Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Minimisation of Multiplicity Tree Automata

Bachelorarbeit

Leipzig, Dezember, 2023

vorgelegt von

Krümpelmann, Hanno
Studiengang Informatik B. Sc.

Betreuender Hochschullehrer: Prof. Dr. Andreas Maletti
Algebraische und logische Grundlagen der Informatik

Contents

0.1 Introduction

This thesis is concerned with the concept of Multiplicity Tree Automata (MTA). These were first introduced by Berstel and Reutenauer [1] in their paper 'Recognizable formal power series on trees' under the terminology of tree series. I specifically use a definition of MTAs based on the paper 'Minimisation of multiplicity tree automata' by Kiefer et al. [2] and write about the minimisation algorithm which is specified in the aforementioned paper.
The results of this thesis are three-fold: First, the concept of MTAs is illustrated extensively by a number of examples, allowing easier access to a complicated theoretical construct. Second, the minimisation algorithm is made explicit, completely given in pseudo code, and accompanied by an example run, while in the original paper [2] only one part of the algorithm was made explicit, with the rest given in more abstract terms. Third, MTAs in general and the minimisation algorithm in particular are implemented from scratch in Rust and attached to the thesis. Instructions as to how the code can be executed are provided and with suitable examples the correctness of the code is illustrated.

Chapter 1

Preliminaries

1.1 Structure of the Thesis

The thesis begins by laying the groundwork with basic definitions in the first chapter. Some foundations in linear algebra and automata theory are needed which might not have been part of the basics covered in a foundational curriculum geared to computer scientists. This section contains formal definitions which will be needed both for defining MTAs and developing the minimisation algorithm. This introduction begins by defining those concepts followed by a few select examples to illustrate the various constructs.
The second chapter of the thesis introduces the MTA as a theoretical construct. Alongside, three illustrative examples are described which are used throughout the whole thesis.
With this foundation of the underlying theoretical concepts, the third chapter begins by briefly considering some existing implementations of tree automata, and then introduces the attached implementation.
In the following chapter 4, the main result from [2], the efficient minimisation of those automatons is presented concretely. The algorithm will be derived from some theoretical results from the aforementioned paper, it will be made explicit in the form of pseudo code alongside an illustrative example and some additional explanations.
Finally, in chapter 5, the implementation of the minimisation algorithm will be presented. This is done by executing the minimisation algorithm on one of the examples from the end of chapter 2, and the weights of two example trees are calculated to illustrate that it does indeed minimise the automaton correctly.

1.2 Definitions

This thesis closely follows the aforementioned paper [2, Minimisation of Multiplicity Tree Automata] from Kiefer et al.. The definitions used correspond mostly to those in this paper. The same is true for the proofs, which are all either attributed to the relevant section in [3] or to other papers when appropriate.

1.2.1 Basic Notation

Let $\mathbb{N}$ denote the natural numbers without zero and $\mathbb{N}_0$ the natural numbers with zero.
Let A be a matrix with n rows and m columns. A_i denotes the i^{th} row, A^j the j^{th} column and $A_{i,j}$ denotes its $(i,j)^{th}$ entry. Let I_n be the identity matrix of order n. Given a field $\mathbb{F}$ and a set $S \subseteq \mathbb{F}$, let $\langle S \rangle$ denote the vector subspace of $\mathbb{F}^n$ that is spanned by S.
The set $\{1, 2, ..., n\}$ is written as $[n]$.

1.2.2 Alphabet

Let Σ be a finite alphabet, ϵ be the empty word and Σ^* denote the set of all words over Σ. The length of a word $w \in \Sigma^*$ is denoted by $|w|$.
The following notations for any $n \in \mathbb{N}_0$ will be used:

- $\Sigma^n := \{w \in \Sigma^* : |w| = n\}$ - words of length n
- $\Sigma^{\leq n} := \bigcup_{l=0}^{n} \Sigma^l$ - words shorter or equal to n
- $\Sigma^{<n} := \Sigma^{\leq n} \backslash \Sigma^n$ - words shorter than n

Given two words $x, y \in \Sigma^*$, let xy denote the concatenation of x and y.
A ranked alphabet is a tuple (Σ, rk) where Σ is a non-empty finite set and rk is a mapping $rk : \Sigma \to \mathbb{N}_0$. ($\Sigma$, rk) will be generally shortened to Σ.
Let $\Sigma_k := rk^{-1}(\{k\})$ denote the set of all $k-ary$ symbols for every $k \in \mathbb{N}_0$. Σ has rank r if $r = max\{rk(\sigma) : \sigma \in \Sigma\}$.

1.2.3 Kronecker Product

Definition 1.1 *Let A be an $[m_1 \times n_1]$ matrix and B an $[m_2 \times n_2]$ matrix. The Kronecker product of A by B, written as $A \otimes B$, is an $(m_1 m_2) \times (n_1 n_2)$ matrix where:*

$$(A \otimes B)_{(i_1-1)m_2+i_2,(j_1-1)n_2+j_2} = A_{i_1,j_1} * B_{i_2,j_2}$$

for every $i_1 \in [m_1], i_2 \in [m_2], j_1 \in [n_1], j_2 \in [n_2]$.

Example 1.1 *Illustration for the Kronecker product of two 2x2 matrices:*

$$\begin{aligned}\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} &= \begin{bmatrix} a_{11} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{12} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ a_{21} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{22} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}\end{aligned} \tag{1.1}$$

Furthermore, let $A^{\otimes k}$ be the k-fold Kronecker power of a matrix A, which is defined inductively by $A^{\otimes 0} = I_1$ and $A^{\otimes k} = A^{\otimes(k-1)} \otimes A$. The Kronecker product is bilinear, associative and has the following mixed-product property: If the two products $A \cdot C$ and $B \cdot D$ of any matrices A, B, C, D are defined, then it also holds that $(A \otimes B) \cdot (C \otimes D) = (A \cdot C) \otimes (B \cdot D)$.
The Kronecker Product is bilinear and associative. It has the mixed product property: $(A \otimes B) * (C \otimes D) = (A * C) \otimes (B * D)$. It is not commutative, not even with regards to the identity matrix [4][Theorem 24].

1.2.4 Row and Column Spaces

Let A be a $m \times n$ Matrix with entries in the field $\mathbb{F}$.
The row space of A, written as $RS(A)$ is the subspace spanned by the rows of A with dimensions of $\mathbb{F}^n$. The column space of A is the subspace $CS(A)$ of $\mathbb{F}^m$ spanned by the columns of A.

$$RS(A) = \langle v \cdot A : v \in \mathbb{F}^n \rangle$$
$$CS(A) = \langle A \cdot v^\top : v \in \mathbb{F}^m \rangle$$

Example 1.2 *Consider the matrix*

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

The two rows $\begin{bmatrix} 1 & 2 & 4 \end{bmatrix}$ *and* $\begin{bmatrix} 3 & 1 & 2 \end{bmatrix}$ *span* $RS(A)$*. Note that they are linearly independent and form a Basis of* $RS(A)$*. The three columns of* A*,* $\begin{bmatrix} 1 & 3 \end{bmatrix}^\top$*,* $\begin{bmatrix} 2 & 1 \end{bmatrix}^\top$*,* $\begin{bmatrix} 4 & 2 \end{bmatrix}^\top$ *span* $CS(A)$*. Note that* $\begin{bmatrix} 2 & 1 \end{bmatrix}^\top$ *and* $\begin{bmatrix} 4 & 2 \end{bmatrix}^\top$ *are linearly dependent and as such aren't both needed to form the basis of the subspace formed by* $CS(A)$*. Instead one of them will suffice.*

Chapter 2

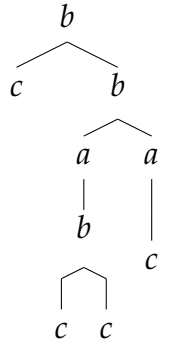
Multiplicitiy Tree Automata

2.1 Trees and contexts

2.1.1 Trees

The set of $\Sigma - trees$ or *trees* for short, written as T_Σ, is the smallest set T satisfying the following two conditions:
(i): $\Sigma_0 \subseteq T$
(i): if $k \geq 1, \sigma \in \Sigma_k$ and $t_1, \ldots, t_k \in T$ then $\sigma(t_1, \ldots, t_k) \in T$

Example 2.1 *A simple tree $t \in T_\Sigma$ for $\Sigma = a, b, c$*



The height of a tree t, denoted as $height(t)$ is defined inductively by $height(t) = 0$ if $t \in \sigma_0$, and $height(t) = 1 + max_{i \in [k]} height(t_1)$ if $t = \sigma(t_1, \ldots, t_k)$ for some $k \geq 1, \sigma \in \Sigma, t_1, \ldots, t_k \in T_\Sigma$.
For any $n \in \mathbb{N}_0$ let:

- $T_\Sigma^n := \{t \in T_\Sigma : height(t) = n\}$
- $T_\Sigma^{\leq n} := \bigcup_{l=0}^{n} T_\Sigma^l$
- $T_\Sigma^{<n} := T_\Sigma^{\leq n} \backslash T_\Sigma^n$

2.1.2 Contexts

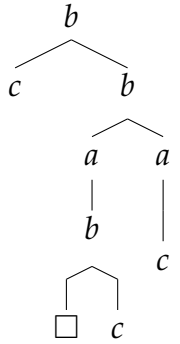
This section introduces a concept called context. Contexts allow a concatenation like operation on trees, extending a tree called context at a predefined point with another tree or context. [5]

