathbb{1}^T\ \alpha_{|\mathbf{o}|-1})^{-1} \cdot \alpha_t \circ \beta_t \tag{25}$$

$$\boldsymbol{\xi}_t = ((\mathbb{1}^T\ \alpha_{|\mathbf{o}|-1})^{-1} \cdot\ \boldsymbol{P}) \circ (\alpha_t \otimes (\beta_{t+1} \circ \boldsymbol{\omega}_{t+1})) \tag{26}$$

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

$$\boldsymbol{\gamma}_t = \begin{bmatrix} \gamma_{s_0}(t) \\ \vdots \\ \gamma_{s_{|S|-1}}(t) \end{bmatrix}, \ \boldsymbol{\xi}_t = \begin{bmatrix} \xi_{s_0 s_0}(t) & \cdots & \xi_{s_0 s_{|S|-1}}(t) \\ \vdots & \ddots & \vdots \\ \xi_{s_{|S|-1} s_0}(t) & \cdots & \xi_{s_{|S|-1} s_{|S|-1}}(t) \end{bmatrix} \tag{27}$$

We can update the parameters with matrix operations as follows:

$$\boldsymbol{P} = (\mathbb{1} \oslash \gamma) \bullet \xi \tag{28}$$

$$\omega_s(o) = (\mathbb{1} \oslash \gamma) \bullet (\sum_{t=1}^{|\mathbf{o}|-1} \gamma_t \otimes \mathbb{1}_{yt}^{|\mathbf{o}|-1}) \tag{29}$$

$$\boldsymbol{\pi} = \boldsymbol{\gamma}_1 \tag{30}$$

Where $\oslash$ denotes Hadamard division (elementwise division) product 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 row vector of ones, γ and ξ are the sum of the respective vectors over all time steps t:

$$\gamma = \sum_{t=1}^{|\mathbf{o}|-1} \gamma_t,\ \xi = \sum_{t=1}^{|\mathbf{o}|-1} \xi_t \tag{31}$$

3 Implementation

In this section, we will discuss the implementation of the project. We will start by discussing the tools used in the implementation, followed by the transition from matrices to ADDs. Finally, we will discuss the implementation of the matrix operations using ADDs.

3.1 CUDD

The Colorado University Decision Diagram (CUDD) library [15] is a powerful tool for implementing and manipulating decision diagrams, including Binary Decision Diagrams (BDDs) and ADDs. ADDs are compact representations of functions, often used to handle large state spaces symbolically and efficiently.

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, but it also ensures it can be used in C++ programs.

TABLE 1
Binary encoding of a vector V of size 4

Vector Index	Value	Binary Encoding
1	1	0000
2	2	0001
3	3	0010
4	4	0011

3.2 Storm

Storm is a versatile probabilistic model checking tool designed to verify the correctness and properties of stochastic models [8]. A key feature of Storm is its ability to take a model as input and represent it symbolically, allowing for efficient manipulation and analysis. It does this by converting the model into a symbolic representation, such as a BDD or an ADD, using the CUDD library.

We use this symbolic representation to get acces to the ADD representation of the model, which is then used to perform the matrix operations using CUDD.

3.3 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 into an ADD, the vector must first be interpreted as a square matrix. This step ensures compatibility with the ADD representation, which organizes data hierarchically.

Consider the following vector:

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

This vector corresponds to a matrix of size 4×4.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Representing this as an ADD requires $log_2(n)$ layers of nodes, where n is the size of the matrix. In this case, the vector has 4 elements, so it requires $log_2(4 \times 4) = 4$ layers of nodes, as the binary representation of 4 indices spans 2 bits. The binary representation of the vector entries is shown in Table 1, the rest of the matrix indices is filled with zeros.

The ADD representation of this vector is shown in Figure 1. The binary encodings determine the structure of the decision diagram, where each entry in the vector is stored as a terminal node. The paths to these nodes are dictated by the binary representation of their indices.

The conversion of a matrix to an ADD is similar to that of a vector, but with an additional layer of nodes to represent the rows. The ADD can however be reduced as shown in Figure Figure 2. This reduction is done by removing the dublicated terminating nodes, removing the redundant nodes and merging the nodes with the same children.

3.4 Matrix operations using ADDs

The matrix operations are implemented using ADDs. The matrix operations implemented are matrix transpose, matrix addition, matrix multiplication, Hadamard product, Hadamard division, Kronecker product and Khatri-Rao product.

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

and

$$B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

are used as examples in the following sections.

Their ADDs representations are shown in Figure 3 and Figure 4 respectively.

3.4.1 Matrix Transpose

The matrix transpose is implemented by swapping the nodes in the ADD, so that the rows become columns and the columns become rows. The transpose of matrix A is

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

The ADD representation of the transpose is shown in Figure 5.

3.4.2 Matrix addition

Matrix addition is implemented by adding the ADDs terminal nodes together. The sum of matrices A and B is

$$A + B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

The ADD representation of the sum is shown in Figure 6.

3.4.3 Matrix multiplication

Matrix multiplication is implemented by performing the dot product of the rows and columns of the matrices. The product of matrices A and B is

$$A \times B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

The ADD representation of the product can be seen in Figure 7.

3.4.4 Hadamard product

The Hadamard product is implemented by multiplying the corresponding elements of the matrices together. The Hadamard product of matrices A and B is

$$A \circ B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \circ \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

The ADD representation of the Hadamard product is shown in Figure 8.