Let $\Box$ be a nullary symbol not contained in Σ. If $\Box$ occurs as part of a tree which has a height greater than one it always takes a leaf position, since it is nullary. The set C_Σ of Σ-contexts (contexts for short) is the set of all $\{\{\Box\} \cup \Sigma\}$-trees in which $\Box$ occurs exactly once. As a result if $c \in C_\Sigma$ and $height(c) = 0$ it follows that $c = \Box$.
Let $n \in \mathbb{N}_0$. Let C_Σ^n denote the set of all contexts $c \in C_\Sigma$ where the distance between the root an the $\Box$-labeled node of c is equal to n.
A *subtree* of $c \in C_\Sigma$ is a Σ-tree consisting of a node in c and all of its descendants. Given a set $S \subseteq T_\Sigma$, $C_{\Sigma,S}^n$ is defined as the set of all contexts $c \in C_\Sigma^n$ where every sub-tree of c is an element of S. This amounts to building a set of contexts in which each context is constructed out of a number of given trees in S

- $C_\Sigma^{\leq n} := \bigcup_{l=0}^{n} C_\Sigma^l$
- $C_\Sigma^{<n} := C_\Sigma^{\leq n} \backslash C_\Sigma^n$
- $C_{\Sigma,C}^{\leq n} := \bigcup_{l=0}^{n} C_{\Sigma,S}^l$
- $C_{\Sigma,S}^{<n} := C_{\Sigma,S}^{\leq n} \backslash C_{\Sigma,S}^n$

Given $c \in C_\Sigma$ and $t \in T_\Sigma \sqcup C_\Sigma$, I write $c[t]$ for the tree obtained by substituting t for $\Box$ in c.

Example 2.2 *A simple context $c \in C_\Sigma^4$ for $\Sigma = a, b, c$*



2.1.3 Weighted Tree Series - Formal Power Series on Trees

Weighted Tree Series (WTS for short) are the underlying theoretical concept behind the multiplicity tree automata which will be introduced in the next section. WTS are a construct that associates each a tree $t \in T_\Sigma$ with an element from a field $\mathbb{F}$. More intuitively, in our examples, a tree series associates a number with a

tree, representing some kind of attribute of this tree. One simple example would be a WTS counting how often a certain symbol appears in a given tree. A more sophisticated example might be a WTS which accepts an arithmetic expression represented by a tree and evaluates it.
As mentioned in the introduction, the concept of *Formal Power Series on Trees* was introduced in [1, Recognisable formal power series on trees]. All results presented in this section are taken from his 1982 paper and are merely a short introduction to supply the theoretical and historical basis for the introduction of multiplicity tree automata.
For a detailed understanding one should consult the original paper, since I only mention the very basics here.

Definition 2.1 *Let T_Σ be a set of $\Sigma -$ trees and $\mathbb{F}$ a commutative field. A mapping from T_Σ to $\mathbb{F}$ is defined as a weighted tree series:*

$$f : T_\Sigma -> \mathbb{F}$$

By using a field as the co-domain one is able to associate a tree with a precise numerical value in practical applications. This is used in fields like natural language processing where trees often do not represent exact information but instead deal with the results of statistical analysis generated by data analysis of texts and other inputs [6]. Fields like the rational numbers are a natural operating space for such applications. The implementation presented in this thesis uses 64-Bit floating point numbers.

2.1.4 Recognisability of Tree Series

It turns out there are some tree series which are recognisable and some which are not. This notion of recognisability is directly linked to both the notion of the *Hankel Matrix* and that of the minimisation of tree automata.

2.1.5 Hankel Matrix of a Tree Series

Definition 2.2 *Hankel Matrix*
A tree series $f : T_\sum \to \mathbb{F}$ is represented by its Hankel Matrix. It is defined as:

$$H : T_\sum x C_\sum \to \mathbb{F} \text{ such that } H_{t,c} = f(c[t]) \text{ for every } t \in T_\sum \text{ and } c \in C_\sum$$

One can see that the columns of the matrix correspond to the possible contexts of the Tree Series and the rows of the matrix to the various trees. The values themselves represent the concatenation of the tree and the context in the respective row and column.

Example 2.3 *Let $\Sigma = a, b, c$ with $rk(c) = 0, rk(a) = 1, rk(b) = 2$. This is an example of a Hankel Matrix of a tree series f which will later be introduced as an example of a Multiplicity Tree Automata and which will be defined at that point. For now it is intended as a simple illustration of the structure of a Hankel Matrix.*

	$\square$	*b(□,c)*	*a(□)*	...
a(c)	*f(a(c))*	*f(a(b(c,c)))*	*f(a(a(c)))*	...
b(c,c)	*f(b(c,c))*	*f(b(b(c,c),c))*	*f(a(b(c,c)))*	...
c	*f(c)*	*f(b(c,c))*	*f(a(c))*	...
⋮	⋮	⋮	⋮	⋱

In this thesis, only examples of recognisable tree series will be considered, since, as will be shown later, those are able to be minimised. That said, to illustrate what exactly either a recognisable or unrecognisable tree series is, let us consider a few examples mentioned in [1].

Example 2.4 *Let Σ be a ranked alphabet and T_Σ a set of trees. The following tree series over T_Σ are recognisable:*

- *$num()$, where $num(t)$ is the number of nodes of a tree t.*
- *$eval()$, where $eval(t)$ is a computation on a tree which represents an arithmetic expressions without division. This example is defined and implemented later on.*
- *$length()$, with $length(t)$ sum of all path lengths from the root to the nodes of tree t.*

These tree series are not recognisable:

- *$height()$, where $heigh(t)$ is the height of a tree t.*
- *$eval()$, where t is a tree which represents an arithmetic expressions with division and $eval(t)$ is the value of said arithmetic tree.*

2.2 Multiplicity Tree Automata

The following part of is concerned with Tree Automata. The Weighted Tree Automata which are defined over a field are call Multiplicity Tree Automata (MTAs). They can be seen as an extension of bottom-up tree automata or a strict generalization of stochastic tree automata since they allow to define functions over any field $\mathbb{F}$ [5]. Informally, they can be seen as a automaton that associates a tree with a value in a field. A natural choice for such a field is $\mathbb{Q}$.

Definition 2.3 *Let $\mathcal{F}$ be a field. A $\mathbb{F}$-* ***multiplicity tree automata*** *is a 4-tuple $A = (n, \Sigma, \mu, \gamma)$*

- *$n \in \mathbb{N}$ is the dimension of the automaton, representing the number of states.*
- *Σ is a ranked alphabet. The automaton operates over trees which consist of these states, where each state has $rk(\sigma)$ children as defined in 2.1.*
- *$\mu = \{\mu(\sigma) : \sigma \in \Sigma\}$ is the* ***tree representation****. For every symbol $\sigma \in \Sigma$, $\mu(\sigma) \in \mathbb{F}^{n^{rk(\sigma)} \times n}$ represents the transition matrix of μ. These matrices store the weights of transitions from $rk(\sigma)$ origin states to the n possible destination states.*
- *$\gamma \in \mathbb{F}^{n \times 1}$ is the final weight vector applied after a complete run. It includes a weight for each of the n possible final states.*

Note that at this point an automata based on this definition can be completely represented in terms of matrices. Both the tree representation and the final weight vector are defined as such. This already implies a matrix-based implementation. It is furthermore possible to make statements about MTAs by using proofs which operate mostly on matrices.
This is very helpful since linear algebra in general and matrices in particular are well researched topics with extensive resources both in terms of the theoretical background as well as documented libraries available in every major programming language.

Definition 2.4 *Let $\mathcal{A}$ be a MTA with $\mathcal{A} = (n, \Sigma, \mu, \gamma)$.*

- *$|\mathcal{A}|$ denotes the* ***size*** *of $\mathcal{A}$, $|\mathcal{A}| := \Sigma_{\sigma \in \Sigma} n^{rk(\sigma)+1} + n$. This is the total number of entries in the transition matrices in μ and of the final weight vector.*
- *Extend the tree representation μ from Σ to T_Σ by defining $\mu(\sigma(t_1, ..., t_k)) := (\mu(t_1) \otimes ... \otimes \mu(t_k)) \cdot \mu(\sigma)$ be an* ***extension of the tree representation*** *.*
- *The automaton $\mathcal{A}$* ***recognizes*** *the tree series $||\mathcal{A} : T_\Sigma \to \mathbb{F}$ where $||\mathcal{A}||(t) = \mu(t) \cdot \gamma$ for every $t \in T_\Sigma$.*

As one can see, evaluating a tree over a MTA is done via both matrix multiplication and the use of the Kronecker product. As already pointed out, matrix multiplications are readiliy available in numerous efficient algorithms.

Note the difference in dimension between μ applied to a symbol $\sigma \in \Sigma$ and a tree $t \in T_\Sigma$:

- $\mu(\sigma)$ with $\sigma \in \Sigma_k$ is defined as a matrix $\mathbb{F}^{n^{rk(\sigma)} \times n}$
- $\mu(\sigma(t_1, ..., t_k))$ with $\{t_1, ..., t_k\} \subseteq T_\Sigma$ or $\mu(t)$ with $t \in T_\Sigma$ is defined as a matrix $\mathbb{F}^{1 \times n}$

This matrix indicates the weight in all states at the end of a run of t over $\mathcal{A}$. After $\mu(t) \cdot \gamma$, the final matrix has the dimensions $[1 \times n] \times [n \times 1] = [1 \times 1]$

In addition to the extension of the tree representation from symbols to trees, it will be further extended to also incorporate contexts, as defined in 2.1.2.

Definition 2.5 *Let $\mathcal{A}$ be a MTA with $\mathcal{A} = (n, \Sigma, \mu, \gamma)$.*

- *Let $\Box$ be a unary symbol and $\mu(\Box) := I_n$. Extend the tree representation μ from T_Σ to C_Σ. Define $\mu(c) \in \mathbb{F}^{n \times n}$ for every $c = \sigma(t_1, ..., t_k) \in C_\Sigma$ inductively as $\mu(c) := (\mu(t_1) \otimes ... \otimes \mu(t_k)) \cdot \mu(\sigma)$. For every $t \in T_\Sigma \dot{\cup} C_\Sigma$ and $c \in C_\Sigma$ it holds that $\mu(c[t]) = \mu(t) \cdot \mu(c)$.*
- *The size of an automaton A, written as $|A|$, is the total number of entries in all transition matrices and the final weight vector, i.e., $|A| := \sum_{\sigma \in \Sigma} n^{rk(\sigma)+1} + n$.*

Intuitively, $\mu(c[t])$ returns the weight of a tree with the tree consisting of the concatenation of t and c at the position of $\Box$.
Two MTAs are called equivalent if $||\mathcal{A}_1|| = ||\mathcal{A}_2||$ and a MTA is called minimal if there exists no equivalent MTA which has a strictly smaller dimension.

Note 2.1 *On the difference between Top-Down and Bottom-Up Weighted Tree automata: As described in [7], their expressive power is equal, with a slight difference in terms of bottom-up- and top-down-determinisability. Here I only deal with bottom-up tree automata, though the included code is written in a way that allows an easy extension to the top-down case, the only thing missing is some code that converts a top-down automata into a bottom-up automata. But that goes beyond the topic of this thesis.*

2.3 Examples of MTAs

Before continuing with the more abstract minimisation algorithm it is a good idea to get a better intuition and understanding by looking at a few illustrations of the theoretical groundwork. In the following section there are three examples of MTAs which, in increasing complexity, illustrate both the definition of an MTA and how they are used to calculate values of various trees.

1. An automaton which counts occurences of a symbol in a tree. This is a classic example already mentioned in [1].
2. An automaton which computes the value of a tree which reperesents an arithmetic function with addition, subtraction and multiplication. This example originates from [3].
3. An automaton which mixes RGB colors according to simple ratios of the RGB values. This example is an original creation.

2.3.1 Counting occurrences of a in a tree

Example 2.5 *Counting occurrences of a*

Let $\mathcal{A} = (n, \sum, \mu, \gamma)$ be a MTA. $n = 2$ which denotes the amount of states, $\sum = \{a, b, c\}$ is the alphabet with $rk(a) = 1$, $rk(b) = 2$, and $rk(c) = 0$. $\gamma = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$ is the final weight vector, and μ a tree representation with the following content:

$$a = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} b = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \quad c = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{2.1}$$

This automaton counts the number of occurrences of the symbol a in the input tree. This example was first introduced for formal power series for trees in [1], used in [5] and only very slightly adjusted to conform to the representation of MTAs used in this thesis.
There are two states in this automaton, called q_1 and q_2 in the following paragraphs. The transition matrix of b defines the transitions in the following way:

$$b = \begin{bmatrix} b(q_1, q_1) \xrightarrow{1} q_1 & b(q_1, q_1) \xrightarrow{0} q_2 \\ b(q_1, q_2) \xrightarrow{0} q_1 & b(q_1, q_2) \xrightarrow{1} q_2 \\ b(q_2, q_1) \xrightarrow{0} q_1 & b(q_2, q_1) \xrightarrow{1} q_2 \\ b(q_2, q_2) \xrightarrow{0} q_1 & b(q_2, q_2) \xrightarrow{0} q_2 \end{bmatrix} \tag{2.2}$$

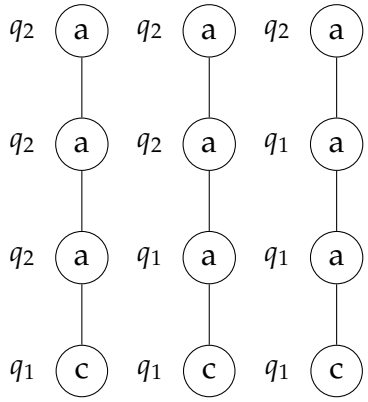
The basic intuition is as follows:

- Every non-zero run of the automaton corresponds to exactly one occurrence of a and has a weight of 1.
- The weight of a tree is the sum of the weight of all successful runs and as such the number of occurrences of a.
- The final weight vector $\gamma = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$ only counts the weight of successful runs ending in state q_2.
- The only way to achieve such a run is if it includes one of the following transitions: $a(q_1) \xrightarrow{1} q_2$ or $a(q_2) \xrightarrow{1} q_2$ or $b(q_1, q_2) \xrightarrow{1} q_2$ or $b(q_2, q_1) \xrightarrow{1} q_2$.
- c only has a non-zero transition to q_1 and as such isn't counted in any way in the end result.
- b only has non-zero transitions into q_2 if one of its input states already was q_2.
- As a result the only way to get a non-zero run ending in q_2 is if the run contains the transition $a(q_1) \xrightarrow{1} q_2$.
- Once there is a single occurrence of q_2 inside the current run it is not possible to have a non-zero transition to back to state q_1. Such a run wouldn't have a non-zero weight.
- Only one a is counted each run, since the transition $b(q_2, q_2) \xrightarrow{0} q_2$ is of weight zero.

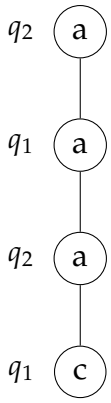
Lets take a look at the successful runs over an example tree to further illustrate this:

The simple example tree t_1 illustrates the basic idea behind this automaton quite well:

t_1 has exactly three successful runs, each with weight 1:



This is a run with weight zero in comparison. $a(q_2) \rightarrow_0 q_1$ has the weight of zero and as a result the whole run with these specific states has a weight of zero as well.



This is confirmed by computing $||A||(t_1)$ by applying the definition of $\mu(t)$:

$$\begin{aligned} \mu(t_1) &= \mu(c(a(a(a)))) \\ &= \mu(a) \cdot \mu(a) \cdot \mu(a) \cdot \mu(c) \\ &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 3 \end{bmatrix} \end{aligned} \tag{2.3}$$

with $||A||(t_1) = \mu(t) \cdot \gamma = \begin{bmatrix} 1 & 3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \end{bmatrix}^T = 3$.

2.3.2 Evaluation of arithmetic expressions

Example 2.6 *Evaluation of arithmetic expressions*

Let $\mathcal{A} = (n, \sum, \mu, \gamma)$ be a MTA with $n = 2$ states, $\sum = \{+, -, x, 0, 1\}$ a ranked alphabet with $rk(+) = rk(-) = rk(x) = 2$ and $rk(0) = rk(1) = 0$. $\gamma = [0, 1]^T$ as the final weight vector and μ a tree representation with the following transition matrices for the binary symbols:

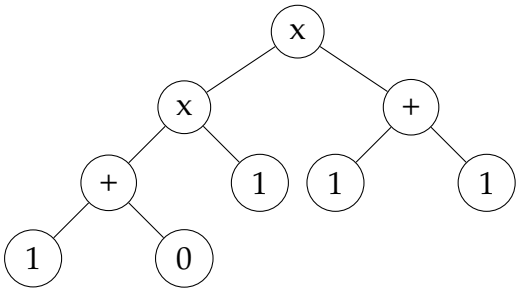
$$\mu(+) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \mu(-) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \mu(x) = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

and for the nullary symbols:

$$\mu(1) = \begin{bmatrix} 1 & 1 \end{bmatrix} \qquad \mu(0) = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

This example evaluates arithmetic expressions over addition, subtraction and multiplication. It originates from [3].
Note the absence of division, with which the tree series would not recognisable, as mentioned in 2.4.

Let $t_1 =$



$$
\begin{aligned}
\mu(t_1) &= \mu(x(x(+(1,0),1),+(1,1))) \\
&= \mu(x(x(+(1,0),1),+(1,1))) \\
&= (\mu(x(+(1,0),1)) \otimes \mu(+(1,1))) \cdot \mu(x) \\
&= ((\mu(+(1,0)) \otimes \mu(1)) \cdot \mu(x)) \otimes ((\mu(1) \otimes \mu(1)) \cdot \mu(+)) \cdot \mu(x) \\
&= (((\mu(1) \otimes \mu(0)) \cdot \mu(+)) \otimes \mu(1)) \cdot \mu(x)) \otimes ((\mu(1) \otimes \mu(1)) \cdot \mu(+)) \cdot \mu(x) \\
&= ((([1 \quad 1] \otimes [0 \quad 1]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}) \otimes [1 \quad 1]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}) \otimes (([1 \quad 1] \otimes [1 \quad 1]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix})) \cdot \mu(x) \\
&= (([1 \quad 0 \quad 1 \quad 0] \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}) \otimes [1 \quad 1]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}) \otimes [1 \quad 2]) \cdot \mu(x) \\
&= (([1 \quad 1 \quad 1 \quad 1]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}) \otimes [1 \quad 2]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \\
&= ([1 \quad 1]) \otimes [1 \quad 2]) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = [1 \quad 2 \quad 1 \quad 2] \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \\
&= [1 \quad 2]
\end{aligned}
\tag{2.4}
$$

The final weight of t_1 can be calculated as follows: $||A||(t_1) = \mu(t_1) \cdot \gamma = [1 \quad 2] \cdot [0 \quad 1]^T = 0 \cdot 1 + 2 \cdot 1 = 2$. One can see that the final weight matches the expectations when calculating the computational tree by hand.