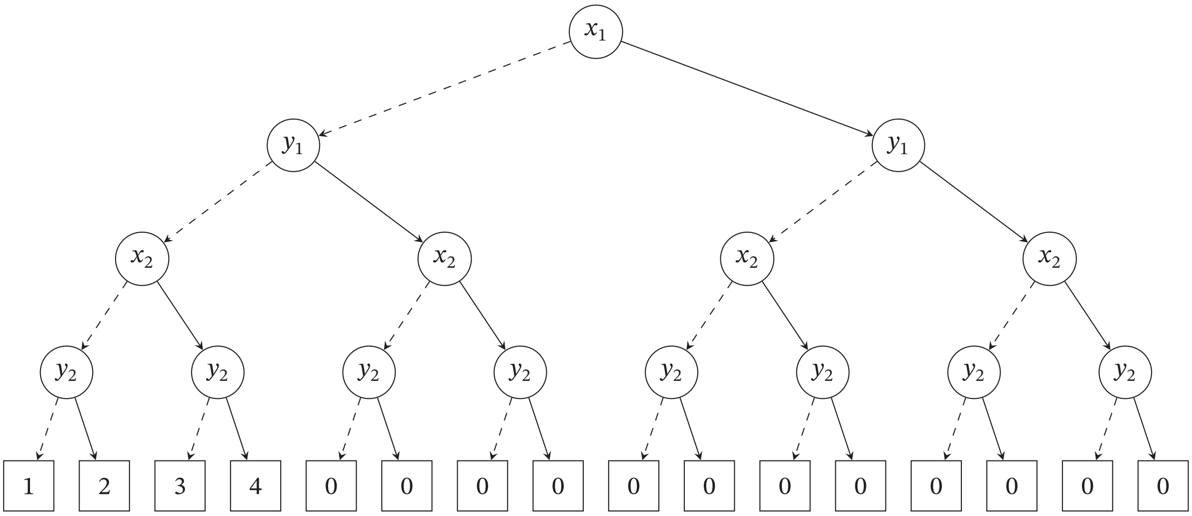

Fig. 1. Vector V represented as an ADD

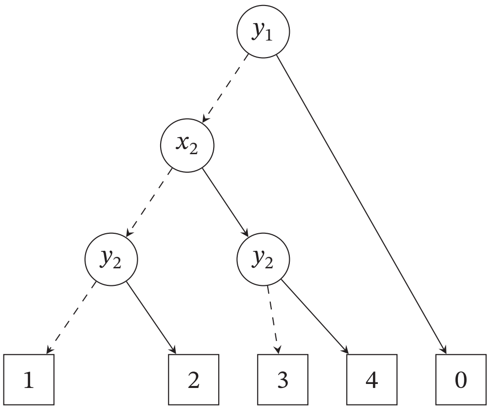

Fig. 2. Reduced ADD of matrix V

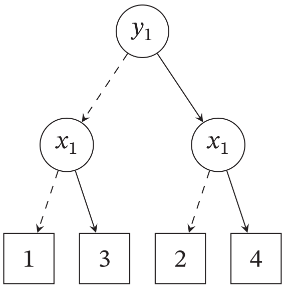

Fig. 5. Matrix A Transposed

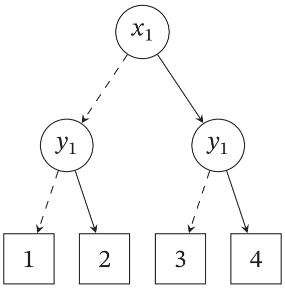

Fig. 3. Matrix A in ADD

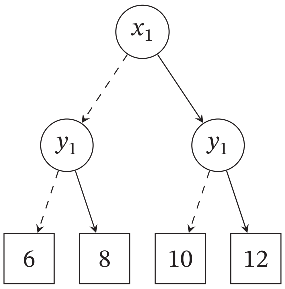

Fig. 6. Sum of matrices A and B

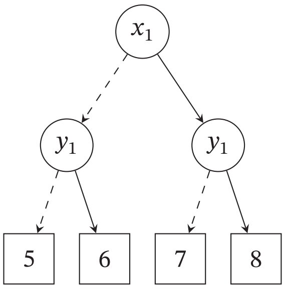

Fig. 4. Matrix B in ADD

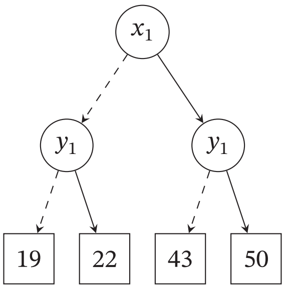

Fig. 7. Product of matrices A and B

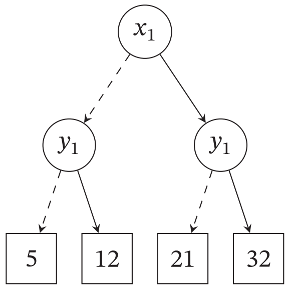

Fig. 8. Hadamard product of matrices A and B

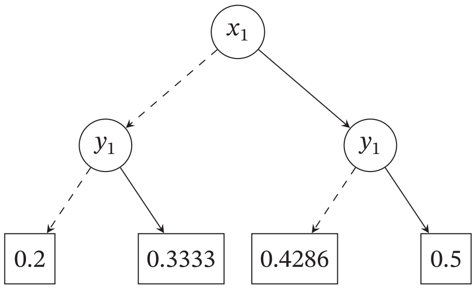

Fig. 9. Hadamard division of matrices A and B

3.4.5 Hadamard division

The Hadamard division is implemented by dividing the corresponding elements of the matrices. The Hadamard division of matrices A and B is

$$A \oslash B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \oslash \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.3333 \\ 0.4286 & 0.5 \end{bmatrix}$$

The ADD representation of the Hadamard division is shown in Figure 9.

3.4.6 Kronecker product

The Kronecker product is implemented by multiplying each element of the first matrix by the second matrix. The Kronecker product of matrices A and B is

$$A \otimes B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

The ADD representation of the Kronecker product is shown in Figure 10.

3.4.7 Khatri-Rao product

The Khatri-Rao product is implemented by multiplying one element of the first matrix by the row of the second matrix. The Khatri-Rao product of matrices A and B is

$$A \bullet B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \bullet \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

The ADD representation of the Khatri-Rao product is shown in Figure 11.

4 Definitions

Definition 3 (Markov Chain). A Markov chain 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.
- $\ell : S \rightarrow \mathcal{L}$ is a labeling function, which assigns a label to each state.
- $\tau : S \rightarrow \mathcal{D}(S)$ is a transition function. The model moves from state s to state s' with probability $\tau(s, s')$.
- π: is the initial distribution, the model starts in state s with probability $\pi(s)$.

Intuitively, a Markov chain is a model that starts in a state s with probability $\pi(s)$, and then transitions to a new state s' with probability $\tau(s, s')$. The model continues to transition between states according to the transition function.

Definition 4 (Hidden Markov Model). A Hidden Markov Model (HMM) is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, \tau, \pi)$, where $S, \mathcal{L}, \tau, \pi$ are defined as above, and:

- $\ell : S \rightarrow D(\mathcal{L})$ is the emission function. The model emits a label l in state s with probability $\ell(s, l)$.

Intuitively, an HMM is a model that starts in a state s with probability $\pi(s)$, then emits a label l with probability $\ell(s, l)$, and transitions to a new state s' with probability $\tau(s, s')$. The model continues to emit labels and transition between states according to the emission and transition functions.

Definition 5 (Markov Decision Process). A Markov Decision Process (MDP) is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, A, \{\tau_a\}_{a \in A}, \pi)$ where $S, \mathcal{L}, \ell, \pi$ are defined as above, and:

- A is a finite nonempty set of actions.
- $\tau_a : S \rightarrow \mathcal{D}(S)$ is a transition function for each action $a \in A$. The model moves from state s to state s' with probability $\tau_a(s, s')$ when action a is taken.

Intuitively, an MDP is a model that starts in a state s with probability $\pi(s)$, then emits a label $\ell(s)$ and, it can recieve an action $a \in A$ and transition to a new state s' with probability $\tau_a(s, s')$.

4.1 Continuous-Time

In the previous definitions, the models are discrete-time models, where time advances in fixed, regular steps. For example, in a discrete-time Markov chain, the system transitions between states at each step or tick of a clock, and the probability of moving from one state to another is governed by the transition function $\tau(s, s')$. This means that transitions can only happen at specific time intervals (e.g., after every second, every minute, etc.).

In contrast, continuous-time models allow transitions to occur at any time, rather than at fixed intervals. The time between transitions is variable and follows a continuous distribution. This introduces the concept of transition rates rather than discrete transition probabilities.

Definition 6 (Continuous-Time Markov Chain). A Continuous-Time Markov Chain (CTMC) is a tuple $\mathcal{M} = (S, \mathcal{L}, \ell, R, \pi)$, where $S, \mathcal{L}, \ell, \pi$ are defined as above, and:

- $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is the rate function. The model transitions from state s to state s' with rate $R(s, s')$.

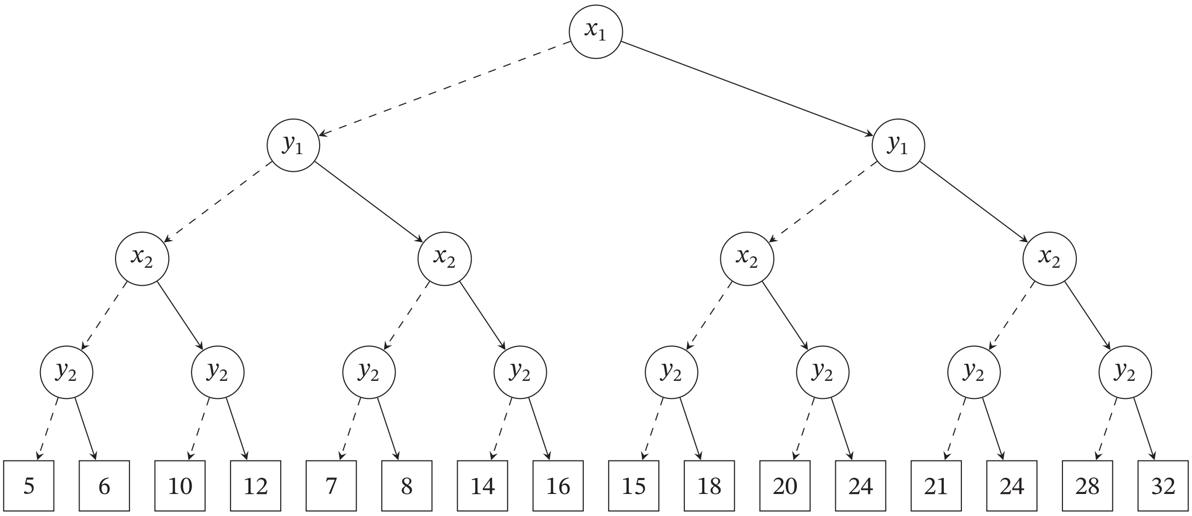

Fig. 10. Kronecker product of matrices A and B

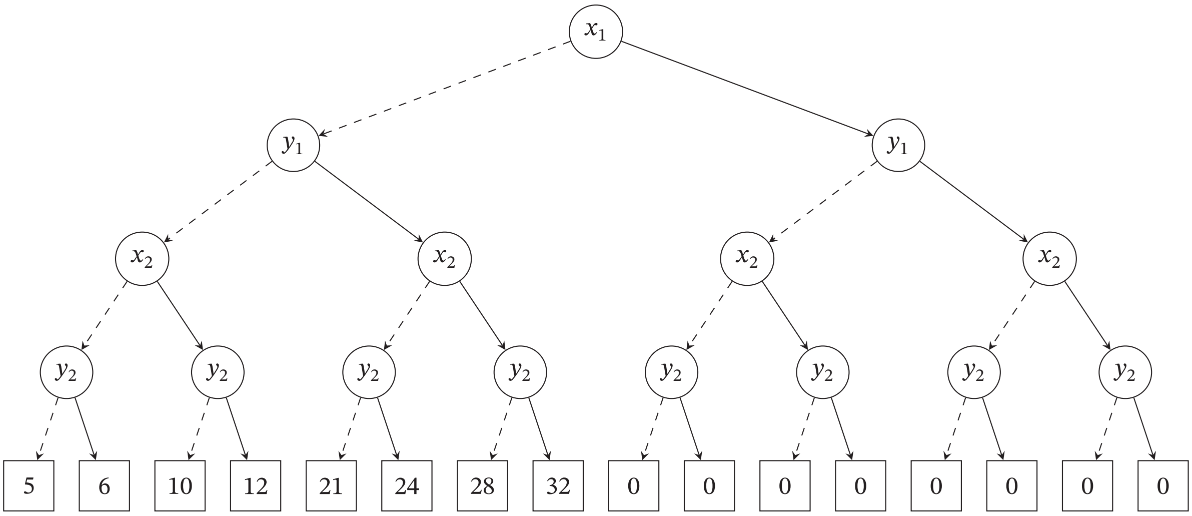

Fig. 11. Khatri-Rao in ADD

For two states s and s', $R(s, s')$ gives the rate at which the system moves from state s to state s'. A higher rate means a faster transition.

A Continuous-Time Markov Chain (CTMC) is a type of Markov model where the time between transitions is not fixed but is governed by exponential distributions. If there are more then one outgoing transition from a state, we get race-conditions, the first transition to occur is the one that will be taken. The time spent in a state before transitioning to a new state is called $dwell - time$. This is exponentially distributed with a rate $E(s) = \sum_{s' \in S} R(s, s')$. The probability of transitioning from state s to state s' is $R(s, s')/E(s)$, the time spent in s is independent from the properbility of transitioning to s'.

4.2 Matrix Representation

The transition function τ can be represented as a matrix, where each element $\tau(s, s')$ is the probability of transitioning from state s to state s'. The matrix representation of τ is called the transition matrix. The transition matrix is a square matrix with dimensions $|S| \times |S|$, where $|S|$ is the number of states in the model. The transition matrix is a stochastic matrix, meaning that the sum of each row is equal to 1, meaning all the probabilities of transitioning from state s to all other states sum to 1.

If we take an example of a model with two states $S = \{s_1, s_2\}$, the transition matrix τ is defined as:

$$\tau = \begin{bmatrix} \tau(s_1, s_1) & \tau(s_1, s_2) \\ \tau(s_2, s_1) & \tau(s_2, s_2) \end{bmatrix} \tag{32}$$

We can give an example of a transition matrix for a model with two states, where the model transitions from state s_1 to state s_2 with probability 0.4 and transitions from state s_2 to state s_1 with probability 0.5:

$$\tau = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix} \tag{33}$$

The initial distribution π is a vector that represents the probability of starting in each state. The initial distribution is a stochastic vector, meaning that the sum of all probabilities is equal to 1. The initial distribution π is a vector with dimensions

$|S|$, where $|S|$ is the number of states in the model. Each element $\pi(s)$ is the probability of starting in state s.

$$\pi = \begin{bmatrix} 0.6 \\ 0.5 \end{bmatrix} \tag{34}$$

The labeling function ℓ can be represented as a matrix, where each element $\ell(s, l)$ is the probability of emitting label l in state s. The matrix representation of ℓ is called the emission matrix. The emission matrix is a matrix with dimensions $|S| \times |\mathcal{L}|$, where $|\mathcal{L}|$ is the number of labels in the model. The emission matrix is a stochastic matrix, meaning that the sum of each row is equal to 1, meaning all the probabilities of emitting a label in state s sum to 1.

If we take an example of a model with two states $S = \{s_1, s_2\}$ and two labels $\mathcal{L} = \{l_1, l_2\}$, the emission matrix ℓ is defined as:

$$\ell = \begin{bmatrix} \ell(s_1, l_1) & \ell(s_1, l_2) \\ \ell(s_2, l_1) & \ell(s_2, l_2) \end{bmatrix} \tag{35}$$

We can give an example of an emission matrix for a model with two states and two labels, where the model emits label l_1 in state s_1 with probability 0.7 and emits label l_2 in state s_2 with probability 0.6:

$$\ell = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \tag{36}$$

The rate function R can be represented as a matrix, where each element $R(s, s')$ is the rate of transitioning from state s to state s'. The matrix representation of R is called the rate matrix. The rate matrix is a square matrix with dimensions $|S| \times |S|$, where $|S|$ is the number of states in the model. The rate matrix is a non-negative matrix, meaning that all elements are greater than or equal to 0.

$$R = \begin{bmatrix} R(s_1, s_1) & R(s_1, s_2) \\ R(s_2, s_1) & R(s_2, s_2) \end{bmatrix} \tag{37}$$

If we take an example of a model with two states $S = \{s_1, s_2\}$, the rate matrix R is defined as:

$$R = \begin{bmatrix} 0.5 & 0.3 \\ 0.2 & 0.4 \end{bmatrix} \tag{38}$$

5 HMM Example

5.1 Setup

We have a simple HMM with, two hidden states S_1 and S_2, two observation symbols: O_1 and O_2 and an observation sequence $O = \{O_1, O_2, O_1\}$.

The HMM parameters are:

Transition matrix A (probability of moving from one state to another):

$$A = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix}$$

Emission matrix B (probability of emitting observation given a state):

$$B = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}$$

Initial state probability vector π (probability of starting in each state):

$$\pi = \begin{bmatrix} 0.8 & 0.2 \end{bmatrix}$$

5.2 Expectation step

In the expectation step we calculate α and β.

5.2.1 Forward step α

We first compute the forward probabilities $\alpha_t(i)$, which represent the probability of being in state i at time t after observing the first t symbols.