2.3.3 RGB color-mixing

Example 2.7 *A simple form of color mixing*

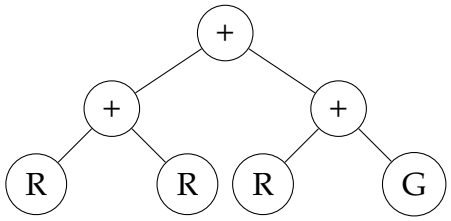
The following automata was constructed to simulate the mixing of RGB colors. It uses the three base colors of the RBG (red, green and blue) color space and combines them in pairs of twos to finally obtain the correct mixtures with regards to the RGB values.
It is apparent to anyone who has worked with RGB values before, that mixing them with simple ratios as done in this example, does not achieve the same kind of mixing as you would expect using water colors or something similar in real life. It still serves as a very instructive example for the way the different runs over a tree are computed.
Let $A = (n, \Sigma, \mu, \gamma)$ be a MTA with $n = 3$ states, q_1, q_2 and q_3. $\Sigma = \{R, B, G, +\}$ with R, B, G being nulary and and $+$ being binary.
Let the transition matrices of A be:

$$\mu(R) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \mu(B) = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \mu(G) = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

$$\mu(+) = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0 \\ 0.5 & 0 & 0.5 \\ 0.5 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0.5 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

The final weight vector is $\gamma = \begin{bmatrix} 256000000 & 256000 & 256 \end{bmatrix}^T$, which allows the representation of the final RBG value in a single number. The basic idea of the automaton is to compute a $\mu(t)$ for a tree t where $\mu(t) = [r, b, g]$ and r, b, g are the ratios of the colors with $r + b + g = 1$. The computation of $[r, b, g] \cdot \gamma = [256000000 \cdot r + 256000 \cdot b + 256 \cdot g]$ gives a single number that encodes the values of all three colors.
Looking at the transition matrix $\mu(+)$ one observes interesting pattern. For each combination of three states q_a, q_b, q_c, the weights of the corresponding row always add up to 1. This makes sense, since the transition matrix 'splits' the input weights evenly between two colors.
If one was so inclined, it would be easy to extend the symbol $+$ to have $rk(+) = 3$ by adding rows of the type $\begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$, which would 'split' three input color evenly.

Let $t_1 =$



$$\begin{aligned}
\mu(t_1) &= \mu(+(+(R,R),+(B,G))) \\
&= (\mu(+(R,R)) \otimes \mu(+(R,R))) \cdot \mu(+) \\
&= ((\mu(R) \otimes \mu(R)) \cdot \mu(+)) \otimes (((\mu(B) \otimes \mu(G)) \cdot \mu(+)) \cdot \mu(+)) \\
&= ((\begin{bmatrix}1 & 0 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0 & 0\end{bmatrix}) \cdot \mu(+) \otimes (((\begin{bmatrix}0 & 1 & 0\end{bmatrix} \otimes \begin{bmatrix}0 & 0 & 1\end{bmatrix}) \cdot \mu(+)) \cdot \mu(+)) \\
&= \begin{bmatrix}0.5 & 0.25 & 0.25\end{bmatrix}
\end{aligned} \tag{2.5}$$

With $||A||(t_1) = \mu(t_1) \cdot \gamma = \begin{bmatrix}0.5 & 0.25 & 0.25\end{bmatrix} \cdot \begin{bmatrix}256000000 & 256000 & 256\end{bmatrix}^T = 0.5 \cdot 256000000 + 0.25 \cdot 256000 + 0.25 \cdot 256 = 128064064$. As described, one can interpret this value, with appropriate normalisation, as an RGB value, though obviously the colors don't exactly match our intuition.
This is an interesting example since it illustrates that even though a MTA 'only' associates one value with each tree, that value can have a quite complicated meaning and encode a lot of information.

Chapter 3

Implementation of Multiplicity Tree Automata

Currently, there exist a number of tree automata frameworks which already implement different kinds of tree automata. Those are written in different languages and are mostly tailored to facilitate further research:

- Tiburon [8] implements weighted top down tree automata. It is mostly concerned with natural language processing.
- Mona [9] deals with finite-state automata which operate over binary decision diagrams.
- Forest Fire [10] implements a large number of algorithms for tree automata.
- Timbuk 3.2 [11] implements unweighted bottom-up non-deterministic finite tree automata as part of a tool that is designed with achieving proofs of reachability over Term Rewriting Systems in mind.
- libVATA [12] implements unweighted non-deterministic finite tree automata.

Among the libraries listed above Timbuk [11] has seen a large amount follow up research based on its code, with the last paper extending it being from 2018 [13]. Others have never seen much activity after their initial release. Forest Fires website [14] for example seems to be offline at the time of writing.
libVATA is accompanied by a very detailed design document and has seen some follow-up work after its initial release, which made it the most promising library on which to base the implementation of the minimisation described in this thesis. After some consideration the decision to simply re-implement a very basic MTA instead of extending an existing library was made. The definition of an MTA described in 2.3 only relies on basic linear algebra, therefore the benefit of relying on an existing code-base seemed smaller than the work required to familiarize myself with that code-base.
The programming language of choice for the implementations done for this thesis

is Rust. Two libraries are used beyond the existing standard library. One is a numeric algebra library called *nAlgebra* [15] which provides basic matrix implementations and operations. The other one is a small implementation of a tree structure called *trees* [16]. It is only used to save and traverse trees in a structured fashion without providing further functionality. Both libraries are provided under the Apache-2.0 license.
The code is formatted based on the default Rust style. An exception was made regarding variable names which include non-standard letters like σ and γ. This hopefully allows the reader to better compare parts of the thesis and the code.

3.1 Multiplicity Tree Automata Format

To write and save MTAs, a simple text format was chosen. All transitions of each symbol μ are encoded in a single line of the format:

$$\sigma\ \{left\_side\_states\}\ \text{->}\ target\_state\ weight,$$

where σ is the symbol associated with the transition matrix μ which contains this specific transition, $\{left\ side\ states\}$ lists the input states, of which $rk(\sigma)$ exist. *target state* is the output state of the transition, and is limited to one state. The final weight vector is encoded in the format:

$$!\ state\ \text{->}\ weight.$$

While parsing this file the amount of states n is computed by counting the number of states which are present. The alphabet is constructed out of each distinct σ with its rank being computed based on the number of input states.

Example 3.1 *Encoding of the example 2.3.1:*

```
1  c -> q1 1
2  c -> q2 0
3
4  a q1 -> q1 1
5  a q1 -> q2 1
6  a q2 -> q1 0
7  a q2 -> q2 1
8
9  b q1 q1 -> q1 1
10 b q1 q1 -> q2 0
11 b q1 q2 -> q1 0
12 b q1 q2 -> q2 1
13 b q2 q1 -> q1 0
```

```
14          b q2 q1 -> q2 1
15          b q2 q2 -> q1 0
16          b q2 q2 -> q2 0
17
18          ! q1 0
19          ! q2 1
20
```

After parsing, the MTA is saved in a simple struct. The transition matrices which form the tree representation are stored as dynamically sized matrices over 64-Bit floating point numbers.
Both, the alphabet and the transition matrices are stored in a hashmap with the key being the associated symbol $\sigma \in \Sigma$.

```
1 struct WeightedTreeAutomatonMatrix {
2     num_states: int,
3     ranked_alphabet: HashMap<String, int>,
4     transition_matrices: HashMap<String, DMatrix<f64>>,
5 }
```

The tree representation is saved as a hash map containing the transition matrix of each symbol, with the symbol being the key. The final weight vector is part of that hash map with the key **!**. The ranked alphabet is saved in a hash map which allows efficient access.
A hash map was chosen since after the initial step of parsing, all interaction with the tree representation and the alphabet is limited to read access.
The number of states is saved as an unsigned integer for convenience, mostly just to prohibit constant recalculation of it. It is calculated during the parsing from the number of states which are part of the tree representation and the final weight vector.

3.2 Computation of $||A||(t)$

Finally the definition of the tree representation in 2.3 implies a simple recursive algorithm to compute $\mu(t)$ and $||A||(t)$.
rec denotes a recursive call of the algorithm. Afterwards $\mu(t) * \gamma = ||A||(t)$ gives the value of t. The distinction between the different amount of child nodes is used to distinguish between multiplication and the Kronecker product implied in the definition.

Data: ℚ-multiplicity tree automaton A(n, Σ, μ, γ), tree t
Result: $\mu(t)$

```
σ := root(t).symbol
m := root(t).num_children()
if m == 0 then
    return μ(σ)
if m == 1 then
    v := rec(t.get_child[0])
    return μ(σ) · v
if m > 1 then
    result := rec(t.get_child[0] ⊗ rec(t.get_child[1])
    for i in 2..m do
        result = result ⊗ rec(t.get_child[i])
    result = result · μ(σ)
    return result
```

Algorithm 1: Recursive computation of μ(t)

Chapter 4

Minimisation of Multiplicity Tree Automata

The first part of this thesis was concerned with introducing multiplicity tree automata, giving illustrative and comprehensible examples and showing a simple way to use the algebraic foundation to implement these examples. The following, second part will introduce a procedure to minimise a given multiplicity tree automata, as outlined in [2].

4.1 Theoretical foundations of minimisation

4.1.1 Rank of Hankel matrix

Theorem 4.1 *[5] [17] Let Σ be a ranked alphabet, $\mathbb{F}$ be a field, and $f : T_\Sigma \to \mathbb{F}$ a tree series. Let H be the Hankel matrix of f. Then, f is recognised by some MTA if and only if H has finite rank over $\mathbb{F}$. In case H has finite rank over $\mathbb{F}$, the dimension of a minimal MTA recognising f is rank(H) over $\mathbb{F}$.*

Note that while the matrix H, as a result of its definition in 2.1.5, is always infinite in its dimensions, the rank of H doesn't need to be.
There are multiple ways to define or compute the rank of the matrix. The one which will become relevant in this context is the number of linearly independent rows, which is equal to the number of linearly independent columns. [18, p. 204]

4.1.2 Forward space and backward space

First fix a Field $\mathbb{F}$ and an Automaton $A = (n, \Sigma, \mu, \gamma)$ which will be used in this section.

Definition 4.1 *Let T_Σ be a set of Σ-trees. The forward space $\mathcal{F}$ is the row space $\mathcal{F} := \langle \mu(t) : t \in T_\Sigma \rangle$.*

This vector space is spanned by the transition matrices $\mu(t)$ of all possible trees in T_Σ. All these transition matrices have dimension $1 \times n$. To give further context, if one would compute $\mu(t) * \gamma$ with $\mu(t) \in \mathbb{F}$, one would get $||A||(t)$.

Proposition 4.1 *The forward space $\mathcal{F}$ has the following properties:*

(a) [2, Lemma 3.1] The forward space $\mathcal{F}$ is the smallest vector space V over $\mathcal{F}$ such that for all $k \in \mathbb{N}_0$, $v_1, ..., v_k \in V$, and $\sigma \in \Sigma_k$ it holds that $(v_1 \otimes \cdots \otimes v_k) \times \mu(\sigma) \in V$
(b) [19, Main Lemma 4.1] The set of row vectors $\{\mu(t) : t \in T_\Sigma^{\leq n}\}$ spans $\mathcal{F}$.

Here, (a) implies that the extended tree representation μ represented by $(v_1 \otimes \cdots \otimes v_k) \times \mu(\sigma) \in V$ is closed over the forward space.
This means that it is possible to take a tree t out of the set of trees for which $\mu(t) \in \mathcal{F}$ and use that tree as one of the leafs in another node σ, together with $rk(\sigma) - 1$ other trees, for which $\mu(t_i) \in \mathcal{F}$. The result of $(\mu(t_1) \otimes \cdots \otimes \mu(t_r k(\sigma)) \cdot \mu(\sigma)$ is a transition matrix $\mu(t') \in \mathcal{F}$ for a new tree $t' \in T_\Sigma$ with $\mu(t') \in V$.
Now, (b) implies that it is possible to construct the forward space by only looking at the trees of size smaller than the number of states. This is essential, since it implies that there is a finite number of trees which span $\mathcal{F}$, since the set $\{\mu(t) : t \in T_\Sigma^{\leq n}\}$ is, by definition, finite.
An algorithm to compute a minimal set of row vectors which span the forward space is already implied: Find the smallest set of linearly independent vectors in $\{\mu(t) : t \in T_\Sigma^{\leq n}\}$ which forms a basis for $\mathcal{F}$. This will be used in the next sections.

Definition 4.2 *Let C_Σ be a set of Σ-contexts. The backward space $\mathcal{B}$ is the column space $\mathcal{B} := \langle \mu(c) \cdot \gamma : c \in C_\Sigma \rangle$.*

Proposition 4.2 *[2, 3.2] Let $S \subseteq T_\Sigma$ be a set of trees such that $\{\mu(t) : t \in S\}$ spans $\mathbb{F}$. Then, the following properties hold:*

(a) The backward space $\mathcal{B}$ is the smallest vector space V over $\mathbb{F}$ such that:
  (1) $\gamma \in V$
  (2) For every $v \in V$ and $c \in C^1_{\Sigma,S}$ it holds that $\mu(c) \times v \in V$.
(b) The set of column vectors $\{\mu(c) \cdot \gamma : c \in C^{\leq n}_{\Sigma,S}\}$ spans $\mathcal{B}$.

Since, according to (a), $\gamma \in V$ and according to the definition of the forward space $\mu(t) \in \mathcal{F}$ for all $t \in T_\Sigma$, it is easy to see that one can compute $||A||(t) = \mu(t) \cdot \gamma$ for every t by multiplying a $1 \times n$ matrix $f \in \mathcal{F}$ with a $n \times 1$ matrix $b \in \mathcal{B}$.

Similarly to (b) in 4.1, (b) in this proposition implies that there is a finite number of contexts that span $\mathcal{B}$. Once again this implies a potential algorithm to compute a minimal set of column vectors which span $\mathcal{B}$. That said, as will be seen later, (a) implies an even more efficient way to compute the basis of $\mathcal{B}$. This way will be shown in detail, alongside an example which will illustrate the exact meaning of $C^1_{\Sigma,S}$.

4.1.3 A minimal automata

Lemma 4.1 *[2, Lemma 3.3] A minimal automaton equivalent to $\mathcal{A}$ has $m := rank(F \cdot B)$ states.*

This theorem implies that there is a relationship between the the minimum size of the automata on the one hand and the $\mu(t)$ with $t \in T_\Sigma$ and the $\mu(c)$ with $c \in C_\Sigma$ which both form the Hankel Matrix and span the forward and backward spaces on the other. This relationship will be used in the following section to arrive at the minimisation algorithm.

Definition 4.3 *Let $m := rank(F \cdot B)$ and $\tilde{F}$ be a matrix $\tilde{F} \in \mathbb{F}^{m \times n}$, which span $RS(F \cdot B)$. Let $\tilde{A} = (m, \Sigma, \tilde{\mu}, \tilde{\gamma})$ be a multiplicity tree automata with $\tilde{\gamma} = \tilde{F} \cdot \gamma$ and*

$$\tilde{\mu}(\sigma) \cdot \tilde{F} = \tilde{F}^{\otimes k} \cdot \mu(\sigma) \quad \textit{for every } \sigma \in \Sigma_k \tag{4.1}$$

Theorem 4.2 *[2, Proposition 3.4] The MTA $\tilde{A} = (m, \Sigma, \tilde{\mu}, \tilde{\gamma})$ is well-defined and a minimal MTA equivalent to $\mathcal{A}$.*

As already described in the respective sections, 4.1 (b) and 4.2 (a) imply algorithms which compute matrices F and B.
Definition 4.3 now suggests two necessary steps to finally arrive at $\tilde{A}$. First the computation of $\tilde{F}$, based on F and B and secondly solving equation 4.1 for each instance of $\sigma \in \Sigma$.
This algorithm will be given concretely and illustrated by example in the following section.

4.2 The minimisation algorithm

The algorithm which Keifer et al. [2] give, consists of three separate steps:

- **Step I** computes a matrix F which spans the forward space.
- **Step II** computes a matrix B which spans the backwards space.
- **Step III** computes $\tilde{F}$ as described in lemma 4.3 and solves the equation 4.1, which in turn gives a minimised automaton as output.

Step I of the algorithm is explicitly given by Keifer et al. in [2, Table 1] and was adopted without any changes. **Step II** and **Step III** on the other hand are only contained implicitly in the mentioned paper. They are made explicit in this thesis by giving a description in the form of pseudo code. For the sake of brevity, the algorithms are generally kept shorter and more general than an actual implementation would require. Furthermore a supporting algorithm 4.3.2 by [20, Tzeng] was slightly adjusted and is given as pseudo-code in the supplementary material.
An complete implementation of the algorithm, consisting of **Step I**, **Step II**, **Step III** is given as supplementary material in 7.1, and serves to illustrate the algorithm as a whole. Care was taken to inlude a large number of comments, which can be used as further material if parts of the following sections remain unclear.
The pseudo code is missing some implementation specific information, sometimes in relation to the language used, sometimes to the linear algebra library, and sometimes there are certain conditional statements which limit the amount of iterations or similar improvements to execution speed which are implemented in the actual source code but not mentioned in the pseudo code. In general the the pseudo code clarity was preferred while in the actual implementation there are a number of small changes which result in improvements for speed and memory. As a result the source code differs slightly from the pseudo code.
That said, the main structure of the algorithms is mirrored in the attached source code and it should be easy to follow it while reading its pseudo counterpart.
The pseudo code follows general convention, with:

- **F := []** initialises an empty matrix.
- **H := {key : value}** initialises a hash map with a single key-value pair.
- **for i in a..b** is a for-loop which iterates over the integers from a to b-1.

4.2.1 Accompanying example

This chapter contains an example, which will hopefully illustrate some of the more complex aspects of the presented algorithm. It is also provided in the attached code, with all intermediate results given as part of this example having been computed by this implementation.