5.2.1.1 Initialization at $(t = 1)$:

$$\alpha_1 = \pi \circ B_{y1}$$

Where B_{y1} is the first column of the emission matrix, corresponding to observation O_1

(i.e., $B_{y1} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}$) and $\circ$ represents the Hadamard product. So, we get:

$$\alpha_1 = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \circ \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.56 \\ 0.08 \end{bmatrix}$$

5.2.1.2 Induction (for $t = 2, 3, \ldots, T$): For subsequent timesteps, we compute:

$$\alpha_{t+1} = B_{yt+1} \circ (A^T \alpha_t)$$

Where A^T is the transpose of the transition matrix. Let's apply this to compute the forward probabilities for $t = 2$ and $t = 3$:

At $t = 2$ (observation O_2):

$$\alpha_2 = B_{y2} \circ (A^T \alpha_1)$$

We have:

$$B(y2) = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix}$$

and

$$A^T = \begin{bmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{bmatrix}$$

We get:

$$\alpha_2 = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} \circ \left(\begin{bmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.56 \\ 0.08 \end{bmatrix} \right) = \begin{bmatrix} 0.1128 \\ 0.1584 \end{bmatrix}$$

At $t = 3$ (observation O_1):

$$\alpha_2 = B_{y1} \circ (A^T \alpha_2)$$

We get:

$$\alpha_3 = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} \circ \left(\begin{bmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.1 \\ 0.1584 \end{bmatrix} \right) = \begin{bmatrix} 0.102816 \\ 0.049728 \end{bmatrix}$$

5.2.2 Backward step β

The backward probabilities $\beta_t(i)$ represent the probability of observing the rest of the sequence starting from time $t+1$, given that the system is in state i at time t.

Initialization (at $t = T = 3$)

$$\beta_T = \mathbf{1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

5.2.2.1 Induction (for $t = T-1, T-2, \dots, 1$): For earlier timesteps, we compute:

$$\beta_t = A(\beta_{t+1} \circ B_{yt+1})$$

At $t = 2$ (observation O_1):

$$\beta_2 = A(\beta_3 \circ B_{y1})$$

$$B_{y1} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}, \quad \beta_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

We get:

$$\beta_2 = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix} \cdot \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} \right)$$

$$\beta_2 = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix}$$

At $t = 1$ (observation O_2):

$$\beta_1 = A(\beta_2 \circ B_{y2})$$

We have:

$$B_{y2} = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix}, \quad \beta_2 = \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix}$$

$$\beta_1 = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix} \cdot \left(\begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} \circ \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix} \right)$$

$$\beta_1 = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.174 \\ 0.33 \end{bmatrix} = \begin{bmatrix} 0.2364 \\ 0.252 \end{bmatrix}$$

5.3 Step 3: Compute γ and ξ

5.3.1 Compute γ

We can compute γ by

$$\gamma_t = (\mathbb{1}^T \cdot \alpha_T)^{-1} \cdot (\alpha_t \circ \beta_t)$$

$$\alpha_T = \begin{bmatrix} 0.089628 \\ 0.053328 \end{bmatrix}$$

$$\mathbb{1}^T \cdot \alpha_T = 0.089628 + 0.053328 = 0.152544$$

This is the total probability of oberserving our sequence $O = \{O_1, O_2, O_1\}$

Now we can compute γ_t for each time stamp.

At t=1: We have

$$\alpha_1 = \begin{bmatrix} 0.56 \\ 0.08 \end{bmatrix}, \quad \beta_1 = \begin{bmatrix} 0.2364 \\ 0.252 \end{bmatrix}$$

We take the Hadamard product of this.

$$\alpha_1 \circ \beta_1 = \begin{bmatrix} 0.56 \cdot 0.2364 \\ 0.08 \cdot 0.252 \end{bmatrix} = \begin{bmatrix} 0.132384 \\ 0.02016 \end{bmatrix}$$

We normalize the first part and take the scalar product.

$$\gamma_1 = \frac{1}{0.152544} \cdot \begin{bmatrix} 0.132384 \\ 0.02016 \end{bmatrix} = \begin{bmatrix} 0.8678414 \\ 0.1321589 \end{bmatrix}$$

At t = 2:

We have:

$$\alpha_2 = \begin{bmatrix} 0.1074 \\ 0.1584 \end{bmatrix}, \quad \beta_2 = \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix}$$

The Hadamard product is:

$$\alpha_2 \circ \beta_2 = \begin{bmatrix} 0.1074 \cdot 0.58 \\ 0.1584 \cdot 0.55 \end{bmatrix} = \begin{bmatrix} 0.062292 \\ 0.08712 \end{bmatrix}$$

We normalize the first part and take the scalar product.

$$\gamma_2 = \frac{1}{0.152544} \cdot \begin{bmatrix} 0.062292 \\ 0.08712 \end{bmatrix} = \begin{bmatrix} 0.42888609 \\ 0.57111391 \end{bmatrix}$$

At t = 3

We have:

$$\alpha_3 = \begin{bmatrix} 0.089628 \\ 0.053328 \end{bmatrix}, \quad \beta_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The Hadamard product is:

$$\alpha_3 \circ \beta_3 = \begin{bmatrix} 0.089628 \\ 0.053328 \end{bmatrix}$$

We normalize the first part and take the scalar product.

$$\gamma_3 = \frac{1}{0.152544} \cdot \begin{bmatrix} 0.089628 \\ 0.053328 \end{bmatrix} = \begin{bmatrix} 0.67400881 \\ 0.32599119 \end{bmatrix}$$

5.3.2 Calculating ξ

We calculate ξ by

$$\xi_t = ((\mathbb{1}^T \alpha_T)^{-1} \cdot A) \circ (\alpha_t \otimes (\beta_{t+1} \circ B_{yt+1})^T)$$

We start by calculating $((\mathbb{1}^T \alpha_T)^{-1} \cdot A)$: From before, we have

$$(\mathbb{1}^T \alpha_T)^{-1} = \frac{1}{0.152544} = 6.996$$

We have:

$$A = \begin{bmatrix} 0.6 & 0.4 \\ 0.5 & 0.5 \end{bmatrix}$$

We get:

$$8.996 \cdot A = \begin{bmatrix} 6.996 \cdot 0.6 & 6.996 \cdot 0.4 \\ 6.996 \cdot 0.5 & 6.996 \cdot 0.5 \end{bmatrix} = \begin{bmatrix} 4.198 & 2.798 \\ 3.498 & 3.498 \end{bmatrix}$$

We can now calculate $\alpha_1 \otimes (\beta_2 \circ B_{y2})^T$. We have :

$$\alpha_1 = \begin{bmatrix} 0.56 \\ 0.08 \end{bmatrix}, \quad \beta_2 = \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix}, \quad B_{y2} = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix}$$

We calculate $\beta_2 \circ B_{y2}$:

$$\beta_2 \circ B_{y2} = \begin{bmatrix} 0.58 \\ 0.55 \end{bmatrix} \circ \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 0.174 \\ 0.33 \end{bmatrix}$$

Outer product:

$$\alpha_1 \otimes (\beta_2 \circ B_{y2})^T = \begin{bmatrix} 0.56 \\ 0.08 \end{bmatrix} \otimes \begin{bmatrix} 0.174 & 0.33 \end{bmatrix}$$

$$= \begin{bmatrix} 0.09744 & 0.1848 \\ 0.01392 & 0.0264 \end{bmatrix}$$

We can now calculate ξ_1

$$\xi_1 = \begin{bmatrix} 4.198 & 2.798 \\ 3.498 & 3.498 \end{bmatrix} \circ \begin{bmatrix} 0.09744 & 0.1848 \\ 0.01392 & 0.0264 \end{bmatrix}$$

$$\xi_1 = \begin{bmatrix} 0.38325991 & 0.03650094 \\ 0.60572687 & 0.08653241 \end{bmatrix}$$

At t=2:
We have:

$$B_{y1} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}, \quad \alpha_2 = \begin{bmatrix} 0.1074 \\ 0.1584 \end{bmatrix}, \quad \beta_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Hadamard product for $\beta_3 \circ B_{y1}$

$$\beta_3 \circ B_{y1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}$$

Outer product:

$$\alpha_2 \otimes \begin{bmatrix} 0.7 & 0.4 \end{bmatrix} = \begin{bmatrix} 0.07518 & 0.04296 \\ 0.11088 & 0.06336 \end{bmatrix}$$