Example 4.1 *Let $\mathcal{A} = (n, \sum, \mu, \gamma)$ be 'a'-counting MTA which was described in 2.3. $n = 2$, $\Sigma = \{a, c\}$ is the alphabet with $rk(a) = 1$ and $rk(c) = 0$. $\gamma = \begin{bmatrix}0 & 1\end{bmatrix}^T$, and μ a tree representation with the following content:*

$$a = \begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} b = \begin{bmatrix}1 & 0\\ 0 & 1\\ 0 & 1\\ 0 & 0\end{bmatrix} \quad c = \begin{bmatrix}1 & 0\end{bmatrix} \tag{4.2}$$

I construct a automaton which doubles the number of states of A:
$\mathcal{B} = (n', \Sigma, \mu', \gamma')$ with $n' = 4$, $\gamma' = \begin{bmatrix}0 & 1 & 0 & 1\end{bmatrix}^T$ is the final weight vector, and μ' a tree representation with the following content:

$$a = \begin{bmatrix}1 & 1 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 1\\ 0 & 0 & 0 & 1\end{bmatrix} c = \begin{bmatrix}1 & 0 & 1 & 0\end{bmatrix}$$

$$b = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0\end{bmatrix}^T$$

It is easy to see that $||B||(t) = 2 \cdot ||A||(t)$ for all $t \in T_\Sigma$. In this case, it is obvious that there exists a minimal automaton $\tilde{B}$ of B. One possible construction would be $B' = (n, \Sigma, \mu, \gamma')$ with $\gamma' = 2 \cdot \gamma$. The algorithm will construct an equivalent automaton, which will differs from $\tilde{A}'$ in its basis as described in [21] [Proposition 4].

4.2.2 Step I: "Computation of Matrix F - Spanning the Forward Space"

The algorithm begins with a sub-step which computes the matrix F, which spans the forward space. The construction of this matrix is based on the already discussed implications of Proposition 4.1 (b) in 4.1.
The general idea is the following: All trees $t \in T_\Sigma^{\leq n}$ are constructed, $\mu(t)$ is computed and if $\mu(t)$ is linearly independent from its predecessors it is added as a row to F.
This is done by iterating over all possible symbols σ [line 4] and all combination of child trees [line 5] by creating new values $\mu(t)$ [var v] from previous linearly independent transition matrices $\mu(t)$ and the current symbol σ.

Data: $\mathbb{Q}$-multiplicity tree automaton $\mathrm{A}(n, \Sigma, \mu, \gamma)$
Result: Matrix F whose rows form a basis of the forward space $\mathcal{F}$

```
1  i := 0
2  j := 0
3  F := []
4  while i ≤ j do
5  |  for every σ in Σ do
6  |  |  for (l_1, ..., l_rk(σ)) ∈ [i]^rk(σ) \ [i − 1]^rk(σ do
7  |  |  |  v := (F_{l_1} ⊗ ··· ⊗ F_{l_rk(σ)}) · μ(σ)
8  |  |  |  if v ∉ ⟨F_1, ..., F_j⟩ then
9  |  |  |  |  j := j + 1
10 |  |  |  |  F.add_row(v)
11 |  i := i+1
12 return F
```

Algorithm 2: Step I - Computation of Matrix F

General notes about Step I:

- j represents the row index of the Matrix F which is already linearly independent rows.
- $(l_1, ..., l_{rk(\sigma)}) \in [i]^{rk(\sigma)} \setminus [i-1]^{rk(\sigma}$ in [line 6] are all tuples as defined by $[i]^{rk(\sigma)}$ without those that don't include the number i. These are internally generated by an algorithm called **next_tuple**, which can be seen at 4.3.1.
- Each row F_j represents a $[1 \times n]$ matrix, where $F_j = \mu(t)$ for some $t \in T_\Sigma^{\leq n}$.
- During the first iteration with $j = 0$ there exist no rows F_j in F which can be used in the computation of $v := (F_{l_1} \otimes \cdots \otimes F_{l_{rk(\sigma)}}) \cdot \mu(\sigma)$. As a result the first (few) row(s) of F are defined by those σ where $rk(\sigma) = 0$. The

accompanying implementation includes a case distinction which prevents any issues of that kind.

Example 4.2 *Computation of* ***Step I*** *for automaton B*
Computation of *F* ***for MTA-****B:*
In this example, one possible solution for matrix F which spans $\mathcal{F}$ for the automaton B is:

$$F = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \tag{4.3}$$

In this case row $F_1 = \mu'(c)$ and row $F_2 = \mu'(a(c))$. In this specific instance these two rows are also generated by the only potential trees that come into questions, since they are the only trees with $height(t) < 2$ which can be constructed. Since both rows are linearly independent they are both added to F.

4.2.3 Step II: "Computation of Matrix B - Spanning the Backward Space"

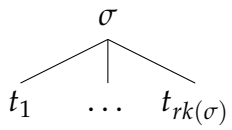
This step is concerned with the computation of a matrix B which columns span the column space $\mathcal{B}$.
As already mentioned, Proposition 4.2 (b) implies the computation of a matrix B, similarly to the computation of F in **Step I**. The solution in **Step I** was to compute all $t \in T_\Sigma^{\leq n}$. This would also be possible in the case of B by computing $\{\mu(c) \cdot \gamma : c \in C_{\Sigma,S}^{\leq n}\}$, but (a) already implies a more efficient solution:

1. Construct a set of matrices $M \in \mathcal{M} := \{\mu(c) : c \in C_{\Sigma,S}^1\}$, where $S \subseteq T_\Sigma$ is a set of trees such that $\{\mu(t)\}$ spans F.
2. Use an algorithm described by Tzeng [20] to construct the basis of the vector space V with $\gamma \in V$ and $M \cdot v \in V$.
3. B is formed by the linearly independent columns which span the vector space V.

The mentioned algorithm by Tzeng is given as pseudo code with additional notes in 4.3.2. The implementation is part of the attached code.

General notes about Step II:

- Each c has the following structure, with one $t_i = \square$.

c = 

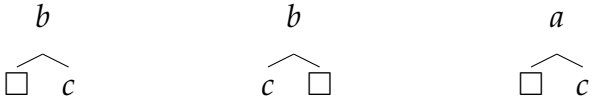
- S is the set of all rows of $F_i \in F$, where F_i is a vector $\mu(t)$ with $t \in T_\Sigma^{\leq n}$ and with dimension $1 \times n$.
- The Kronecker Product is not commutative. As a result one needs to compute a new M for each possible position of I_n.
- Computing all $M \in \mathcal{M}$ can be done by iterating over all symbols $\sigma \in \Sigma$, computing $(t_{l_1}, ..., t_{l_{rk(\sigma)}}) \cdot \mu(\sigma)$ for all $(l_1, ..., l_{rk(\sigma)}) \in [i]^{rk(\sigma)}$ and replacing one t_1 with $\square$.
- $(l_1, ..., l_{rk(\sigma)}) \in [i]^{rk(\sigma)}$ in [line 4] once again generates a number of tuples by internally calling **next_tuple** as described in 4.3.1. This time all tuples are considered, not only those which include at least one instance of i.

Data: Matrix F from **Step I** and automaton A
Result: Matrix B whose rows form a basis of the backwards space $\mathcal{B}$

```
1  m := {}
2  for σ ∈ Σ with rk(σ) > 0 do
3  |  for pos in 0..rk(σ) do
4  |  |  for (l_1, ..., l_rk(σ)) ∈ [i]^rk(σ) do
5  |  |  |  m' := [1]
6  |  |  |  v := [F_l1, ..., F_lrk(σ)]
7  |  |  |  v[pos] = I_n
8  |  |  |  for v in V do
9  |  |  |  |  m' = m' ⊗ v
10 |  |  |  m.push(m' * μ(σ))
11 B := tzeng(γ, B, M)
12 return B
```

Algorithm 3: Step II - Computation of Matrix B

Example 4.3 *Computation of **Step II** for automaton B*
Construction of** $M \in \mathcal{M}$ **for MTA B:
As described above, $F = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$ *is a possible result of **Step I** for automata B, where* $F_1 = \mu'(c)$ *and* $F_2 = \mu'(a(c))$.
Let c_1, c_2, c_3 *be the following contexts:*



*Then $\mu(c_1), \mu(c_2), \mu(c_3) \in \mathcal{M}$ according to the definition of $\mathcal{M}$ above and as computed in **Step II** for the automaton B.*

***Computation of Matrix B for automaton** B:*

*In this example one possible solution computed by **Tzeng()** based on $\mathcal{M}$, as described above, is:* $B = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}^T$

*This matrix can look slightly different depending on the sequence in which the elements of $\mathcal{M}$ are processed by **tzeng()**. This might change from run to run, depending on what kind of data-structure is used to save the matrices M, since not all data-structures in all languages guarantee a deterministic ordering.*

This is the place in the attached example implementation where non-determinism resides. Note that the Matrix has dimensions 4×2, which fits with the prediction of the minimal MTA $\tilde{B}$ being of dimension 2.

4.2.4 Step III: "Computation of the Minimal Automaton"

Step III is concerned with first computing $\tilde{F}$, as described in definition 4.3 and afterwards solving equation 4.1 for each $\sigma \in \Sigma$.

1. $\tilde{F}$ is computed by only including those rows F_j which are linearly independent from the previous rows if multiplied by B. Notice this doesn't imply that one should add $F_j \cdot B$ to $\tilde{F}$, but instead the original row F_j.
2. Fill a hash map with instances of $\tilde{F}^{\otimes k}$ needed to solve equation 4.1.
3. Compute a decomposition of $\tilde{F}$.
4. Compute the right side of the equation, $\tilde{F}^{\otimes k} \cdot \mu(\sigma)$ for each $\sigma \in \Sigma$ and then, going row by row, solve the equation and fill up the matrix $\tilde{\mu}(\sigma)$.
5. Compute the final weight vector and return the minimized automaton $\tilde{A}$.

Data: Matrix F from **Step I**, Matrix B from **Step II**, Automaton A
Result: Minimal Q-multiplicity tree automaton $\tilde{A}(m, \Sigma, \mu', \gamma')$

$\tilde{F}$:= []
for row in F **do**
    f := row · B
    if $f \notin \langle \tilde{F}_1, \ldots, \tilde{F}_{rank(\tilde{F})} \rangle$ **then**
        $\tilde{F}$.add(row)
$\tilde{F}^{\otimes k} = \{0 : I_1\}$
$\tilde{F}^{\otimes k}$.insert$(1, \tilde{F})$
max := $rk(\sigma')$ with $\sigma' \in \Sigma$ and $rk(\sigma') > max(\{rk(\sigma) : \sigma \in \Sigma \backslash \{\sigma\}\})$
for k in 2..max **do**
    $\tilde{F}^{\otimes k}$.insert$(k, F^{\otimes k}[k-1] \otimes \tilde{F})$
decomp = $\tilde{F}^{\intercal}$.decomp()
$m = \tilde{F}$.count_rows()
$\tilde{\mu} = \{\}$
for $\sigma \in \Sigma$ **do**
    $R = \tilde{F}^{\otimes k}[rk(\sigma)] * \mu(\sigma)$
    $\mu(\tilde{\sigma})$ = []
    for r in R.rows() **do**
        $\tilde{r}$ = decomp.solve$(r^{\intercal})$
        $\mu(\tilde{\sigma})$.add$(\tilde{r})$
    $\tilde{\mu}$.add$(\tilde{\mu}(\sigma))$
$\tilde{\gamma} = \tilde{F} \cdot \gamma$
return $\tilde{A}(m, \Sigma, \tilde{\mu}, \tilde{\gamma})$

Algorithm 4: Step III - Solve

General notes about Step II:

- $\tilde{F}^{\otimes k}$ should be a hashmap to allow quick read access. Since $\tilde{F}^{\otimes k}$ is needed for the computation of σ with $rk(\sigma)$ and each $\tilde{F}^{\otimes k}$ is computed with $\tilde{F}^{\otimes k} = F^{\otimes k-1} \otimes F$ it makes sense to simply compute $\tilde{F}^{\otimes k}$ for the largest $rk(\sigma)$ and save all intermediate values.
- In [line 10] a decomposition of $\tilde{F}$ is generated, which in the following for-loop will be used to solve equation 4.1. This can be reduced to solving a linear equation, which is a long standing problem in computer science, see for example [22].
- R in [line 14] is the right side of equation equation 4.1.
- $\tilde{\mu}(\sigma)$ has dimension $[m^{rk(\sigma)} \times m]$.

Example 4.4 *Computation of **Step III** for automaton B*
***Computation of** $\tilde{F}$:*
It is easy to see that both rows of F in this example, when they are multiplied by B, are still linearly independent of each other. As a result one arrives at

$$\tilde{F} = F = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

***Computation of** $\tilde{\mu}(\sigma)$ **for all** $\sigma \in \Sigma$:*
By solving equation equation 4.1, it is possible to compute $\tilde{\mu}(\sigma)$ for all $\sigma \in \Sigma$. This is mostly just an exercise in solving linear equations:

$$\mu(a) = \begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix} \mu(b) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ -1 & 2 \end{bmatrix} \mu(c) = \begin{bmatrix} 1 & 0 \end{bmatrix} \gamma = \begin{bmatrix} 0 & 2 \end{bmatrix}^T$$

One concrete example is given in the following to demonstrate the concrete meaning of $\tilde{F}$:

$$\tilde{\mu}(c) \cdot \tilde{F} \cdot B = \tilde{F}^{\otimes k} \cdot \mu(c) \cdot B$$

$$\tilde{\mu}(c) \cdot \tilde{F} \cdot B = \mu(c) \cdot B$$

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$$

As one can see $\tilde{F}$ acts as a kind of 'translation layer', where it 'translates' the weight of the two news states into the weight of the four old states.

$$\tilde{\mu}(a) \cdot \tilde{F} \cdot B = \tilde{F}^{\otimes k} \cdot \mu(a) \cdot B$$

$$\begin{bmatrix} 0 & 1 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

The value of $\tilde{F}^{\otimes k}$ in case of $rk(\sigma) > 1$ similarly 'translates' the child nodes of σ from the weight of the two new states into the weight of the four old states. This becomes apparent by looking at the definition of $\mu(\sigma(t_1, ..., t_k)) := (\mu(t_1) \otimes ... \otimes \mu(t_k)) \cdot \mu(\sigma)$, which mirrors the structure of $\tilde{F}^{\otimes k}$.

4.3 Supplemental Algorithms

4.3.1 Generating Permutations with Repetition

Data: Vector $[x_1, ..., x_k]$, integer n, boolean force_n
Result: Vector $[y_1, ..., y_k]$ or None

```
1  if k == 0 then return None
2  j := k - 1
3  while x[j] == n do
4  |  if j == 0 then return None
5  |  j = j - 1
6  for l in j..k do
7  |  if x[l] == n then
8  |  |  x[l] = 1
9  |  else
10 |  |  x[l] = x[l] + 1
11 if (n in x) or (force_n == false) then
12 |  return x
13 else
14 |  x = next_tuple(x, n, force_n)
15 |  return x
```

Algorithm 5: Compute next permutation of the input particular vector

The problem of generating all possible permutations with repetition is a well know problem. This algorithm is needed in two instances: once in **Step I** to compute the tuples while also excluding every tuple which does not contain an instance of i and once in **Step II** for the computation of $\mathcal{M}$. It is not necessary to exclude certain tuples in the second instance, so it made sense to include a simple

case distinction.
While the function does call itself if it is instructed to exclude certain tuples, it calls itself at most $n - 1$ times while looking for the next instance containing n, as a result there is no real concern regarding speed and memory usage.

4.3.2 Tzeng's Algorithm

Data: Weight vector γ, set of matrices $\mathcal{M}$
queue = [γ]
B = empty_matrix
while queue not empty **do**
    c = queue.pop()
    if c $\notin \langle B^1, ..., B^{\text{num\_columns\_B}} \rangle$ **then**
        B.append_column(c)
        for m in $\mathcal{M}$ **do**
            new_c = m * c
            queue.push(new_c)
return B

Algorithm 6: Computation of a Matrix B which columns spans V

This is a slightly adjusted implementation of Tzeng's algorithm from [20].
B is initialized as an empty matrix. $\mathcal{M}$ is computed in **Step II**, γ is simply the weight vector of our original automaton.
Each iteration of the while-loop checks the independence of the top most vector in the queue against the most up to date version of B and, in the case of c being linearly independent of all previously added column vectors of B, B is updated by adding c as a new column.
Afterwards, all possible new vectors are generated by multiplying all matrices $m \in \mathcal{M}$ with c.
For further details or a proof of correctness the reader is referred to Tzeng's paper.

Chapter 5

Implementation of the minimisation algorithm

Attached to the thesis is the complete source code of the implementation. This consists of a Rust project which is structured along the standard guidelines of the language.
An attached **README** contains instructions regarding building the project, executing the project and creating detailed debug information. The debug mode can be enabled for each module of the project, this includes the parsing, the computation of weights for trees and each part of the minimisation process separately.
It is also possible to enable the debug mode for the whole project at once, though this is not recommended since the full debug information of a complete run is very large.
The project includes aa *automata* folder, containing text-files which define the three examples in 2.3, based on the format defined in 3.1.
Each of the example MTAs is then duplicated with its state-count and final weight doubled. This is done the same way as described in the example case which was given for the a-counter in 4.2.1.
One can execute the main binary as follows:

minimisation_of_multiplicity_tree_automata –automaton "computation"

with the argument for the parameter **automaton** either **rgb**, **counter**, **computation** or **all**.
As mentioned, further information can be found inside the attached **README**.
Care was taken to include a lot of comments in the source code and structure it in a way that is comprehensible and self-explanatory. Reading the Rust source code and the included pseudo code side by side should allow for a good understanding of the algorithm.

5.0.1 Example of the execution with argument 'computation'

Let A be the original example and B the example with the increased state count. Let T be a set of trees which will be used to illustrate the successful minimisation. The following paragraph details the actions performed by the binary when called with the **–automaton computation** argument.