We can now calculate ξ_2:

$$\xi_2 = \begin{bmatrix} 4.198 & 2.798 \\ 3.498 & 3.498 \end{bmatrix} \circ \begin{bmatrix} 0.07518 & 0.04296 \\ 0.11088 & 0.06336 \end{bmatrix}$$

$$\xi_2 = \begin{bmatrix} 0.07341938 & 0.06873304 \\ 0.03726872 & 0.0523348 \end{bmatrix}$$

At t=3:
We have:

$$B_{y1} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}, \quad \alpha_3 = \begin{bmatrix} 0.089628 \\ 0.053328 \end{bmatrix}, \quad \beta_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Hadamard product for $\beta_3 \circ B_{y1}$

$$\beta_3 \circ B_{y1} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \circ \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 0.7 \\ 0.4 \end{bmatrix}$$

Outer product:

$$\alpha_3 \otimes \begin{bmatrix} 0.7 & 0.4 \end{bmatrix} = \begin{bmatrix} 0.062740 & 0.035852 \\ 0.037329 & 0.021331 \end{bmatrix}$$

We can now calculate ξ_3:

$$\xi_2 = \begin{bmatrix} 4.198 & 2.798 \\ 3.498 & 3.498 \end{bmatrix} \circ \begin{bmatrix} 0.062740 & 0.035852 \\ 0.037329 & 0.021331 \end{bmatrix}$$

$$\xi_3 = \begin{bmatrix} 0.2839837 & 0.09127753 \\ 0.13480176 & 0.06519824 \end{bmatrix}$$

5.4 Update values

$$\hat{\pi} = \gamma_1 = \begin{bmatrix} 0.86784141 \\ 0.1321589 \end{bmatrix}$$

$$\hat{A} = (\mathbb{1} \oslash \gamma) \bullet \xi$$

$$\hat{B} = (\mathbb{1} \oslash \gamma) \bullet (\sum_{t=1}^{T} \gamma_t \otimes \mathbb{1}_{yt}^{T})$$

When referring to γ, we use the sum of the probabilities:

$$\gamma = \sum_{t=1}^{T} \gamma_t$$

and ξ:

$$\xi = \sum_{t=1}^{T} \xi_t$$

We therefore calculate:

$$\gamma = \begin{bmatrix} 0.86784141 \\ 0.1321589 \end{bmatrix} + \begin{bmatrix} 0.42888609 \\ 0.57111391 \end{bmatrix} + \begin{bmatrix} 0.67400881 \\ 0.32599119 \end{bmatrix} = \begin{bmatrix} 1.97073631 \\ 1.02926369 \end{bmatrix}$$

And

$$\xi = \begin{bmatrix} 0.38325991 & 0.03650094 \\ 0.60572687 & 0.08653241 \end{bmatrix} + \begin{bmatrix} 0.07341938 & 0.06873304 \\ 0.03726872 & 0.0523348 \end{bmatrix}$$

$$+ \begin{bmatrix} 0.2830837 & 0.09127753 \\ 0.13480176 & 0.06519824 \end{bmatrix} = \begin{bmatrix} 0.739763 & 0.19651152 \\ 0.77779736 & 0.20406545 \end{bmatrix}$$

We can now calculate

$$\mathbb{1} \oslash \gamma = \begin{bmatrix} \frac{1}{2.0923} \\ \frac{1}{1.1352} \end{bmatrix}$$

We can now calculate $\hat{A}$

$$\hat{A} = \begin{bmatrix} \frac{1}{2.0923} \\ \frac{1}{1.1352} \end{bmatrix} \bullet \begin{bmatrix} 0.9897 & 0.7370 \\ 0.5670 & 0.3888 \end{bmatrix} = \begin{bmatrix} 0.37537391 & 0.19092437 \\ 0.39467348 & 0.198226353 \end{bmatrix}$$

We calculate $\hat{B}$ We first calculate the sum of the outer products:

$$\sum_{t=1}^{T} \gamma_t \otimes \mathbb{1}_{yt}^{T}$$

At t = 1:

$$\gamma_1 \otimes \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 0.86784141 \\ 0.1321589 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 0.86784141 & 0.13215859 \\ 0 & 0 \end{bmatrix}$$

At t = 2:

$$\gamma_2 \otimes \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.42888609 \\ 0.57111391 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0.42888609 & 0.57111391 \end{bmatrix}$$

At t = 3:

$$\gamma_3 \otimes \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 0.67400881 \\ 0.32599119 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 0.67400881 & 0.32599119 \\ 0 & 0 \end{bmatrix}$$

We summearize these to get:

$$\begin{bmatrix} 0.86784141 & 0.13215859 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0.42888609 & 0.57111391 \end{bmatrix} +$$

$$\begin{bmatrix} 0.67400881 & 0.32599119 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 1.54185022 & 0.45814978 \\ 0.42888609 & 0.57111391 \end{bmatrix}$$

$$\hat{b} = \begin{bmatrix} \frac{1}{2.0923} \\ \frac{1}{1.1352} \end{bmatrix} \bullet \begin{bmatrix} 1.54185022 & 0.45814978 \\ 0.42888609 & 0.57111391 \end{bmatrix}$$

$$= \begin{bmatrix} 0.78237266 & 0.23247645 \\ 0.41669214 & 0.55487618 \end{bmatrix}$$

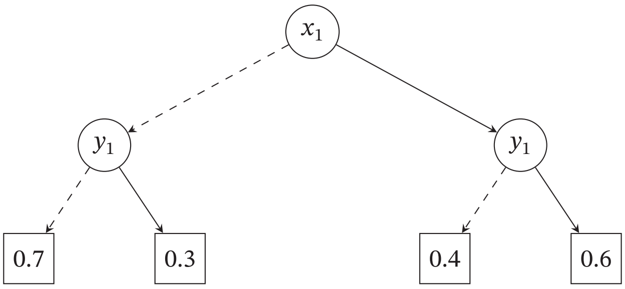

Fig. 12. B-matrix representation in ADD

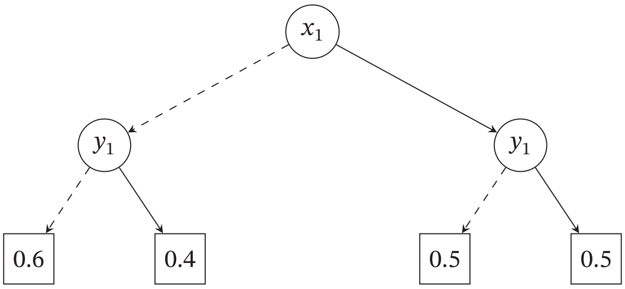

Fig. 13. A-matrix representation in ADD

5.5 ADD representation

As we only need one bit to represent the the rows and columns with one bit, we only need one variable for the them, as x_1 is the variable for rows and y_1 is the variable for column.

We first make the matrices into ADD representation.

We can now use the ADD representation to calculate α and β.

When using ADD's it is important to remember, if we need to take a row from a matrix, we fix the input to the ADD by setting the x-variables to the desired row. An example is taking the third row of a matrix with 8 rows, we set, $x_1 = 1, x_2 = 1, x_3 = 0$ and $x_4 = 0$. if we need to take the second column, we set $y_1 = 1, y_2 = 0$ and $y_3 = 0, y_4 = 0$. Hadamard product is row-wise multiplication of the matrices. So to calculate the Hadamard product of two matrices, we set the x-variables to the same row in both matrices and multiply the corresponding nodes in the ADDs. To calculate a Hadamard product in ADD, we multiply the corresponding nodes in the ADDs, as shown in the following figure.

Matrix multiplication is done by fixing the input to the first matrix and the output to the second matrix. We then sum the result of the Hadamard product of the rows of the first matrix and the columns of the second matrix. This is shown in the following figure.

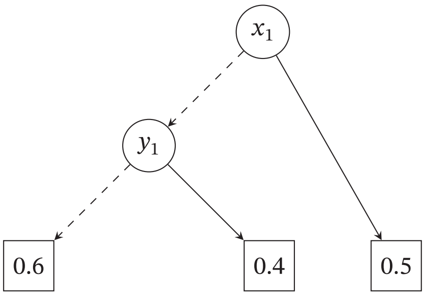

Fig. 14. A-matrix representation in ADD caption reduced

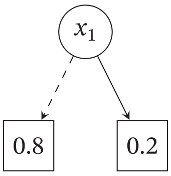

Fig. 15. π-matrix representation in ADD

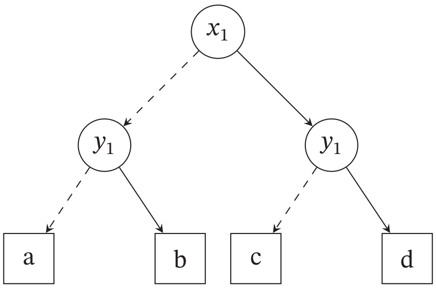

Fig. 16. Matrix A in ADD

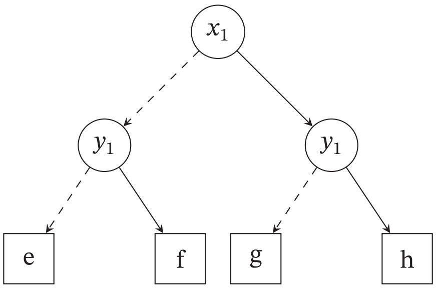

Fig. 17. Matrix B in ADD

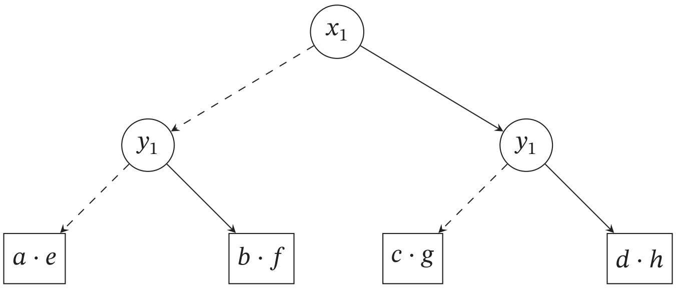

Fig. 18. Hadamard product of A and B in ADD

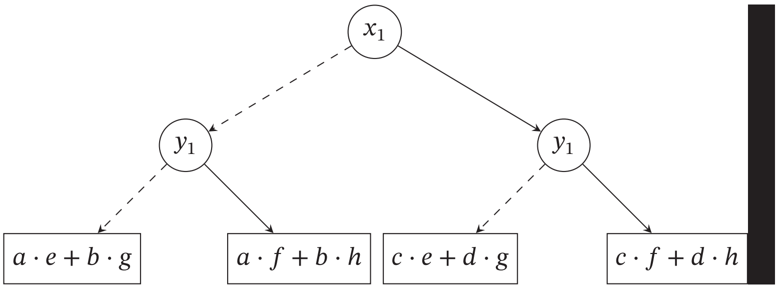

Fig. 19. Matrix multiplication of A and B in ADD

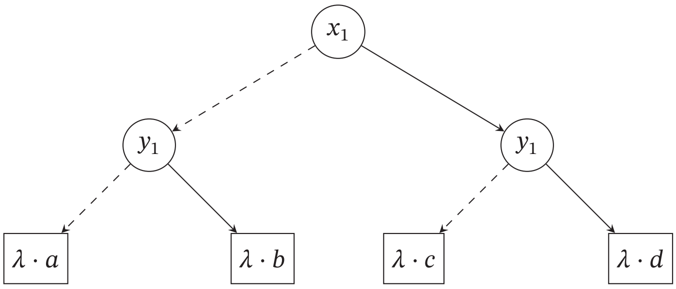

Fig. 20. Scalar product in ADD

6 Class Diagram

The class diagram in Figure 25 shows the relationships between the different classes in the system. The `Model` class represents the underlying model of the system, which can be a CTMC, DTMC, HMM and MDP. The `Model` class has a aggregation relationship with the `Algorithm` class, which represents the functions used in the Baum-Welch algorithm, with and without ussing log-semiring. The `Algorithm` class has a dependency relationship with the `CUDD` class, which is a wrapper for the CUDD library. The `CUDD` class is used to perform the matrix operations as ADD's required by the Baum-Welch algorithm. The `Model` class also has a aggregation relationship with the `CUDD` class, as the `Model` class uses the `CUDD` class to perform the ADD operations.

6.1 `Model` Class

The `Model` class serves as the foundation for representing various probabilistic models like CTMC, MDP, and DTMC. It holds fields needed to describe these models, such as the transition matrices, emission probabilities, and initial states, all represented using Algebraic Decision Diagrams (ADDs). The `Model` class also provides methods for training models, such as the Baum-Welch algorithm.

Attributes:

- `Type_model`: Defines the type of model (e.g., CTMC, MDP, DTMC).
- `transfer`: A list of ADD structures representing state transition probabilities.
- `Emission`: An ADD for the emission probabilities (relevant in Hidden Markov Models).
- `pi`: The initial state distribution, also stored as an ADD.
- `training_set`: A collection of observed data.

Methods:

- `Instantiate_with_parameters(prismfile, parameters: Dictionary)`: Instantiates a model with specified parameters.
- `Instantiate_without_parameters(prismfile)` Creates a model without additional parameters.
- `Baum-welsh(log, Model)`: This method implements the Baum-Welch algorithm for training Hidden Markov Models, utilizing various operations from the `Algorithm` class.

6.2 `CUDD` Class

The `CUDD` class (CUDD Manager) is responsible for managing ADDs. These ADDs are crucial in representing the probabilistic data structures used in the `Model` class. `CUDD` provides a set of operations that allow mathematical and logical manipulation of these diagrams.

Attributes:

- `rowvars, colvars`: Representing variables used in the ADD structures.
- `ADD`: The main data structure for storing probabilities or logical expressions.
- `Manager`: A control structure that coordinates operations on ADDs.

Methods:

- `Hadamard(), Log_Hadamard()`: Perform element-wise operations on ADDs.
- `Matrix_mul(), Log_matrix_mul()`: For matrix multiplications.
- `Sum(), Transpose()`: Additional helper methods for summing and transposing ADDs.

6.3 `Algorithm` Class

The `Algorithm` class encapsulates various methods for performing probabilistic calculations. These methods are mainly used for inference in models such as Hidden Markov Models (HMM) and Markov Chains.

Methods:

- `calculate_alpha(), calculate_beta()`: Compute the forward (alpha) and backward (beta) probabilities, respectively.
- `calculate_gamma(), calculate_xi()`: Intermediate probability calculations needed for parameter estimation and model training.

Each method operates on the ADD structures created and managed by the `CUDD` class, ensuring efficient computation of the probabilities.

6.4 Relationships Between Classes

6.4.1 `Model` to `Algorithm`: Association Relationship

The `Model` class uses the `Algorithm` class to compute the forward-backward probabilities and other values necessary for inference. The Baum-welsh(log, Model) method in `Model` invokes the relevant methods from `Algorithm` (calculate_alpha(), calculate_beta(), etc.) during the training process of HMMs. These methods, while called collectively in Baum-Welch, can also be used independently to perform specific calculations.

[redacted]4.2 `Model` to `CUDD`: Aggregation Relationship

The `Model` class contains several attributes (transfer, Emission, pi) that are represented as ADDs, managed by the `CUDD` class. This relationship is best represented as an aggregation, where the `Model` holds instances of ADD but does not directly manage their internal workings. Instead, `CUDD` provides the operations required to manipulate and operate on these diagrams, such as matrix multiplication or element-wise functions (Hadamard products). The `Model` depends on `CUDD` for these operations, making it an integral part of the system's backend.

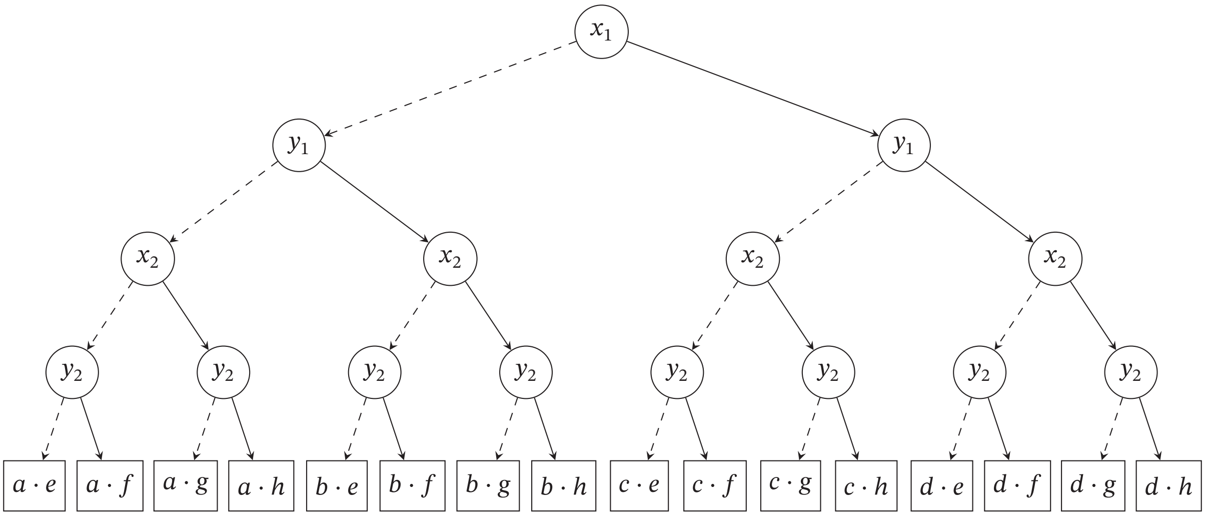

Fig. 21. Kroneker product in ADD

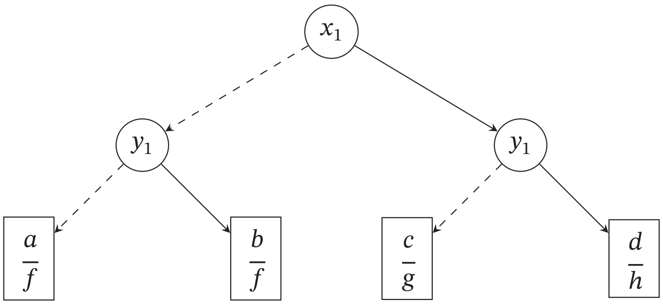

Fig. 22. Hadamard division of A and B in ADD

6.4.3 `Algorithm` to `CUDD`: Dependency Relationship

The `Algorithm` class depends on the `CUDD` class for all its operations on ADDs. Every method in `Algorithm` (e.g., calculate_alpha(), calculate_gamma()) relies on ADD operations provided by `CUDD`, such as Matrix_mul() and Hadamard(). This is represented by a dependency relationship, where `Algorithm` calls `CUDD`'s methods to perform its computations.

7 Experiment

The purpose of the experiments is to evaluate the performance of different implementations of the Baum-Welch algorithm when applied to various CTMC models. Specifically, we compare our implementation that utilize ADDs with and without the log semiring to other implementations from Jajapy, namely the original Jajapy implementation of the Baum-Welch algorithm and the improved SUDD implementation. By analyzing both parameter estimation accuracy and runtime, we aim to determine the efficiency and reliability of each implementation across multiple model types. We also investigate the scalability of the Baum-Welch algorithm by increasing the number of states in the model and comparing the runtime of each implementation.

We use 5 different models to test the performance of the Baum-Welch algorithm. Each model provides a different structure and complexity level for testing the Baum-Welch algorithm. The models are derived from CTMCs and include:

- **Polling**: A model representing a server polling multiple queues, often used in network communication scenarios.
- **Cluster**: Simulates clusters of entities competing for shared resources.
- **Tandem**: A series of interconnected queues where entities move sequentially from one queue to the next.
- **Philosophers(I) and Philosophers(II)**: Models representing the classic synchronization problem where entities (philosophers) compete for limited resources (e.g., forks), with Philosophers2 containing one additional parameter.

In every model, we have a number of parameters we want to estimate. The number of parameters to estimate in each model is shown in Table 2. The polling model is the smallest model with 2 parameters to estimate, while philosophers(II) is the largest model, with 4 parameters to estimate.'

TABLE 2
Number of parameters to estimate in each model

Model	Num of parameters
Polling	2
Cluster	3
Tandem	3
Philosophers(I)	3
Philosophers(II)	4

7.1 Parameter Estimation Accuracy

The first experiment measures the time and accuracy of the Baum-Welch algorithm for each implementation. We generate observation sequences from each model and use the Baum-Welch algorithm to estimate the parameters of the model. We compare the estimated parameters with the true parameters of the model to evaluate the accuracy of the estimation.

The experiment is made with the following steps:

1) We load the model.
2) Generate an observation sequence from the model with untimed steps.

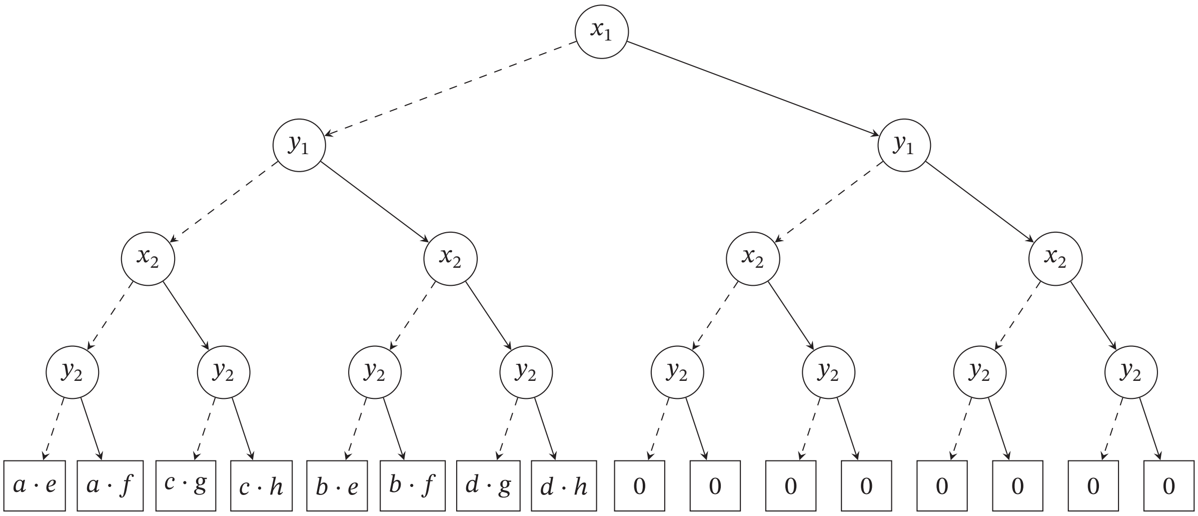

Fig. 23. Katri-Rao in ADD

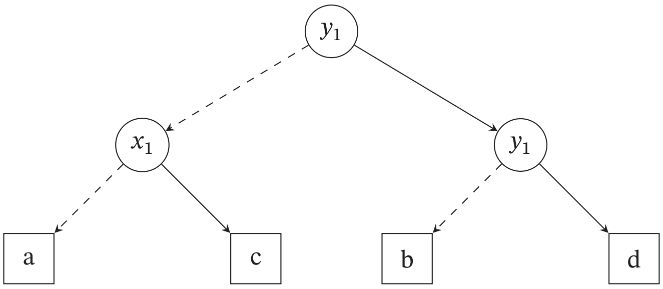

Fig. 24. transpose in ADD

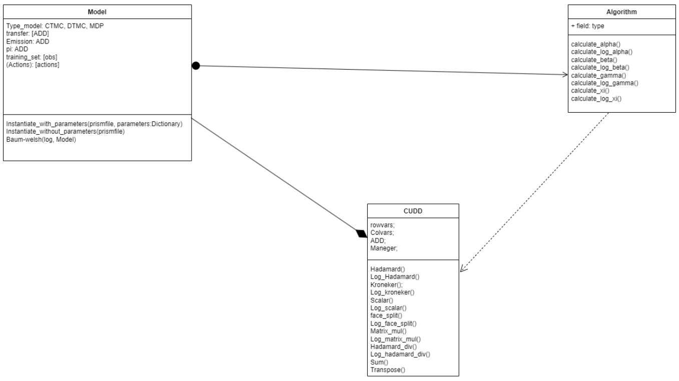

Fig. 25. Class diagram of the system

3) Calculate the model's parameters using each variant of the Baum-Welch algorithm, recording both the parameter estimates and runtime.
4) Repeat steps 2 and 3 ten times without changing the model, the sequence length, or whether the sequence is timed or untimed. The results are averaged to account for any variance in runtime and estimation accuracy.
5) Repeat with timed steps to observe the effects of timing information on estimation accuracy and runtime.
6) Compare the estimated parameters with the true parameters and record the runtime for each implementation.
7) Move to the next model and repeat the entire process.

Following the experiment, the estimated parameters for each implementation are compared with the true parameters for accuracy assessment. Additionally, runtime for each Baum-Welch implementation is recorded. These results are presented in tables and plots to facilitate a direct comparison of performance across models and between implementations.

Table 3, 4, 5, 6 and 7 show detailed results for each model in terms of runtime and estimation accuracy (relative formula error and relave parameter error).

7.2 Scalability experiment

We also test the scalability of the Baum-Welch algorithm by increasing the number of states in a model. We use the Tandem model and increase the number of states from 28 to 1225. We then compare the runtime of the Baum-Welch algorithm for each implementation. We run the experiment 10 times for each number of states and compare the runtime of the Baum-Welch algorithm for each implementation.

We use plots to illustrate the scalability by showing runtime across an increasing number of states for the Tandem model, highlighting each implementation's computational efficiency.

7.3 Results

The comparison of the Baum-Welch implementations is based on the following criteria:

- **Parameter estimation accuracy**: The accuracy of the estimated parameters compared to the true parameters.
- **Runtime**: The time it takes to estimate the parameters.
- **Scalability**: How the runtime scales with the number of states in the model.

The results are displayed as tables and plots to facilitate a direct comparison of performance across models and between implementations.

Acronyms

AAU	Aalborg University. 1
ADD	Algebraic Decision Diagram. 2, 6–8

TABLE 3
Comparison of Baum-Welch implementations for Polling

Implementation	Timed Observations				Untimed Observations			
	Time(s)	Iteration	avg δ	avg ϕ	Time(s)	Iteration	avg δ	avg ϕ
CuPAAL	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
CuPAAL_log	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
SUDD	23.5	3.5	0.1	0.1	23.5	3.5	0.1	0.1
SUDD_log	0.5	3.5	0.1	0.1	0.5	3.5	0.1	0.1
Jajapy	9.5	3.5	0.1	0.1	9.5	3.5	0.1	0.1

TABLE 4
Comparison of Baum-Welch implementations for Cluster

Implementation	Timed Observations				Untimed Observations			
	Time(s)	Iteration	avg δ	avg ϕ	Time(s)	Iteration	avg δ	avg ϕ
CuPAAL	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
CuPAAL_log	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
SUDD	23.5	3.5	0.1	0.1	23.5	3.5	0.1	0.1
SUDD_log	0.5	3.5	0.1	0.1	0.5	3.5	0.1	0.1
Jajapy	9.5	3.5	0.1	0.1	9.5	3.5	0.1	0.1

TABLE 5
Comparison of Baum-Welch implementations for Tandem

Implementation	Timed Observations				Untimed Observations			
	Time(s)	Iteration	avg δ	avg ϕ	Time(s)	Iteration	avg δ	avg ϕ
CuPAAL	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
CuPAAL_log	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
SUDD	23.5	3.5	0.1	0.1	23.5	3.5	0.1	0.1
SUDD_log	0.5	3.5	0.1	0.1	0.5	3.5	0.1	0.1
Jajapy	9.5	3.5	0.1	0.1	9.5	3.5	0.1	0.1

TABLE 6
Comparison of Baum-Welch implementations for Philosophers(I)

Implementation	Timed Observations				Untimed Observations			
	Time(s)	Iteration	avg δ	avg ϕ	Time(s)	Iteration	avg δ	avg ϕ
CuPAAL	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
CuPAAL_log	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
SUDD	23.5	3.5	0.1	0.1	23.5	3.5	0.1	0.1
SUDD_log	0.5	3.5	0.1	0.1	0.5	3.5	0.1	0.1
Jajapy	9.5	3.5	0.1	0.1	9.5	3.5	0.1	0.1

TABLE 7
Comparison of Baum-Welch implementations for Philosophers(II)

Implementation	Timed Observations				Untimed Observations			
	Time(s)	Iteration	avg δ	avg ϕ	Time(s)	Iteration	avg δ	avg ϕ
CuPAAL	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
CuPAAL_log	132.5	3.5	0.1	0.1	132.5	3.5	0.1	0.1
SUDD	23.5	3.5	0.1	0.1	23.5	3.5	0.1	0.1
SUDD_log	0.5	3.5	0.1	0.1	0.5	3.5	0.1	0.1
Jajapy	9.5	3.5	0.1	0.1	9.5	3.5	0.1	0.1

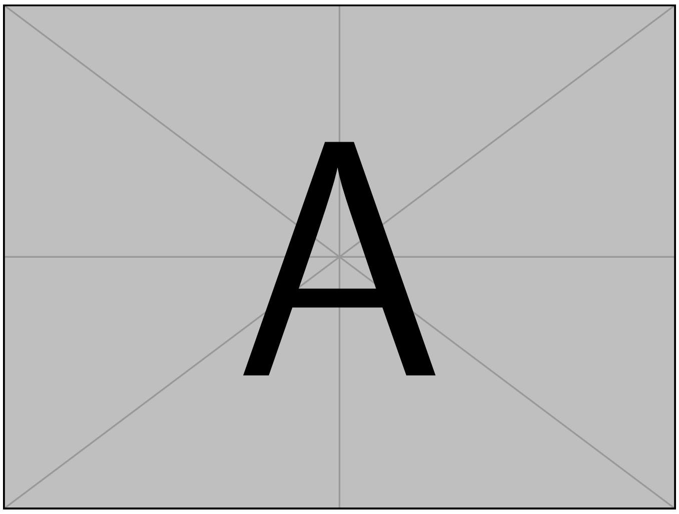

Fig. 26. Scalability of Baum-Welch implementations for the Tandem model

BDD	Binary Decision Diagram. 6
CTMC	Continuous Time Markov Chain. 2–4
DTMC	Discrete Time Markov Chain. 2
pCTMC	Parametric Continuous Time Markov Chain. 2–4

References

[1] A. A. Markov and J. J. Schorr-Kon, *Theory of algorithms*. Springer, 1962.

[2] P. Milazzo, “Analysis of covid-19 data with prism: Parameter estimation and sir modelling,” in *From Data to Models and Back*, Springer International Publishing, 2021, pp. 123–133, ISBN: 978-3-030-70650-0.

[3] 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.

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

[5] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[6] E. M. Clarke, “Model checking,” in *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, Springer, 1997, pp. 54–56.

[7] 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.

[8] 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,

[9] 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]`.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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. eprint: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/a7me0f/alma9921651457905762.

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

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.