- **Read and parse the grammar file, construct two automata A and B.**
- **Print automaton A**:

  Automaton Type: BottomUp, Ranked Alphabet: {0: 0, +: 2, -: 2, x: 2, !: 2, 1: 0}, Final Weight Vector: $\begin{bmatrix}0 & 1\end{bmatrix}^T$, Transition Matrices:

$$\mu(1) = \begin{bmatrix}1 & 1\end{bmatrix} \mu(0) = \begin{bmatrix}0 & 1\end{bmatrix}$$
$$\mu(+) = \begin{bmatrix}1 & 0\\ 0 & 1\\ 0 & 1\\ 0 & 0\end{bmatrix} \mu(-) = \begin{bmatrix}1 & 0\\ 0 & -1\\ 0 & 1\\ 0 & 0\end{bmatrix} \mu(x) = \begin{bmatrix}1 & 0\\ 0 & 0\\ 0 & 0\\ 0 & 1\end{bmatrix} \tag{5.1}$$

- **Minimise A, create a new automaton $\tilde{A}$.**
- **Print automaton $\tilde{A}$**:

  Automaton Type: BottomUp, Ranked Alphabet: {0: 0, +: 2, -: 2, x: 2, !: 2, 1: 0}, Final Weight Vector: $\begin{bmatrix}0 & 1\end{bmatrix}^T$, Transition Matrices:

$$\mu(1) = \begin{bmatrix}0 & 1\end{bmatrix} \mu(0) = \begin{bmatrix}1 & 0\end{bmatrix}$$
$$\mu(+) = \begin{bmatrix}1 & 0\\ 0 & 1\\ -1 & 1\\ 0 & 2\end{bmatrix} \mu(-) = \begin{bmatrix}1 & 0\\ 2 & -1\\ 0 & 1\\ 1 & 0\end{bmatrix} \mu(x) = \begin{bmatrix}1 & 0\\ 1 & 0\\ 1 & 0\\ 0 & 1\end{bmatrix} \tag{5.2}$$

- **Print automaton B:**

  Automaton Type: BottomUp, Ranked Alphabet: {0: 0, +: 2, -: 2, x: 2, !: 4, 1: 0}, Transition Matrices:

$$\gamma = \begin{bmatrix}0\\1\end{bmatrix} \mu(1) = \begin{bmatrix}1 & 1 & 1 & 1\end{bmatrix} \mu(0) = \begin{bmatrix}1 & 0 & 1 & 0\end{bmatrix}$$

$$\mu(+) = \begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0\end{bmatrix}^T$$

$$\mu(-) = \begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&-1&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&-1&0&0&1&0\end{bmatrix}^T \tag{5.3}$$

$$\mu(x) = \begin{bmatrix}1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\end{bmatrix}^T$$

- **Minimise B, create a new automaton $\tilde{B}$.**
- **Print automaton $\tilde{B}$:**

Automaton Type: BottomUp, Ranked Alphabet: {0: 0, +: 2, -: 2, x: 2, !: 2, 1: 0}, Transition Matrices:

$$\gamma = \begin{bmatrix}0\\2\end{bmatrix} \mu(1) = \begin{bmatrix}0 & 1\end{bmatrix} \mu(0) = \begin{bmatrix}1 & 0\end{bmatrix}$$

$$\mu(+) = \begin{bmatrix}1&0\\0&1\\0&1\\-1&2\end{bmatrix} \mu(-) = \begin{bmatrix}1&0\\2&-1\\0&1\\1&0\end{bmatrix} \mu(x) = \begin{bmatrix}1&0\\1&0\\1&0\\0&1\end{bmatrix} \tag{5.4}$$

- **Compute the values $||A||(t)$, $||\tilde{A}||(t)$, $||B||(t)$, $||\tilde{B}||(t)$ for a number of trees $t \in T$ and print these values:**

Tree: +(x(1 1) +(1 1))
$||A||(t)$: 3.0 = 3.00
$||\tilde{A}||(t)$: 3.0 ≈ 3.00
$||B||(t)$: (3.0) x 2 = 6.00
$||\tilde{B}||(t)$: (3.0) x 2 ≈ 6.00

Tree: x(+(1 1) +(1 +(1 +(1 1))))
$||A||(t)$: 8.0 = 8.00
$||\tilde{A}||(t)$: 8.0 ≈ 8.00
$||B||(t)$: (8.0) x 2 = 16.00
$||\tilde{B}||(t)$: (8.0) x 2 ≈ 16.00

This result showcases the idempotence of the minimisation algorithm when applied to an MTA which is already minimal. As a result $||A|| = ||\tilde{A}||$.
It furthermore illustrates the equality between the original MTA A, the automaton B with its doubled states and doubled final weight, and the automaton $\tilde{B}$ with its minimal amount of states and its consistent final weights after the minimisation procedure.
Minimizing the other two automata and testing them on a number of additional trees can be done by executing the code itself.

5.0.2 Numerical precision

The final interesting result is an illustration of the problem most numeric implementations of any algorithm face. Each value in the matrices which are computed is represented by a 64-Bit floating point number. The issues with precision that arise when working with these numbers, especially during repeated computations, are well known.
Thus the values, both for the transition matrices as well as for the computed weights, which are given in the example above are only approximation of the integers which were used for the initialization of the non-minimised MTAs. For example a value of 0, might very well be saved as $-0.00000000000000001570092458683775$ internally.
This obviously would pose issues when using this code to deal with real world examples, since those often are not defined in terms of integers or simple fractions, but instead based on weights which are organically created by iterating over large data sets. That said, there are obviously a large number of ways to deal with this issue as it has been an area of research for a long time.

5.1 Possible future work

There are a number of potential ideas which could be build upon the existing implementation. Care was taken to make the code itself easily extendable, so most of these should be achievable without major refactoring.
The main practical improvement which immediately comes to mind is the extension of the minimisation to include some kind of merging of states which produce similar results or pruning of transitions having little influence on the result. Currently the algorithm computes a new MTA which is completely equivalent to the input automaton. Looking at real world examples quickly shows that organically generated automata or grammars often have transitions with

extremely low weights, or transitions which similar results etc.
To achieve a higher confidence in the results of the implementation a automated testing suite which generates trees and automata would obviously be another step. This would also allow useful benchmarks based on state-count, tree size, etc. The parser is already able to correctly read top-down automata, the next step would be to implement conversion of the top-down automata to a bottom-up automata.

Chapter 6

Conclusion

In this thesis I dealt with Multiplicity Tree Automata (MTA) based on the paper 'Minimisation of multiplicity tree automata' by Kiefer et al. [2]. In aforementioned paper this theoretical construct was defined and a minimisation algorithm for it was specified.
Based on this paper I first illustrate the concept of MTAs with a number of examples. Afterwards in the second part of the thesis I present the minimisation algorithm in great detail, accompanying it with an example run. The algorithm itself, which consists of three parts, is completely given in pseudo code, while in the original paper [2] only one part of the algorithm was made explicit.
The main achievement is the implementation of Multiplicity Tree Automata from scratch in the Rust language. It is possible to create MTAs in this framework, it is possible to use these MTAs to compute weights over trees and finally the minimisation algorithm itself is implemented completely. Instructions as to how the code can be executed are provided and with a few suitable examples the correctness of the code is illustrated.

Bibliography

[1] Jean Berstel and Christophe Reutenauer. Recognizable formal power series on trees. *Theoretical Computer Science*, 18(2):115–148, 1982.

[2] Stefan Kiefer, Ines Marusic, and James Worrell. Minimisation of multiplicity tree automata. In *International Conference on Foundations of Software Science and Computation Structures*, pages 297–311. Springer, 2015.

[3] Ines Marušic and James Worrell. Complexity of equivalence and learning for multiplicity tree automata. *Journal of Machine Learning Research*, 16:2465–2500, 2015.

[4] Bobbi Jo Broxson. The kronecker product. 2006.

[5] Amaury Habrard and Jose Oncina. Learning multiplicity tree automata. In *International Colloquium on Grammatical Inference*, pages 268–280. Springer, 2006.

[6] Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, 2006.

[7] Christian Pech. *Kleene type results for weighted tree automata.* PhD thesis, Dresden University of Technology, Germany, 2003.

[8] Jonathan May and Kevin Knight. Tiburon: A weighted tree automata toolkit. In *International Conference on Implementation and Application of Automata*, pages 102–113. Springer, 2006.

[9] Nils Klarlund. Mona & fido: The logic-automaton connection in practice. In *International Workshop on Computer Science Logic*, pages 311–326. Springer, 1997.

[10] Loek Cleophas and Kees Hemerik. Forest fire: A taxonomy-based toolkit of tree automata and regular tree algorithms. In *International Conference on Implementation and Application of Automata*, pages 245–248. Springer, 2009.

[11] Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 695–706. Springer, 2001.

[12] Ondřej Lengál. An efficient finite tree automata library. *arXiv preprint arXiv:1204.3240*, 2012.

[13] Thomas Genet. Completeness of tree automata completion. In *FSCD 2018-3rd International Conference on Formal Structures for Computation and Deduction*, pages 1–20, 2018.

[14] Forestfire homepage. `http://www.fastar.org`. Accessed: 2023-11-27.

[15] nalgebra homepage. `https://web.archive.org/web/20231115134426/https://www.nalgebra.org/`. Accessed: 2023-11-27.

[16] trees package documentation. `https://web.archive.org/web/20220626230601/https://docs.rs/crate/trees/latest`. Accessed: 2023-11-27.

[17] Symeon Bozapalidis and Olympia Louscou-Bozapalidou. The rank of a formal tree power series. *Theoretical Computer Science*, 27(1-2):211–215, 1983.

[18] David Poole. *Linear algebra: A modern introduction*. Nelson Education, 2014.

[19] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[20] Wen-Guey Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21(2):216–227, 1992.

[21] Symeon Bozapalidis and Athanasios Alexandrakis. Représentations matricielles des séries d'arbre reconnaissables. *RAIRO-Theoretical Informatics and Applications*, 23(4):449–459, 1989.

[22] Peng Wang, Shaoshuai Mou, Jianming Lian, and Wei Ren. Solving a system of linear equations: From centralized to distributed algorithms. *Annual Reviews in Control*, 47:306–322, 2019.

Chapter 7

Attachments

7.1 Source code and binary

The source code and a compiled binary are attached on a physical drive to the physical copies of this thesis.
Furthermore the source code and a compiled binary are distributed together with the digital copy of this thesis.

„Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.“

Ort:

Datum:

Unterschrift: