Implementing restricted hidden Markov model algorithms for gene prediction
Daniel Lysdal Kjeldsen, 20042273
Jakob Hundal Sandholm, 20021868

Master’s Thesis, Computer Science
October 2012
Advisor: Christian Nørgaard Storm Pedersen
In this thesis, we examine a recently developed method, which extends the forward and Viterbi algorithm for hidden Markov models.

Genes are the parts of DNA that are used in the synthesis of proteins, and hidden Markov models are an important tool in the analysis of biological sequences such as DNA. We examine the restricted hidden Markov model algorithms in the context of predicting genes in bacterial DNA sequences.

First, we make an independent implementation of the restricted algorithms and find that the restricted Viterbi algorithm performs better than the traditional Viterbi. Second, we find the time and space consumption of the restricted algorithms to be a limiting factor in gene prediction. Therefore, we describe and apply different improvements to reduce the time and space consumption of the restricted algorithms. By these improvements, we manage to reduce the time and space consumption significantly. Third, we describe an extension of hidden Markov models that allows us to reduce the number of hidden states when building models for gene prediction, and we adapt the improved restricted algorithms to this formalism.
Contents

Abstract iii

1 Introduction 3
  1.1 Our contribution .....................................  3
  1.2 Thesis structure .....................................  4

2 Background 5
  2.1 Molecular biological background .........................  5
  2.2 Hidden Markov models ................................  7
    2.2.1 Definitions ......................................  7
    2.2.2 The forward algorithm .............................  9
    2.2.3 The Viterbi algorithm ............................. 11
    2.2.4 Model selection .................................. 12
  2.3 Finite automata ..................................... 13

3 Methods 17
  3.1 The restricted algorithms .............................. 17
    3.1.1 The restricted forward algorithm .................. 18
    3.1.2 The restricted Viterbi algorithm ................. 28
    3.1.3 Improvement of the time consumption ............. 35
  3.2 Space improvements for the restricted Viterbi algorithm . 36
    3.2.1 The checkpoint algorithm ........................ 36
    3.2.2 Approximate linear space ........................ 37
  3.3 Improving the restricted forward parameter table size .... 45
    3.3.1 Theoretical considerations ........................ 46
    3.3.2 The cut off method ................................ 47
  3.4 Multiple emission hidden Markov models ................ 54
    3.4.1 The forward and Viterbi algorithms .............. 56
    3.4.2 The restricted forward algorithm ................. 58
    3.4.3 The restricted Viterbi algorithm ................. 62

4 Experiments 71
  4.1 Accuracy of the restricted Viterbi algorithm ............ 71
  4.2 Time complexity of the restricted algorithms ............ 75
    4.2.1 The restricted forward algorithm ................. 75
    4.2.2 The restricted Viterbi algorithm ................. 80
  4.3 Improvement of the time consumption ........................ 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4 Space improvement for the restricted Viterbi algorithm</td>
<td>82</td>
</tr>
<tr>
<td>4.5 The cut off method for the restricted forward algorithm</td>
<td>85</td>
</tr>
<tr>
<td>4.6 MEHMM time and space complexity</td>
<td>89</td>
</tr>
<tr>
<td>4.7 Summary of experimental results</td>
<td>96</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>97</td>
</tr>
<tr>
<td>5.1 Future work</td>
<td>97</td>
</tr>
<tr>
<td>Bibliography</td>
<td>100</td>
</tr>
<tr>
<td>A Occurrence distribution examples</td>
<td>101</td>
</tr>
<tr>
<td>B Time complexity of the restricted Viterbi algorithm</td>
<td>103</td>
</tr>
<tr>
<td>C Division of work</td>
<td>107</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

During the last few decades, *hidden Markov models* have become an increasingly important tool in the analysis of biological sequences. Algorithms based on hidden Markov models are now used to solve a variety of problems, including *gene prediction* [15], *multiple alignment* [7] and *protein structure prediction* [12]. Applications of hidden Markov models also include problems outside of bioinformatics such as *speech recognition* [18] and *image processing* [14].

As the employment of hidden Markov models expands, so does the requirements of the algorithms based on these models. Improving and adapting hidden Markov model algorithms for different applications is a continuing field of research.

An example hereof is the basic *Viterbi algorithm*, for which several alternatives have been developed. The most recent of these alternatives is the *restricted Viterbi algorithm* [20], which are claimed to be more accurate than the basic Viterbi algorithm at the cost of increased resource consumption.

The method is currently under development at Aarhus University, and thus there is interest in verifying the initial results. In this thesis, we perform an independent implementation of the algorithms with the intent of verifying their expected running time and accuracy in the context of a simple gene prediction model. Because the time and space consumption pose a major challenge to the usefulness of the algorithms in a gene prediction context, we furthermore examine, implement and evaluate ways of decreasing the time and space consumption.

1.1 Our contribution

We introduce relevant background material in the context of our thesis. We present and implement the contents of a new research article [20] introducing the restricted algorithms, which are central to the thesis. Subsequently, we combine the article with research results improving the space consumption for the basic Viterbi algorithm and show that similar improvements are possible for the restricted Viterbi algorithm. Furthermore, we develop and implement a reduction of the size of the algorithms’ main table by approximating the result. Additionally, we describe and implement a more general model type and adjust
the presented algorithms to fit this model type. Finally, we select a series of relevant experiments in order to verify the postulated properties and to examine the usefulness of the improvements.

1.2 Thesis structure

In chapter 2, we explain the background material needed to put our thesis in context and to understand the methods (chapter 3).

We continue in chapter 3 with the methods: the restricted algorithms, some time and space improvements hereof, and an extension of the HMM formalism that is relevant to gene prediction.

Afterwards, in chapter 4, we perform a collection of experiments to verify the claims of chapter 3, such as measures of accuracy, time and space complexity, and effectiveness of the improvements.

Finally, in chapter 5, we conclude on our results and consider relevant future work.
Chapter 2

Background

In this chapter, we introduce the relevant background material needed to put the rest of the thesis in context and to understand the formalisms that our methods build upon.

2.1 Molecular biological background

During the presentation and development of algorithms in this thesis, gene prediction is consistently our application of choice for illustrating practical points and considerations. In order to establish area terminology and show the relevance of the algorithms, we will begin by describing the molecular background of gene prediction in bacteria. The content of this section is based on [10 chapter 11].

Encoded in an organism’s DNA is the blueprint of the entire organism. The sequential data of the DNA, which consists of the four bases adenine (A), cytosine (C), guanine (G) and thymine (T), is continuously used as template for protein synthesis. This process is split into transcription from DNA to RNA and translation from RNA to proteins, cf. the central dogma of molecular biology as depicted in figure 2.1.

\[\text{DNA} \quad \begin{array}{c}
\text{A} \quad \text{C} \quad \text{A} \\
\text{G} \quad \text{A} \\
\text{T} \quad \text{T} \quad \text{A} \\
\text{T} \quad \text{C} \quad \text{C} \quad \text{A}
\end{array} \]

\[\text{RNA} \quad \begin{array}{c}
\text{A} \quad \text{C} \\
\text{A} \\
\text{G} \\
\text{A} \\
\text{U} \\
\text{U} \\
\text{A} \\
\text{U} \\
\text{C} \\
\text{C} \\
\text{A}
\end{array} \]

\[\text{Protein} \quad \begin{array}{c}
\text{Thr} \\
\text{Asp} \\
\text{Tyr} \\
\text{Pro}
\end{array} \]

Figure 2.1: The central dogma of molecular biology

The transcription consists in copying one of the DNA’s two strands while exchanging thymine with uracil (U), and the translation consists in mapping
groups of three RNA bases to one amino acid, complying with the genetic code as shown in figure 2.2.

Three subsequent bases in DNA or RNA are called a codon. Thus, the genetic code indirectly maps codons in the DNA to amino acids in proteins. An important decision in the protein synthesis process is where to depict each codon. Consider part of a DNA strand, CGGTACTAAG. This can be interpreted and split into codons as either CGG TAC TAA G, CG GTA CTA AG or CGGT ACT AAG, differing in the choice of reading frame. Obviously, different reading frames will lead to very different proteins.

The full genome of an organism is subdivided into a large number of genes, each encoding one protein. These genes may overlap, be positioned on either DNA strand and in any of the three reading frames. Which subsequences of the DNA will be transcribed and translated depends on many factors. For instance, a gene must start with a start codon, end with a stop codon and must be preceded by so-called promoter sequences required for initiation of transcription. Additionally, the cell contains regulatory machinery for controlling the transcription and translation extent of specific genes.

Localizing the genes of a genome is not a simple task. Experiments may
reveal information about gene positions, but based on the recent progress in sequencing genomes, methods for predicting genes based on the DNA strands only are desirable. Such methods are called \textit{ab initio} gene prediction methods and encompass the main practical applications considered in this thesis.

\section{2.2 Hidden Markov models}

\subsection{2.2.1 Definitions}

Consider a sequence $X$ of observations, each chosen from a finite set $\mathcal{O}$ of observables. Such a sequence is called a \textit{Markov chain} if it has the \textit{Markov property}, that is, if every element of the sequence is independent of all but the previous element.\footnote{The theory of Markov chains can be extended to consider chains of different order \cite[pp 605-610]{pp}, however this is of little relevance in the current context.} A simple Markov chain is shown in figure \ref{fig:markov-chain}.

![A simple Markov chain](image)

\textbf{Figure 2.3: A simple Markov chain}

Because of the Markov property, modelling a Markov chain corresponds to specifying transition probabilities between each pair of observables. Each step in the observation sequence then corresponds to a model transition between the two observables. A \textit{Markov model} of the chain in figure \ref{fig:markov-chain} can be seen in figure \ref{fig:markov-model}.

![A simple Markov model](image)

\textbf{Figure 2.4: A simple Markov model}

Fundamentally, a Markov model can be viewed in two different ways: as an attempted explanation of an observed sequence of discrete random variables, or as a generator of such sequences.

A \textit{hidden Markov model} (or \textit{HMM}) \cite{HMM} extends the ideas of the simple Markov model. Instead of having a one-to-one correspondence between states in the model and output symbols, hidden Markov models introduce the idea of \textit{hidden states} and \textit{emission probabilities}. As seen in figure \ref{fig:hidden-markov-model}, we here think of a sequence of output variables as a result of a Markov chain of hidden variables.
Subsequent output symbols are still separated by state transitions in the model, but instead of specifying one element from the output alphabet, each state of the model entails a probability distribution (the emission probabilities) over the output alphabet. The output symbol is then chosen according to this distribution. An example of such a model can be seen in figure 2.6.

Figure 2.6: A hidden Markov model. Rectangles represent hidden states, emission probabilities are written inside the rectangles. Arrows with numbers represent transition probabilities.

Given a hidden Markov model and an output sequence generated by the model there is generally no way of determining the corresponding sequence of hidden states with certainty - hence the name “hidden states”.

We will primarily think of an HMM as a modelled explanation of observable sequences. If we have trust in the model, it can then be used predictively to annotate an observable sequence, that is explain the observable sequence by a hidden sequence. Additionally, we will be using hidden Markov models for generating such sequence pairs.

Formally, we define a hidden Markov model as a quintuple $\lambda = (H, O, \Pi, A, B)$, where

- $H$ is a finite set of hidden states $h_1, h_2, \ldots, h_N$
- $O$ is a finite alphabet of output symbols $o_1, o_2, \ldots, o_M$
- $\Pi$ is a vector $(\pi_1, \pi_2, \ldots, \pi_N)$ where $\pi_i$ is the probability of the hidden sequence starting in state $h_i$
- $A$ is a $N \times N$ matrix where each entry $a_{i,j}$ is the probability of a state transition from $h_i$ to $h_j$
- $B$ is a $N \times M$ matrix where each entry $b_{i,j}$ is the probability of emitting output symbol $o_j$ from the state $h_i$

Note that we have no designated stop states, thus the HMM generates infinite sequences by default. We allow finite sequences simply by stopping the HMM when it has generated a sequence of some given length. As for sequence
terminology, we will denote an observed sequence $X = x_1 x_2 \ldots x_T$, where $x_t \in O$ for $t = 1, 2, \cdots, T$, and a sequence of hidden states $Z = z_1 z_2 \ldots z_T$, where $z_t \in H$ for $t = 1, 2, \cdots, T$. Partial sequences will be designated as $X_{t:t'} = x_t \cdots x'_t$ and $Z_{t:t'} = z_t \cdots z'_t$, respectively.

Three problems arise when we want to apply an HMM to real observable sequences \[15\]. First, how do we efficiently compute $P(X|\lambda)$, the probability of a particular observable sequence, given the model? As a simple application of this question, consider the challenge of deciding which of several HMMs fit an observed sequence better. Second, given an observable sequence, how do we find an annotation that is optimal, i.e. best explains the observable sequence? Answering this question corresponds to explaining the observed sequence with basis in the model, or in other words, finding its best annotation. As we shall see, this problem will be of great importance in this thesis. Third, how do we adjust the model parameters ($\Pi$, $A$ and $B$) such that $P(X|\lambda)$ is maximized? Adjusting the parameters based on observations is an important prerequisite for giving the probabilities mentioned in the first two questions statistical reliability.

We refer to these problems as the three basic problems for HMMs. In the next section, we will describe well established algorithms that solve the first two of these problems. The algorithms are highly relevant, because our methods extend these basic algorithms.

The last problem, also referred to as training, can be split into supervised training and unsupervised training. An example of supervised training is training by counting \[6, pp 62-63\], which can be used when hidden state sequence annotations of the observable sequences are available. If no annotations are available, unsupervised training must be used. An example of unsupervised training is the Baum-Welch algorithm \[2\]. In this thesis, we will not be particularly concerned with adjusting the model parameters in order to maximize the statistical reliability of the results. However, for experiments on real DNA sequences, training is naturally an important subject.

### 2.2.2 The forward algorithm

In order to solve the first of the three basic problems, we first want to obtain an expression of the probability of a particular observable sequence $X$ given a HMM. In the following, we will assume some fixed HMM $\lambda$ and therefore refer to this probability as just $P(X)$.

It is easily seen that given a hidden state sequence $Z$, the conditional probability $P(X|Z)$ can be calculated as

$$P(X|Z) = b_{z_1,x_1} b_{z_2,x_2} \cdots b_{z_T,x_T}$$

since the hidden state sequence $Z$ emits the exact sequence $X$ if and only if $z_t$ emits $x_t$ for $1 \leq t \leq T$. The probability of the hidden state sequence itself is given by

$$P(Z) = \pi_{z_1} a_{z_1,z_2} a_{z_2,z_3} \cdots a_{z_{T-1},z_T}$$
since it is equal to the probability of making the independent choices of the correct starting state and the correct transitions throughout the sequence. The joint probability of $X$ and $Z$ is then, cf. the definition of conditional probability,

$$
P(X, Z) = P(X|Z)P(Z) = \pi_{z_1} b_{z_1, x_1} \prod_{t=2}^{T} a_{z_{t-1}, z_t} b_{z_t, x_t}
$$

which is not surprising, since it corresponds to starting in $z_1$, emitting $x_1$ from $z_1$, transitioning from $z_1$ to $z_2$, emitting $x_2$ from $z_2$ and so on. The probability of the observable sequence $X$ can now be expressed by summing over all the possible state sequences $Z$:

$$
P(X) = \sum_Z P(X, Z)
$$

A naïve and obvious approach to evaluating this expression is to directly calculate each term in the sum. Unfortunately, since there are $O(N^T)$ different possible hidden state sequences of length $T$, such a calculation would take exponential time.

The forward algorithm [18] presents an elegant and efficient way of solving the problem. By using dynamic programming, it addresses increasingly growing partial instances of the problem one by one. During this process an $N \times T$ table is filled with values of a forward parameter $\alpha$, defined by:

$$
\alpha_t(z_t) = \mathbb{P}(X_{1:t}, z_t)
$$

In other words, the forward parameter denominates the joint probability of emitting the first $t$ symbols of $X$ and being in a specific hidden state $z_t$ when emitting the last of these symbols. This probability can be split into individual probabilities of emitting the first $t-1$ symbols of $X$ and being in state $z_{t-1}$ for all possible values of $z_{t-1}$ and then at time $t$ making the transition into $z_t$ and emitting $x_t$:

$$
\alpha_t(z_t) = b_{z_t, x_t} \sum_{z_{t-1}} \alpha_{t-1}(z_{t-1}) a_{z_{t-1}, z_t}
$$

In other words, in order to calculate the forward parameter for an entry, we can make use of the previous column of the table, that is, the $N$ values of $\alpha_{t-1}(z_{t-1})$.

The dependency of entry results leads us to considerations regarding the order in which the table should be calculated. In graph theory, a topological ordering is an ordering of the vertices of a directed acyclic graph (DAG) such that if the graph contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering [5, p. 549]. We can model the dependencies of the table as a DAG where we define a vertex $v$ for each table entry, and an edge $(u, v)$ whenever an entry $v$ depends directly on an entry $u$. Now, any topological ordering of this graph is a valid order in which the table entries can be filled. From now on, we also use the term topological ordering on our dynamic programming tables to denote a valid order in which the table entries can be filled. It can be seen from equation 2.1
that a valid order for the forward table is filling the entries for increasing values of \( t \). The specific ordering of cells with equal \( t \)-value is not important here, since none of these cells depend on each other. The usage of a topological ordering based on the \( t \)-value is common to all algorithms presented in this thesis, with the exception of section 3.3.2.

Returning to the forward table, the only missing part is how to fill out the first column of the table, however, each entry \( \alpha_1(z_1) \) is easily seen to be the joint probability of starting in state \( z_1 \) and emitting \( x_1 \). Furthermore, we can sum out the hidden state for any \( t \)-value, and naturally get the probability \( P(X_{1:t}) \). From these considerations the forward algorithm emerges, as defined in the following:

**Base case**

\[
\alpha_1(z_1) = \pi_{z_1} b_{z_1, x_1}
\]

**Recursion case**

\[
\alpha_t(z_t) = b_{z_t, x_t} \sum_{z_{t-1}} \alpha_{t-1}(z_{t-1}) a_{z_{t-1}, z_t}
\]

**Termination**

\[
P(X_{1:T}) = \sum_{z_T} \alpha_T(z_T)
\]

Now, consider the time and space efficiency of the forward algorithm. The initialization step takes \( O(N) \) time, the recursion step takes \( O(N^2 \cdot T) \) time (\( O(N) \) time for each of \( O(N \cdot T) \) cells) and the termination step takes \( O(N) \) time. In total, the forward algorithm therefore has a time complexity of \( O(N^2 \cdot T) \) and, for storing the table, a space consumption of \( O(N \cdot T) \). Actually, the space consumption can easily be brought down to \( O(N) \) by only keeping the current and previous column in memory. However, in some applications, e.g. posterior decoding, we want to be able to access all the entries of the table, and in such cases, we do not have the option of deallocating previous columns.

### 2.2.3 The Viterbi algorithm

In order to solve the second of the three basic problems, we first need to select an optimality criterion. A common criterion is to maximize the total probability of the hidden state sequence given the observable sequence. This criterion is what is used in the Viterbi algorithm [18]. Examples of other optimality criterions will be discussed in section 3.1.

Similar to the forward algorithm, the Viterbi algorithm defines a parameter, \( \omega \). The Viterbi parameter is defined by

\[
\omega_t(z_t) = \max_{Z_{1:t-1}} P(X_{1:t}, Z_{1:t})
\]
\( \omega_t(z_t) \) denominates the probability of the most likely sequence of states \( Z_{1:t} \) ending in state \( z_t \) and outputting the observations \( X_{1:t} \). This probability naturally depends on the probabilities of the most likely sequence of states \( Z_{1:t-1} \) ending in state \( z_{t-1} \) and outputting the observations \( X_{1:t-1} \) for all \( z_{t-1} \). By multiplying each of these probabilities with the probabilities of transitioning from \( z_{t-1} \) to \( z_t \) and emitting \( x_t \) from \( z_t \), we get a number of possible values of \( \omega_t(z_t) \), each depending on the previous state. It is then clear that, similarly to the forward algorithm, we can use a recursive formulation to calculate the maximum. The initial values \( \omega_1(z_1) \) are again equal to the joint probability of starting in state \( z_1 \) and emitting \( x_1 \). Like in the forward algorithm, we can use dynamic programming to fill out an \( N \times T \) table using the same topological ordering, i.e. calculating entries column-wise for increasing values of \( t \). Once the table has been computed, the most likely state sequence \( Z^* \) can be found by backtracking from \( \arg \max_{z_T} \omega_T(z_T) \). The resulting algorithm is shown below:

**Base case**

\[
\omega_1(z_1) = \pi_{z_1} b_{z_1, x_1}
\]

**Recursion case**

\[
\omega_t(z_t) = b_{z_t, x_t} \max_{z_{t-1}} \omega_{t-1}(z_{t-1}) a_{z_{t-1}, z_t}
\]

**Backtracking base case**

\[
z_T^* = \arg \max_{z_T} \omega_T(z_T)
\]

**Backtracking recursion case**

\[
z_t^* = \arg \max_{z_t} (\omega_t(z_t) a_{z_t, z_{t+1}}) \text{ for } t = T-1, \ldots, 1
\]

The backtracking takes \( O(N \cdot T) \) time, since we backtrack \( T \) steps and use \( O(N) \) time at each step. The time and space complexity of the Viterbi algorithm is the same as the forward algorithm, namely a time complexity of \( O(N^2 \cdot T) \), and a space consumption of \( O(N \cdot T) \).

### 2.2.4 Model selection

Model selection is about selecting the set of hidden states in the model together with the transition and emission probabilities that should be fixed, i.e. not influenced by training. While schemes for automatic model selection exist, in gene prediction, model selection is mostly about applying expert knowledge to design a model that achieves the best practical results.

Model selection is a balance between precision and performance. Designing a complex model may capture the problem domain in a more precise way, opening up the possibility of higher precision predictions. However, the time and space consumption rises with the number of hidden states in the model, so selecting the right model for a given application is a weighing of the requirements on
precision and performance. A very simple model designed for gene prediction is depicted in figure 2.7.

![Figure 2.7: A simple HMM for gene prediction. Rectangles represent hidden states and nucleotide emission probabilities are written inside the rectangles. Transition probabilities with value > 0 are represented by arrows with numbers.](image)

Non-coding regions are represented by the state \( N \), while forward and reverse coding regions are represented by states \( C_1, C_2, C_3 \) and \( R_1, R_2, R_3 \) respectively. We capture coding regions with multiple hidden states to be able to differentiate the occurrence of nucleotides in the different positions of the codons.

In this thesis, we will not focus on model selection. Instead, we use this simple gene prediction model as a basis for investigating algorithms related to hidden Markov models and gene prediction.

### 2.3 Finite automata

In this section, we explain the basics of regularity and finite automata. This is relevant because our methods use finite automata in the extension of the basic forward and Viterbi algorithms.

**Regular languages** A language is a set of strings involving symbols from some alphabet. A string over an alphabet \( \Sigma \) is obtained by placing some of the elements of \( \Sigma \) (possibly none) in order. The *null string* (the string of length 0) is a string over \( \Sigma \), no matter what alphabet \( \Sigma \) is. We denote it by \( \Lambda \). For any alphabet \( \Sigma \), the set of all strings over \( \Sigma \) is denoted by \( \Sigma^* \), so a language over \( \Sigma \) is therefore a subset of \( \Sigma^* \). Consider the empty language \( \emptyset \), the language \( \Lambda \) and the simple languages of the form \( a \), where \( a \in \Sigma \). A regular language [16, p. 85] over an alphabet \( \Sigma \) can be obtained from these basic languages using concatenation, union and Kleene * where Kleene * is defined as

\[
L^* = \bigcup_{i=0}^{\infty} L^i
\]

and \( L^k \) is the set of strings that can be obtained by concatenating \( k \) elements of \( L \).
Informally, a *regular expression* is a simplified way of describing a regular language using a convenient syntax and is used in great extent when describing regular languages. A formal definition can be found in [16] pp. 85-87. Note that the syntax for denoting alteration (corresponding to language union) varies, and we have chosen the commonly used “|” in this thesis.

**Deterministic finite automata (DFAs)** A central problem related to regular languages is the problem of **recognizing** a language, i.e. deciding whether a given string is in the language or not. This problem can be decided by an abstract machine called a *finite automaton* (FA). A deterministic FA (DFA) is a quintuple \((Q, \Sigma, q_0, A, \delta)\), where

- \(Q\) is a finite set of states
- \(\Sigma\) is the alphabet, a finite set of input symbols
- \(q_0 \in Q\) is an initial state
- \(A \subseteq Q\) is a set of accepting states
- \(\delta : Q \times \Sigma \rightarrow Q\) is the transition function

Intuitively, the transition function tells us which state to move to if we are in a given state, reading a given input symbol. A more useful function would tell us what state we end up in, if we are in a given state, reading a given input string. We can define the extended transition function \(\delta^* : Q \times \Sigma^* \rightarrow Q\) recursively using the definition of the transition function as follows

\[
\forall q \in Q, y \in \Sigma^*, a \in \Sigma : \delta^*(q, ya) = \delta(\delta^*(q, y), a)
\]

\[
\forall q \in Q : \delta^*(q, \Lambda) = q
\]

If we apply this definition recursively on a string, we evaluate each symbol, one at a time, using the transition function. We now have the foundations covered to define when a DFA **accepts** an input string and when it **rejects** it.

A DFA accepts a string \(x \in \Sigma^*\) if \(\delta^*(q_0, x) \in A\), i.e. if we start in the initial state and make transitions according to the transition function and end up in an accepting state. Otherwise we say that it **rejects** the string. The language

![Figure 2.8: A DFA accepting the language \(L = \{a, c, g, t\}^*\{tta\}\). States are depicted as circles and transitions as arcs with transition symbols. Accept states are marked with a double circle.](image-url)
accepted by the DFA is the set of strings which the DFA accepts. In figure 2.8, a sample DFA is depicted.

According to Kleene’s theorem, for any regular language, we can construct a DFA that recognizes the language [16, pp. 146-148]. This theorem is important to our thesis, since our methods need to construct a DFA for any given regular expression input (see section 3.1).
Chapter 3

Methods

In the previous chapter, we have introduced finite automata, hidden Markov models and algorithms solving some of the basic HMM problems. These subjects are combined by the more advanced and recently developed restricted algorithms [20].

In this chapter, we will take a closer look at the restricted algorithms and different approaches for improving a naïve implementation. Each section contains a theoretical introduction to the method and a practical discussion of implementation choices. Furthermore, we extend the hidden Markov model formalism to allow hidden states to emit more than one symbol, and adapt the algorithms to this extended model.

3.1 The restricted algorithms

The Viterbi algorithm, as earlier presented, predicts the single most likely hidden state sequence. In other words, its optimality criterion is to maximize the total probability of a hidden sequence, given the observables. The Viterbi algorithm generally performs well when this sequence, according to our model, is much more probable than other hidden sequences. However, it becomes relatively less accurate in the case of several candidate sequences of similar probability. To improve the accuracy of the prediction in such cases, other optimality criteria can be used. Several alternatives to the Viterbi algorithm have been developed, each with different criteria and therefore different applicability, given the specific problem.

A few important alternatives are 1-best [13], posterior decoding [18] and posterior Viterbi [8]. 1-best has been shown to perform equally to or better than Viterbi and with similar advantages, i.e. excelling in cases with a single hidden sequence much more probable than the other candidates. Posterior decoding predicts a hidden sequence where the state probability at each position in the sequence is maximized. This has advantages especially when several hidden state sequences are all close to having the highest probability, but a drawback is that it is not restricted to hidden sequences that are legal according to the model. Posterior Viterbi combines the results of posterior decoding with the Viterbi algorithm to find the hidden sequence which has the largest product of
individual state probabilities, while being legal.

The restricted Viterbi algorithm \cite{20} is the most recent alternative to the Viterbi algorithm. It combines the predictive possibilities of hidden Markov models with the pattern recognition of finite automata. Hereby, the algorithm is able to restrict its possible pool of explanations to sequences meeting certain requirements on the number of occurrences of some pattern, hence the name of the algorithm. Furthermore, a related algorithm, the restricted forward algorithm, can be used to compute a probability distribution over the number of pattern occurrences in the hidden state sequence. This opens up the possibility of targeting the restricted Viterbi algorithm towards a reasonable occurrence interval based on the distribution. Thus, the restricted algorithms are based on an optimality criterion of maximizing the probability of a hidden sequence containing an acceptable occurrence count of the regular expression. In the context of gene prediction, this regular expression could correspond to e.g. gene start and stop, thus adjusting the algorithm to only return hidden sequences containing an acceptable number of genes.

Thus, instead of using Viterbi for determining the most likely of all hidden sequences, we can use the restricted Viterbi for determining the most likely of a set of hidden sequences of particular relevance to us. In following sections, we will sometimes refer to the forward and Viterbi algorithms as the basic algorithms in order to clearly distinguish them from the restricted algorithms.

Our contribution in this section is to present the restricted algorithms in a gene prediction context and to ensure numerical stability of the algorithms. Furthermore, we develop pseudocode, which enables us to implement the algorithms. Additionally, we make an improvement regarding the time consumption of the algorithms, specific to the use of our external library for automata \cite{17}.

### 3.1.1 The restricted forward algorithm

**Theory**

Based on the introduction above, a relevant problem is to calculate the probability distribution of $O_r(Z)$, the number of occurrences of $r$ in $Z$. A way to solve this is by using the forward algorithm to run an FA simultaneously with our HMM, giving us information about the current number of occurrences. We want an FA able to recognize $r$ at any position in the hidden sequence, or, in other words, recognize any string of hidden states with $r$ as a suffix. Formally we can write such a string as $(h_1|h_2|...|h_N)^*(r)$, where $H = h_1, h_2, ..., h_N$, and we denote a DFA recognizing this string $\text{FA}_H(r)$.

We define the restricted forward parameter $\widehat{\alpha}$ as follows:

$$\widehat{\alpha}_t(z_t, k, q) = \sum_{Z_1:t-1:O_r(Z_1:t)=k} \mathbb{P}(X_{1:t}, Z_{1:t}, \text{FA}_H(r)_t = q)$$

where $\text{FA}_H(r)_t$ is the state that $\text{FA}_H(r)$ is in at time $t$. The table entries are the probabilities of having observed $X_{1:t}$, being in hidden state $z_t$ and automaton state $q$ at time $t$ and having seen $k$ occurrences of the pattern, corresponding to having visited accepting states $k$ times.
Thus, in order to keep track of the DFA and the number of occurrences, we add 2 dimensions to the forward table. The size of the k-dimension, m, is equal to the maximum possible number of pattern occurrences. The size of the q-dimension equals the number of states in FA_H(r).

A 4-dimensional table can not be visualized in the same way that a 2- or 3-dimensional table can. However, in order to be able to describe the structure and different entries of our 4-dimensional tables, we would like a common way of thinking of such a table. We will be thinking of our 4-dimensional table as a series of 3-dimensional tables, or t-cubes, in each of which all entries have the same t-value.

When calculating a given entry \( \tilde{\alpha}_t(z_t, k, q) \) in the table at time \( t \), we should not sum over all entries at time \( t - 1 \) as in the case of the basic forward algorithm. Let \( \tilde{\alpha}_{t-1}(z_{t-1}, k_{\text{prev}}, q_{\text{prev}}) \) be an entry at time \( t - 1 \). It is easily seen that such an entry contributes to our entry at time \( t \) if and only if it fulfills two requirements. First, there must be a transition from \( q_{\text{prev}} \) to \( q \) in FA_H(r) for it to be a valid FA step. Second, the number of occurrences must be coherent; if \( q \) is an accepting state, \( k \) must equal \( k_{\text{prev}} + 1 \), and if \( q \) is not an accepting state, \( k \) must equal \( k_{\text{prev}} \). We then have:

\[
\tilde{\alpha}_t(z_t, k, q) = \begin{cases} 
\sum_{z_{t-1}: O_z(z_{t-1})=k} \mathbb{P}(X_1:t, Z_1:t, \text{FA}_H(r)_t = q) & \text{if } q \in A \\
\sum_{z_{t-1}: O_z(z_{t-1})=k-1} \mathbb{P}(X_1:t, Z_1:t, \text{FA}_H(r)_t = q) & \text{if } q \notin A 
\end{cases}
\]

Let \( Q'(z_t, q) \) be the set of \( q \)'s predecessors, that is \( \{q' | \delta(q', z_t) = q\} \). We can then rewrite \( \mathbb{P}(X_1:t, Z_1:t, \text{FA}_H(r)_t = q) \) as

\[
\mathbb{P}(X_1:t, Z_1:t, \text{FA}_H(r)_t = q) = \sum_{q' \in Q'(z_t, q)} \mathbb{P}(X_1:t, Z_1:t, \text{FA}_H(r)_{t-1} = q')
\]

Combining these two equations, the table can be computed recursively using base case

\[
\tilde{\alpha}_1(z_1, k, q) = \pi_{z_1} b_{z_1,x_1} \mathbb{1}(q_0 \in Q'(z_1, q)) \cdot \begin{cases} 
\mathbb{1}(q \notin A) & \text{if } k = 0 \\
\mathbb{1}(q \in A) & \text{if } k = 1 \\
0 & \text{otherwise}
\end{cases}
\]

and recursion case

\[
\tilde{\alpha}_t(z_t, k, q) = \begin{cases} 
b_{z_t,x_t} \sum_{z_{t-1}} a_{z_{t-1}, z_t} \sum_{q' \in Q'(z_t, q)} \tilde{\alpha}_{t-1}(z_{t-1}, k, q') & \text{if } q \in A \land k > 0 \\
b_{z_t,x_t} \sum_{z_{t-1}} a_{z_{t-1}, z_t} \sum_{q' \in Q'(z_t, q)} \tilde{\alpha}_{t-1}(z_{t-1}, k-1, q') & \text{if } q \in A \land k = 0 \\
0 & \text{if } q \notin A
\end{cases}
\]
Using the equations above, the algorithm fills out the $\tilde{\alpha}_t$-table. After it is filled out, an entry in the final $t$-cube, i.e. where $t = T$, contains the probability of observing the given string while ending in a certain hidden state $z_T$, a certain automaton state $q$ and having encountered $k$ pattern occurrences along the way. We can get our occurrence probability distribution from calculating the sums of final $t$-cube entries with equal $k$, since:

$$P(O_r(Z_{1:T}) = k|X_{1:T}) = \frac{P(O_r(Z_{1:T}) = k, X_{1:T})}{P(X_{1:T})} = \frac{1}{P(X_{1:T})} \sum_{z_T, q} \tilde{\alpha}_T(z_T, k, q)$$

where $P(X_{1:T})$ can be calculated as the sum of all entries $\tilde{\alpha}_T(z_T, k, q)$ of the final $t$-cube.

The $\tilde{\alpha}$ table has $T \cdot N \cdot m \cdot |Q|$ entries. For every entry, we sum over all previous hidden states and possible predecessors in the DFA and so it takes $O(N \cdot |Q|)$ to compute each entry. This leads to a time complexity of $O(T \cdot N^2 \cdot m \cdot |Q|^2)$ and a space complexity of $O(T \cdot N \cdot m \cdot |Q|)$.

**Implementation**

In this section, we explain the details required to enable us to implement the algorithm, including how to address problems regarding numerical stability and simple ways of improving the space consumption and the size of the $k$-dimension of the restricted forward table. Furthermore, we will present pseudocode for the complete algorithm.

**Numerical stability** When implementing the restricted forward algorithm, an important challenge is obtaining numerical stability. The sum of entry values of each $t$-cube decreases as $t$ increases, since the likelihood of observing any sequence decreases when increasing its length by appending to it. Hence, as our algorithm increments $t$, the probabilities become smaller and smaller, and we risk loss of numerical stability even for fairly short sequences.$^1$

We can achieve numerical stability by altering the algorithm to consider probabilities conditioned by the hidden sequence instead of joint by it. Formally, we define a scaled forward parameter $\hat{\tilde{\alpha}}$ as follows:

$$\hat{\tilde{\alpha}}_t(z_t, k, q) = \sum_{Z_{1:t-1}:O_r(Z_{1:t})=k} \sum_{Z_{1:t-1}:\tilde{\alpha}_r(Z_{1:t})=k} \frac{P(Z_{1:t}, FA_H(r)_t = q|X_{1:t})}{P(X_{1:t})}$$

$$= \frac{\tilde{\alpha}_t(z_t, k, q)}{P(X_{1:t})}$$

$^1$We have observed loss of numerical stability for sequence lengths of only a few hundreds when using 64-bit Java doubles for storing table entries.
As seen above, $\hat{\alpha}$ is just the forward parameter scaled by the inverse of the observable subsequence likelihood, hence the name of the parameter.

Because each $\hat{\alpha}$ $t$-cube sums to $P(X_{1:t})$, each $t$-cube of $\hat{\alpha}$-values will sum to 1:

$$\sum_{z_t,k,q} \hat{\alpha}_t(z_t, k, q) = 1$$

which effectively eliminates any instability problem caused by large $t$-values. Furthermore, by comparing the formula of equation 3.4 with equation 3.3 it is evident that

$$P(O_t(Z_{1:T}) = k|X_{1:T}) = \sum_{z_T,q} \hat{\alpha}_T(z_T, k, q)$$

In other words, we get our occurrence probability distribution directly from the sums of $\hat{\alpha}_T$ entries with equal $k$.

So how do we calculate $\hat{\alpha}$ in practice? A natural approach at time $t$ is to first calculate “unscaled” values using the formulas for the $\tilde{\alpha}$ recursion, then scale the values by the same scaling factor $c_t$, such that they sum to 1, and finally save these values in the table along with storing $c_t$. Scaling a $t$-cube by $c_t$ in this way effectively scales all subsequent $t$-cubes by $c_t$ as well, since their values will be calculated by a series of multiplications and additions of the values in the $t$-cube, and the distributive property holds under these operations.

By filling out the $\hat{\alpha}$-table and scaling each $t$-cube by $c_t$ in this way, we end up with a final $t$-cube scaled by all the $c_t$-values, that is, scaled by $c_{\text{product}} = \prod_{t'=1}^{T} c_{t'}$. We can transform it to the final $t$-cube in $\tilde{\alpha}$ by “unscaling” (i.e. dividing by $c_{\text{product}}$). Furthermore, it is evident that $c_{\text{product}}$ is then equal to the sum of the final $t$-cube in $\tilde{\alpha}$; in other words, $c_{\text{product}}$ equals the probability of the observable sequence.

Having discussed $c$-values from an intuitive point of view, let us now take a closer look at the mathematics behind them. We define $c_t$ as

$$c_t = P(x_t|X_{1:t-1})$$

which entails that

$$P(X_{1:t}) = \prod_{t'=1}^{t} c_{t'}$$

Comparing to equation 3.4 we get that

$$\tilde{\alpha}_t(z_t, k, q) = (\prod_{t'=1}^{t} c_{t'}) \hat{\alpha}_t(z_t, k, q)$$

Inserting this into 3.2, we get that
\[
\tilde{\alpha}_t(z_t, k, q) = \begin{cases} 
\begin{align*}
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \tilde{\alpha}_{t-1}(z_{t-1}, k, q') & \text{if } q \notin A \\
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \tilde{\alpha}_{t-1}(z_{t-1}, k - 1, q') & \text{if } q \in A \land k > 0 \\
& 0 & \text{if } q \in A \land k = 0
\end{align*}
\end{cases}
\]

\[
\left( \prod_{t' = 1}^{t} c_{t'} \right) \hat{\alpha}_t(z_t, k, q) = \begin{cases} 
\begin{align*}
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \left( \prod_{t' = 1}^{t} c_{t'} \right) \hat{\alpha}_{t-1}(z_{t-1}, k, q') & \text{if } q \notin A \\
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \hat{\alpha}_{t-1}(z_{t-1}, k - 1, q') & \text{if } q \in A \land k > 0 \\
& 0 & \text{if } q \in A \land k = 0
\end{align*}
\end{cases}
\]

which we use in the \( \hat{\alpha} \) recursion. For calculating \( c_t \) and the \( \hat{\alpha}_t \)-entries, we define a temporary \( t \)-cube, \( \delta_t \), containing the \( c_t \hat{\alpha}_t(z_t, k, q) \) values:

\[
\delta_t(z_t, k, q) = c_t \hat{\alpha}_t(z_t, k, q) = \begin{cases} 
\begin{align*}
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \hat{\alpha}_{t-1}(z_{t-1}, k, q') & \text{if } q \notin A \\
& b_{z_t,x_t} \sum_{z_{t-1}} \sum_{q' \in Q'(z_{t}, q)} \hat{\alpha}_{t-1}(z_{t-1}, k - 1, q') & \text{if } q \in A \land k > 0 \\
& 0 & \text{if } q \in A \land k = 0
\end{align*}
\end{cases}
\]

Cf. equations \[3.5\] and \[3.7\], we have that:

\[
\sum_{z_t,k,q} \delta_t(z_t, k, q) = \sum_{z_t,k,q} c_t \hat{\alpha}_t(z_t, k, q) = c_t \sum_{z_t,k,q} \hat{\alpha}_t(z_t, k, q) = c_t \tag{3.8}
\]

In other words, by summing the values of \( \delta_t \), we find \( c_t \), and then we can calculate the \( \hat{\alpha}_t \)-entries as

\[
\hat{\alpha}_t(z_t, k, q) = \frac{\delta_t(z_t, k, q)}{c_t}
\]

**Pseudocode** The pseudocode for the restricted forward algorithm with scaling is shown in algorithm \[1\]
Algorithm 1 RESTRICTED-FORWARD (1/2)

Require: $N, \Pi, A, B, \text{regexp}, X$

Ensure: $\hat{\alpha}$ table complete along with the $c$ array

1: $\text{dfa} \leftarrow \text{CONSTRUCT-DFA}(N, \text{regexp})$
2: $Q \leftarrow \text{states}[\text{dfa}]$
3: $m \leftarrow \text{CALCULATE-M}(T, \text{dfa})$
4: $c[0] \leftarrow 0$
5: \textbf{for} $z \leftarrow 0$ to $N - 1$ \textbf{do} \hspace{1cm} \Comment{Base case}
6: \hspace{0.5cm} $\text{val} \leftarrow \Pi[z] \cdot B[z][X[0]]$
7: \hspace{0.5cm} \textbf{for} $q \leftarrow 0$ to $|Q| - 1$ \textbf{do}
8: \hspace{1cm} \textbf{for} $k \leftarrow 0$ to $m$ \textbf{do}
9: \hspace{1.5cm} $\hat{\alpha}[0][z][k][q] \leftarrow 0$
10: \hspace{1cm} \textbf{if} TRANSITION-EXISTS(initial[\text{dfa}], z, q) \textbf{then}
11: \hspace{1.5cm} \textbf{if} accept[q] \textbf{then}
12: \hspace{2cm} $\hat{\alpha}[0][z][1][q] \leftarrow \text{val}$
13: \hspace{1.5cm} \textbf{else}
14: \hspace{2cm} $\hat{\alpha}[0][z][0][q] \leftarrow \text{val}$
15: \hspace{1cm} $c[0] \leftarrow c[0] + \text{val}$
16: \hspace{0.5cm} \textbf{for} $z \leftarrow 0$ to $N - 1$ \textbf{do} \hspace{1cm} \Comment{Perform scaling}
17: \hspace{1cm} \textbf{for} $q \leftarrow 0$ to $|Q| - 1$ \textbf{do}
18: \hspace{1.5cm} \textbf{for} $k \leftarrow 0$ to $m$ \textbf{do}
19: \hspace{2.5cm} $\hat{\alpha}[0][z][k][q] \leftarrow \frac{\hat{\alpha}[0][z][k][q]}{c[0]}$

The intention of the pseudocode is to present the central structure of the algorithm. Therefore, some trivial code has been abstracted away, and code that is independent of the core algorithm has been extracted into methods to be used as subroutines.

At first, we construct the DFA $\mathcal{F}A_{\mathcal{H}}(r)$ in the method CONSTRUCT-DFA, which uses an external DFA library [17]. Additionally, we calculate the size of the $k$-dimension, $m$, in the method CALCULATE-M. Several different approaches to calculating $m$ exist, some of which will be discussed later in this section.

After this initial set-up, we perform the base case of the algorithm, according to equation [3.1]. During the base case, we keep track of the sum of entry values, which we save as $c[0]$. Afterwards, we then scale them by dividing by $c[0]$ so that equation [3.5] is satisfied. During the base case, we use the method TRANSITION-EXISTS to check whether the DFA initial state has an outgoing transition to the appropriate $q$-state using the symbol $z$. 

23
Algorithm 1 RESTRICTED-FORWARD (2/2)

20: for $t \leftarrow 1$ to $T - 1$ do
21:   $c[t] \leftarrow 0$
22: for $z_t \leftarrow 0$ to $N - 1$ do
23:   for $k \leftarrow 0$ to $m$ do
24:     for $q \leftarrow 0$ to $|Q| - 1$ do
25:       if $\text{accept}[q]$ & $k = 0$ then
26:         $\hat{\alpha}[t][z_t][k][q] \leftarrow 0$
27:         continue
28:     
29:     $Q' \leftarrow \text{FIND-PREDECESSORS}(z, q)$
30:     for $z_{t-1} \leftarrow 0$ to $N - 1$ do
31:       $\text{inSum} \leftarrow 0$
32:     for all $q'$ in $Q'$ do
33:       if $\text{accept}[q]$ then
34:         $\text{inSum} \leftarrow \text{inSum} + \hat{\alpha}[t - 1][z_{t-1}][k - 1][q']$
35:       else
36:         $\text{inSum} \leftarrow \text{inSum} + \hat{\alpha}[t - 1][z_{t-1}][k][q']$
37:     $\text{outSum} \leftarrow \text{inSum} + \text{inSum} \cdot A[z_{t-1}][z_t]$
38:     $\hat{\alpha}[t][z_t][k][q] \leftarrow \text{outSum} \cdot B[z_t][X[t]]$
39:     $c[t] \leftarrow c[t] + \hat{\alpha}[t][z_t][k][q]$
40:     
41: for $z \leftarrow 0$ to $N - 1$ do
42:   for $q \leftarrow 0$ to $|Q| - 1$ do
43:     for $k \leftarrow 0$ to $m$ do
44:       $\hat{\alpha}[t][z][k][q] \leftarrow \frac{\hat{\alpha}[t][z][k][q]}{c[t]}$

Similarly, in the recursive case, we calculate the $\delta_t$-values according to equation 3.7 and subsequently scale them by $c[t]$. We do not actually need to store the $\delta_t$-values separately; instead, we just save them in the $\hat{\alpha}_t$-table and then overwrite them when scaling. Determining the set $Q'$ of possible previous DFA states to some state $q$ using the transition symbol $z$ is done in the method FIND-PREDECESSORS. $\text{inSum}$ and $\text{outSum}$ are values of the innermost and outermost sum in equation 3.7 respectively.

After all $t$-cubes are filled out and scaled in this way, the $\hat{\alpha}$-table is complete and the algorithm terminates. In practice, to extract the results, we sum entries in the final $t$-cube with equal $k$-values. These sums form the probability distribution which can then be returned.

To get the real (i.e. not scaled) probabilities for the final $t$-cube, we can multiply $\prod_{t'=1}^{T} c[t']$ with the scaled probabilities, cf. equation 3.6. To avoid numerical instability, we use the Java class BigDecimal for storing such values. Note that multiplying high precision BigDecimal objects is a very time-consuming task, so using BigDecimals instead of doubles in all table entries would slow down the algorithm significantly.

2
http://docs.oracle.com/javase/6/docs/api/java/math/BigDecimal.html
The algorithm as presented in the pseudocode is similar to the restricted Viterbi algorithm, which will be explained in section 3.1.2. Additional implementation details that are common for the two algorithms will be described in that section.

**Improving space consumption**  The \( \tilde{\alpha} \) table consists of \( T \) \( t \)-cubes of size \( N \cdot m \cdot |Q| \), so the space complexity of an algorithm saving all of the \( \tilde{\alpha} \) table will be \( O(T \cdot N \cdot m \cdot |Q|) \), as mentioned in the theory section. Some applications of the forward algorithm (e.g. Baum-Welch training [2]) requires all of the \( \tilde{\alpha} \) table present in memory. However, for our application, we are only interested in the contents of the final \( t \)-cube, so we might be able to decrease the space consumption. In fact, we notice that the recursion step for cube \( t \) only uses entries from cube \( t - 1 \), so a simple and effective way of reducing the space consumption is to only keep the current and previous \( t \)-cube in memory - along with the current product of scaling values, \( \prod_{t'=1}^{t} c_{t'} \), if we want to be able to unscale later. By implementing this simple improvement, the space complexity of our algorithm is reduced to \( O(N \cdot m \cdot |Q|) \) without affecting time complexity. This makes a big difference in practice, and increasingly so for longer sequences.

**Choosing an upper bound for the number of occurrences**  One of the dimensions in the \( \tilde{\alpha} \) table is of size \( m + 1 \), where \( m \) is a chosen upper bound of \( k \), the number of pattern occurrences. In algorithm [1] the calculation of \( m \) is left open to the procedure CALCULATE-M. According to the theory section, \( m \) is equal to the maximum possible number of pattern occurrences, which we will refer to as \( k_{\text{max}} \). In practice, however, it might not be simple to calculate \( k_{\text{max}} \), and so we allow other choices of \( m \), as long as it is an upper bound for the occurrence count, i.e. \( m \geq k_{\text{max}} \). This does not affect the results of the algorithm, but to limit space and time consumption it is important not to choose an unreasonably large \( m \).

In the remainder of this thesis, we will consider \( m \) the maximum number of occurrences that we allow in our restricted forward table. Generally, \( m \) will be chosen such that \( m \geq k_{\text{max}} \), and we will consequently sometimes refer to \( m \) as simply a particular choice of upper bound for the number of occurrences.

A trivial value of the upper bound of number of occurrences is \( T \), since we maximally increase \( k \) by one per \( t \)-value. However, in our implementation, we would like an \( m \)-value that is closer to \( k_{\text{max}} \) while still being a general upper bound. This can be achieved by taking a closer look at the DFA and its limitations regarding the possible frequency of accept states. If the goal were to calculate \( k_{\text{max}} \) exactly, we would need to take the particular HMM into account as well. The reason for this is that each zero probability HMM transition (e.g. from \( z_i \) to \( z_j \)) renders two subsequent DFA transitions (using first \( z_i \) and then \( z_j \) as symbols) impossible in our algorithm. In other words, the existence of zero probability HMM transitions might enable us to decrease an upper bound on the occurrence count further. To simplify matters, however, we will ignore such possible HMM contributions and only attempt to minimize \( m \) based on the DFA.
Now, let $\text{dist}_{\text{init}}$ be the minimum number of steps from the initial state to an accepting state in the DFA. If this number is higher than the number of steps we can take, we know that we cannot reach an accept state and $m$ is consequently zero. Otherwise, we can find the minimum number of steps $\min_{i,j}(\text{dist}_{ij})$ from an accept state $a_i$ to an accept state $a_j$. We can then calculate an upper bound of accept states in a DFA path of length $T$ as

$$
m = \left\lfloor 1 + \frac{T - \text{dist}_{\text{init}}}{\min_{i,j}(\text{dist}_{ij})} \right\rfloor (3.9)$$

where the 1 corresponds to the first occurrence and the fraction corresponds to the maximum number of occurrences in the rest of the string (which has length $T - \text{dist}_{\text{init}}$).

In regard to making sure that $m \geq k_{\max}$, our approach expects the worst case: that we follow a path of length $\text{dist}_{\text{init}}$ from the initial state to an accept state and then consecutively follow a path of length $\min_{i,j}(\text{dist}_{ij})$ from that accept state to the next. This virtual path of steps may of course not exist, but it will correspond to an upper bound for the number of occurrences. In practice, how much $m$ is decreased by when using this approach depends on the regular expression in question.

We have implemented the described method as the standard way of calculating $m$ in our restricted forward algorithm. Pseudocode for the method, \textsc{Calculate-M}, can be seen in algorithm 2.

\begin{algorithm}
\caption{\textsc{Calculate-M}}
\begin{algorithmic}[1]
\Require $T, \text{dfa}$
\Ensure $m$ is a non-trivial upper bound on occurrences
1: $q_{\text{init}} \leftarrow \text{initial dfa state}$ \hfill $\triangleright$ Min. initial distance
2: $\text{dist}_{\text{init}} \leftarrow \text{DIST-TO-ACCEPT}(\text{dfa}, q_{\text{init}})$
3: \If { $\text{dist}_{\text{init}} > T$} \Return 0 \EndIf
4: $\min_{i,j}(\text{dist}_{ij}) = \infty$
5: \For { each $a_x \in \text{accept - states}[\text{dfa}]$} \Do
6: \quad $\min_{x,j}(\text{dist}_{xj}) \leftarrow \text{DIST-TO-ACCEPT}(\text{dfa}, a_x)$
7: \quad \If { $\min_{x,j}(\text{dist}_{xj}) < \min_{i,j}(\text{dist}_{ij})$} \Then
8: \quad \quad $\min_{i,j}(\text{dist}_{ij}) = \min_{x,j}(\text{dist}_{xj})$ \hfill $\triangleright$ Min. subsequent distance
\EndIf
\EndFor
\Return $\left\lfloor 1 + \frac{T - \text{dist}_{\text{init}}}{\min_{i,j}(\text{dist}_{ij})} \right\rfloor$
\end{algorithmic}
\end{algorithm}
Algorithm 3 DIST-TO-ACCEPT

Require: $DFA, q$
Ensure: $\min[dist[a]]$ where $dist[a]$ is the distance from $q$ to an accept state $a$

1: for each $v$ in $dfa$ do
2: \hspace{1em} $color[v] \leftarrow$ WHITE
3: \hspace{1em} $color[q] \leftarrow$ GRAY
4: \hspace{1em} $dist[q] \leftarrow$ 0
5: $Q \leftarrow \emptyset$
6: ENQUEUE($Q, q$)
7: while $Q \neq \emptyset$ do
8: \hspace{1em} $u \leftarrow$ DEQUEUE($Q$)
9: \hspace{1em} for each $v \in \text{Adj}[u]$ do
10: \hspace{2em} if $color[v] = \text{WHITE}$ then
11: \hspace{3em} $dist[v] = dist[u] + 1$
12: \hspace{2em} if $v \in \text{accept states}[dfa]$ then return $dist[v]$
13: \hspace{2em} $color[v] = \text{GRAY}$
14: \hspace{2em} ENQUEUE($Q, v$)
15: \hspace{1em} if $u = q$ then
16: \hspace{2em} $color[u] = \text{WHITE}$
17: \hspace{1em} else
18: \hspace{2em} $color[u] = \text{BLACK}$
19: \hspace{1em} return $\infty$

In CALCULATE-M, we first calculate $dist_{\text{init}}$, then run through all accept states and in order to find the shortest distance $min[dist_{ij}]$ between any two accepting states. Finally, we can calculate $m$ according to equation 3.9. For calculating these distances, a subroutine called DIST-TO-ACCEPT (see algorithm 3) is called, which finds the minimum distance to any accept state in the DFA from the input state $q$. DIST-TO-ACCEPT works similarly to standard breath-first-search, except for the fact that we want it to be able to find a path from $q$ to $q$ itself (in the case that $q$ is an accept state), thus we color $q$ white at the end of the first iteration of the while-loop. In the pseudocode, $Q$ denotes a FIFO-queue for gray-colored states, and ENQUEUE and DEQUEUE are simple functions for adding an element to a queue and removing the first element from a queue, respectively.

This method returns an upper bound for the number of pattern occurrences and is an improvement over setting $m = T$. In a later section 3.3, we will examine approaches for further decreasing the size of the $k$-dimension.

**Summary** In this section, we have introduced the restricted forward algorithm, which can be used to calculate a probability distribution over the number of occurrences of a regular expression in some hidden state sequence. We have also ensured numerical stability, made simple improvements and presented pseudocode for the algorithm.
3.1.2 The restricted Viterbi algorithm

Theory

Given a regular expression $r$, we want to calculate a most likely sequence of hidden states $Z_{1:T}$ restricted by $O_r(Z_{1:t}) \in [l, u]$. In other words, we restrict the Viterbi algorithm to hidden state sequences where our regular expression has matched between $l$ and $u$ times. For gene prediction, we could select our regular expression such that occurrences of the expression correspond to predicted genes. The specifics of such an expression depend on the particular HMM, but from an abstract point of view the expression would have to capture gene boundaries. For the simple gene prediction model in figure 2.7, we could have e.g. $r = (NN(C_1|R_3))((C_3|R_1)NN)$ as described in [20].

The occurrence bound should be set based on the information available. In some cases we might have experimental knowledge of the bounds on the number of genes, where in other cases we have to estimate them computationally. In this case, we can start out by running the forward algorithm to obtain a probability distribution over the number of occurrences. We can then set the bounds to cover, e.g., 95% of the distribution or set both the lower and upper bound to the expected number of occurrences. The choice on how to select the bounds should be according to our objective, e.g. maximizing the MCC value [1] of gene prediction on some given data set.

As with the restricted forward algorithm, we run the Viterbi algorithm simultaneously with our deterministic automata $FA_H(r)$ to give us information about the current number of occurrences. We need to keep track of the run of the DFA, so we must expand our table from the original Viterbi algorithm. We define our $\tilde{\omega}$ parameter to

$$
\tilde{\omega}_t(z_t, k, q) = \max_{Z_{1:t-1}:O_r(Z_{1:t})=k} P(X_{1:t}, Z_{1:t}, FA_H(r)_t = q)
$$

(3.10)

where $k = 0, \ldots, u$ and $FA_H(r)_t$ is the state that FA is in at time $t$. A table entry is the probability of a most likely path having observed $X_{1:t}$, ending in hidden state $z_t$ and automaton state $q$ at time $t$ and having seen $k$ occurrences of the pattern, corresponding to having visited accepting states $k$ times.

The table can be computed recursively using base case

$$
\tilde{\omega}_1(z_1, k, q) = \tilde{\alpha}_1(z_1, k, q)
$$

and recursion case

$$
\tilde{\omega}_t(z_t, k, q) = \begin{cases}
    b_{z_t, x_t} \cdot \max_{z_{t-1}=z_t-1} a_{z_{t-1}, z_t} \cdot \max_{q' \in Q'(z_{t-1}, q)} \tilde{\omega}_{t-1}(z_{t-1}, k, q') & \text{if } q \notin A \\
    b_{z_t, x_t} \cdot \max_{z_{t-1}=z_t-1} a_{z_{t-1}, z_t} \cdot \max_{q' \in Q'(z_{t-1}, q)} \tilde{\omega}_{t-1}(z_{t-1}, k-1, q') & \text{if } q \in A \land k > 0 \\
    0 & \text{if } q \in A \land k = 0
\end{cases}
$$

The base case $\tilde{\omega}_1$ is the same as for $\tilde{\alpha}_1$, namely the probability of starting in hidden state $z_1$ times the probability of emitting symbol $x_1$ from hidden state $z_1$.
for entries where the following applies: a) there exists a transition from the DFA initial state, $q_0$, to $q$ using hidden state $z_1$ and b) either $q$ is not an accepting state and $k = 0$ or $q$ is an accepting state and $k = 1$. For other entries the value should be zero. This is also intuitively what we want, since the simultaneous run of the DFA should only allow valid initial state and transitions according to the first hidden state, $z_1$.

In the recursion case, for some previous hidden state $z_{t-1}$ and some valid previous DFA state $q'$ using hidden state $z_t$ as transition, we can calculate a candidate to $\tilde{\omega}_t(z_t, k, q)$ as the probability of emitting symbol $x_t$ from hidden state $z_t$ times the probability of transitioning from previous state $z_{t-1}$ to current state $z_t$ times $\tilde{\omega}_{t-1}(z_{t-1}, k', q')$ where $k' = k$ if $q$ is not an accepting state and $k' = k - 1$ if $q$ is an accepting state. To calculate $\tilde{\omega}_t(z_t, k, q)$, we must maximize over all previous hidden states and all their corresponding possible previous states in the DFA.

Note that the table contains probabilities of the most likely hidden state sequence restricted on the number of occurrences of the regular expression. To calculate the most likely hidden state sequence, we need to backtrack the $\tilde{\omega}$ table. We backtrack through the $t$ dimension and so our backtracked entries are $(z_t, k, q)$ triplets. Because we restrict the number of occurrences, we must start the backtracking from an entry where $k \in [l, u]$. We therefore start our backtracking from entry $\arg \max_{(z_T, k \in [l, u], q)} \tilde{\omega}_T(z_T, k, q)$. Backtracking an entry takes $O(N \cdot |Q|)$ time and so backtracking can be done in time $O(T \cdot N \cdot |Q|)$.

The $\tilde{\omega}$ table has $T \cdot N \cdot u \cdot |Q|$ entries. For every entry we maximize over all previous hidden states and possible previous states in the DFA and so it takes $O(N \cdot |Q|)$ to compute each entry. This leads to a time complexity of $O(T \cdot N^2 \cdot u \cdot |Q|^2)$ and a space complexity of $O(T \cdot N \cdot u \cdot |Q|)$.

### Implementation

In this section, we explain the details required to enable us to implement the algorithm, including pseudocode and how to address problems regarding numerical stability.

**Numerical stability** In the naïve approach, we represent the $\tilde{\omega}$ table as a multidimensional array containing floating point numbers with 64-bit precision. With this approach we run into the same problem as with the forward algorithm, namely that the probabilities get closer and closer to zero in table entries for increasing values of $t$. At some point, the probabilities are too small to be representable in a 64-bit precision floating point number and so will get the value zero if implemented this way.

We address this problem by instead calculating the logarithm of the probabilities. Because the log function is a *strictly monotonic increasing* function, it preserves the order, thus we have $\max(\log f) = \log(\max f)$ where $f$ is a function. We define a new $\hat{\omega}$ table, where the entries contain the logarithm to the probabilities of the $\tilde{\omega}$ table. We denote this as working in *log space*. When we make the transformation to log space, we need to decide how to represent the
undefined value $\log 0$. We have $\lim_{x \to 0^+} \log x = -\infty$ and using $-\infty$ to represent it also fits well in our context of maximization. For the following equations regarding log space, we therefore resolve the undefined value $\log 0$ to $-\infty$. We get the base case

$$
\tilde{\omega}_1(z_1, k, q) = \log \tilde{\omega}_1(z_1, k, q)
= \begin{cases} 
\log \pi_{z_1} + \log b_{z_1,x_1} & \text{if } q_0 \in Q'(z_1, q) \land \left((q \notin A \land k = 0) \lor (q \in A \land k = 1)\right) \\
-\infty & \text{otherwise}
\end{cases}
$$

(3.11)

and recursion case

$$
\tilde{\omega}_t(z_t, k, q) = \log \tilde{\omega}_t(z_t, k, q)
= \begin{cases} 
\log \left( b_{z_t,x_t} \cdot \max_{z_{t-1}} a_{z_{t-1},z_t} \cdot \max_{q' \in Q'(z_t,q)} \tilde{\omega}_{t-1}(z_{t-1}, k, q') \right) & \text{if } q \notin A \\
-\infty & \text{if } q \in A \land k = 0
\end{cases}
$$

(3.12)

For all the experiments we run in this thesis, the large negative numbers we obtain by moving into log space can be represented by 64-bit floating point numbers without loss of numerical stability.
Algorithm 4 RESTRICTED-VITERBI (1/2)

Require: \(N, \Pi, A, B, r, X, u\)
Ensure: \(\hat{\omega}\) table complete

1: \(dfa \leftarrow \text{CONSTRUCT-DFA}(N, r)\)
2: \(Q \leftarrow \text{states}[dfa]\)
3: \(\text{for } z \leftarrow 0 \text{ to } N - 1 \text{ do}\) \hspace{1cm} \(\triangleright\) Base case
4: \(\text{val} \leftarrow \log(\Pi[z]) + \log(B[z][X[0]])\)
5: \(\text{for } q \leftarrow 0 \text{ to } |Q| - 1 \text{ do}\)
6: \(\text{for } k \leftarrow 0 \text{ to } u \text{ do}\)
7: \(\hat{\omega}[0][z][k][q] \leftarrow -\infty\)
8: \(\text{if TRANSITION-EXISTS}(\text{initial}[dfa], z, q) \text{ then}\)
9: \(\text{if accept}[q] \text{ then}\)
10: \(\hat{\omega}[0][z][1][q] \leftarrow \text{val}\)
11: \(\text{else}\)
12: \(\hat{\omega}[0][z][0][q] \leftarrow \text{val}\)

Pseudocode  In algorithm 4 we present pseudocode for the restricted Viterbi algorithm. In the implementation we have some practical conversions between automaton states and their corresponding integer representation, but for clarity, this has been abstracted away in the pseudocode where we use their integer representation to account for both. Also, we move from 1-indexed sequences to 0-indexed arrays to make the pseudocode as close as possible to the implementation. Code that is trivial or which is not central to the algorithm has been abstracted away into external methods that will be explained in the this text.

As input we need the HMM parameters \(N, \Pi, A, B\) as well as the string of observables \(X\), the regular expression \(r\) and the upper bound \(u\) on the occurrence count. In the method CONSTRUCT-DFA, we use an external library \cite{17} to construct a DFA \(FA_H(r) = (h_1| h_2| \cdots | h_N)^*(r)\) that recognizes strings where the regular expression \(r\) is a suffix.

In the base case, where \(t = 0\), we initialize the table entries to \(-\infty\). In the method TRANSITION-EXISTS(srcState, \(z\), destState), we iterate transitions from srcState to destState and check whether one of these transitions use transition symbol \(z\). We iterate possible values for hidden state \(z\) and DFA state \(q\) and find table entries where a transition from the initial automaton state to \(q\) exists using hidden state \(z\) as transition symbol. This corresponds to the first step of the automaton. Now, according to equation \cite{31} we fill the probability of the table entry for \(k = 0\) if \(q\) is not an accept state and for \(k = 1\) if \(q\) is an accept state since this increases the occurrence number to 1.
In the recursion case, where \( t > 0 \), for increasing values of \( t \), we iterate all possible choices of hidden states \( z_t \), DFA states \( q \) and regular expression occurrences \( k \). If \( q \) is an accept state and \( k = 0 \) we have from equation \([3.12]\) that the table entry is \(-\infty\) since if \( q \) is an accept state, the occurrence number must be greater than zero. Otherwise, we proceed in \textsc{Find-Predecessors}(\( z_t, q \)) with finding the set of DFA states \( Q' \) that has a transition to \( q \) using transition symbol \( z_t \). Because of the way states and transitions are represented in the external library \([17]\), we need to iterate outgoing transitions from all states to find this set. On lines 22-35 we iterate all possible previous hidden states \( z_{t-1} \) and previous DFA states \( q' \in Q' \) to find the value of the maximizations in equation \([3.12]\). The check on line 25 of whether \( q \) is an accept state or not, corresponds to the two different cases in the equation. On line 36 we have found the value of the maximizations and can calculate the value of the table entry.

When the algorithm terminates, the \( \hat{\omega} \) table is complete.
Algorithm 5 RESTRICTED-VITERBI-BACKTRACK (1/2)

Require: \( \tilde{\omega}, N, A, r, l, u \)
Ensure: trace\(_z\)[0], \ldots , trace\(_z\)[T-1] is a most likely sequence of hidden states

1: \( dfa \leftarrow \text{CONSTRUCT-DFA}(N, regexp) \)
2: \( Q \leftarrow \text{states}[dfa] \)
3: \( \text{max} \leftarrow -\infty \)
4: \textbf{for} \( z \leftarrow 0 \) to \( N - 1 \) \textbf{do} \hspace{1cm} \triangleright \text{Base case}
5: \hspace{0.5cm} \textbf{for} \( k \leftarrow l \) to \( u \) \textbf{do}
6: \hspace{1cm} \textbf{for} \( q \leftarrow 0 \) to \( |Q| - 1 \) \textbf{do}
7: \hspace{1.5cm} \text{val} \leftarrow \tilde{\omega}[T-1][z][k][q]
8: \hspace{1.5cm} \textbf{if} \text{val} > \text{max} \textbf{then}
9: \hspace{2cm} \text{max} \leftarrow \text{val}
10: \hspace{2cm} \text{max}_z \leftarrow z
11: \hspace{2cm} \text{max}_k \leftarrow k
12: \hspace{2cm} \text{max}_q \leftarrow q
13: \hspace{1cm} \text{trace}_z[T-1] \leftarrow \text{max}_z
14: \hspace{1cm} \text{trace}_k[T-1] \leftarrow \text{max}_k
15: \hspace{1cm} \text{trace}_q[T-1] \leftarrow \text{max}_q

The pseudocode for backtracking the \( \tilde{\omega} \) table is found in algorithm\textsuperscript{5}. As input we need the \( \tilde{\omega} \) table, the HMM parameters \( N \) and \( A \) as well as the regular expression \( r \), the lower bound \( l \) and the upper bound \( u \) for the occurrence count. In our code, we reuse the constructed DFA instead of constructing it again for the backtracking, but the pseudocode includes it for completeness. Note again the switch from 1-indexed sequences in the theoretical section to 0-indexed arrays in the pseudocode. Initialization of \( \text{max}_z, \text{max}_k, \text{max}_q, \text{innerMax}_k, \text{innerMax}_q \) has been left out to make the pseudocode more readable.

The base case of the backtracking is finding the maximum entry for \( t = T - 1 \), i.e. the maximum entry in the final \( t \)-cube. Now, this entry is restricted on the number of occurrences as it must be within the range \([l, u]\) so only this range of \( k \) values are iterated when searching for the maximum. The maximum entry is found and stored as \( (\text{max}_z, \text{max}_k, \text{max}_q) \) in lines 3-12. The backtracking trace is saved in the designated arrays \( \text{trace}_z[], \text{trace}_k[], \text{trace}_q[] \).
Algorithm 5 RESTRICTED-VITERBI-BACKTRACK (2/2)

16: for $t \leftarrow T - 1$ to 1 do \Comment{Recursion case}
17: \hspace{1em} $max \leftarrow -\infty$
18: \hspace{1em} $Q' \leftarrow$ FIND-PREDECESSORS($trace_z[t], trace_q[t]$)
19: \hspace{1em} for $z_{t-1} \leftarrow 0$ to $N - 1$ do
20: \hspace{2em} $innerMax \leftarrow -\infty$
21: \hspace{2em} for all $q'$ in $Q'$ do
22: \hspace{3em} if $accept[q]$ then
23: \hspace{4em} $inner \leftarrow \hat{\omega}[t-1][z_{t-1}][trace_k[t]-1][q']$
24: \hspace{4em} if $inner > innerMax$ then
25: \hspace{5em} $innerMax \leftarrow inner$
26: \hspace{5em} $innerMax_k \leftarrow trace_k[t] - 1$
27: \hspace{5em} $innerMax_q \leftarrow q'$
28: \hspace{3em} else
29: \hspace{4em} $inner \leftarrow \hat{\omega}[t-1][z_{t-1}][trace_k[t]][q']$
30: \hspace{4em} if $inner > innerMax$ then
31: \hspace{5em} $innerMax \leftarrow inner$
32: \hspace{5em} $innerMax_k \leftarrow trace_k[t]$
33: \hspace{5em} $innerMax_q \leftarrow q'$
34: \hspace{2em} $outer \leftarrow innerMax + \log A[z_{t-1}][z_t]$
35: \hspace{2em} if $outer > max$ then
36: \hspace{3em} $max \leftarrow outer$
37: \hspace{3em} $max_z \leftarrow z_{t-1}$
38: \hspace{3em} $max_k \leftarrow innerMax_k$
39: \hspace{3em} $max_q \leftarrow innerMax_q$
40: \hspace{2em} $trace_z[t-1] \leftarrow max_z$
41: \hspace{2em} $trace_k[t-1] \leftarrow max_k$
42: \hspace{2em} $trace_q[t-1] \leftarrow max_q$

The pseudocode for the recursion case of the backtracking is shown on lines 16-42. The most recent backtracked entry ($trace_z[t], trace_k[t], trace_q[t]$) is used to find the possible previous DFA states $Q'$ and as in the filling of the $\hat{\omega}$ table, all $q' \in Q'$ together with previous hidden states $z_{t-1}$ are iterated to find a maximum. In the backtracking pseudocode, though, we not only find the maximum, but also store in the $trace$ arrays which entry gave rise to it. Observe that the term $\log b_{z_{t-1}x_t}$ remains constant as we are searching through the different previous hidden states $z_{t-1}$. So, since we do not need to recalculate the actual table value, we can ignore that term because it does not affect which entry becomes the maximum.

**Summary** In this section, we have introduced the restricted Viterbi algorithm, which can find the most likely the hidden state sequence restricted on the number of occurrences of a regular expression. We have also ensured numerical stability and presented pseudocode for the algorithm.
3.1.3 Improvement of the time consumption

After implementing the restricted algorithms, we verified through simple experiments that the practical time consumption as well as the space consumption would become a problem if the algorithms were to be used for sequences of realistic lengths.

The time complexity of the restricted forward and Viterbi algorithms both contain a factor $|Q|^2$ since the calculation of every table entry requires us to iterate through the set of possible previous DFA states, $Q'$, when calculating the sum and maximum respectively. We cannot change the time complexity, since in the worst case $|Q'|$ is linear in the number of DFA states, $|Q|$. $|Q'|$ depends on the regular expression, so we cannot say anything general about it, but for gene prediction, we typically choose a simple regular expression, e.g. capturing the number of genes, and we find that for these expressions, $|Q'|$ is typically constant and not linear in $|Q|$. As mentioned in the text on algorithm 4, we iterate through outgoing transitions from all states to find $Q'$ and so even though $|Q'|$ in practice might not be not linear in $|Q|$, we still spend time linear to $|Q|$.

To solve this issue, we choose to precompute the set of possible previous DFA states for each possible combination of transition symbol and state. The table has $N \times |Q|$ entries and each entry contains a set of up to $|Q|$ states. Each set takes time proportional to $|Q|$ to compute so the time and space complexity becomes $O(N \times |Q|^2)$. Now, when running the restricted algorithms, for every table cell, it only requires a constant time table lookup to get the set of possible previous DFS states, $Q'$. If, as claimed, $|Q'|$ for regular expressions relevant to gene prediction is, in practice, typically only constant and not linear in $|Q|$, we save a factor $|Q|$ in the running time. In section 4.3 we perform an experiment to verify this expectation on our simple gene prediction model. Note that in the general case, where $|Q'|$ is linear in $|Q|$, precomputing the possible previous DFA states will still speed up the algorithm’s running time, even though the trade off may not be as beneficial.

Another simple improvement of the naïve implementations of the algorithms is based on the observation that for a given $t$-value, there exists an upper bound for the occurrence count. The most simple, non-trivial upper bound is $t$, i.e. we cannot have visited more than $t$ accepting states, since we only made $t$ transitions in the DFA. Therefore, we know that all entries where $k > t$ will have a probability of zero (or a corresponding value of $-\infty$ in log space). In other words, we have the possibility of skipping the calculation of that part of the table. The benefit of doing this for the restricted Viterbi algorithm is fairly small, since in practice, we choose a $u$ much smaller than $T$. The restricted forward algorithm would benefit more from such an optimization, depending on the relative size of $m$ compared to $T$. We have chosen not to implement this improvement. Instead, in section 3.3.2 we will examine another approach for discarding a part of the table.

In the following sections, we investigate different improvements to the time and space consumption of the restricted algorithms. In most cases, there is a trade off involved. When we improve the time consumption, we usually worsen
the space consumption and vice versa. We have tried to identify and implement ideas where this trade off seems beneficial.

### 3.2 Space improvements for the restricted Viterbi algorithm

In this section we look at different space improvements for the Viterbi algorithm. Our contribution is describing the improvements and adapting them to work in the context of the restricted Viterbi algorithm. Furthermore, we develop pseudocode for one of the improvements and implement it.

First, let us motivate the need to look at space improvements for the restricted Viterbi algorithm. As an example, we might want to run the algorithm in a scenario with observable string length $T = 50,000$, upper bound $u = 1,000$ for the number of occurrences, number of hidden states $N = 20$ and number of states in the automaton $|Q| = 10$. This example represents a slightly bigger HMM for gene prediction, a small regular expression for catching gene boundaries on an observable string of a closer to realistic length with a plausible upper bound on the number of genes. In this example, our $\tilde{\omega}$ table has size $50,000 \cdot 1,000 \cdot 20 \cdot 10$ entries $= 10^{10}$ entries. If every entry holds an 8 byte floating point number, we get a space consumption of $10^{10}$ entries $\cdot$ 8 bytes entry $\approx 75GB$. Even serving just as a small example, we can quickly conclude that examining space improvements will indeed be useful.

Several attempts to reduce the space consumption of the basic Viterbi algorithm have been made. Hirschberg’s method [9], which was originally described for pairwise sequence alignment, can be used to reduce the space complexity of the Viterbi algorithm. In this section, we choose to present the checkpoint algorithm [19] and the approximate linear space method [4]. The former features a simple way to reduce the theoretical space complexity while the latter features an elegant way to reduce the practical space consumption.

#### 3.2.1 The checkpoint algorithm

Recall that we in the Viterbi algorithm start by filling out a two dimensional table $\omega$ containing probabilities of a most likely hidden path ending in a specific hidden state. We then backtrack this table to find a most likely hidden state sequence. In the forward algorithm, for the uses of finding the probability of an observable sequence, we only need the last column of the $\alpha$ table. We can therefore reduce the space consumption by a factor $T$ since we only have to store the two most recent columns of the table as we calculate columns of increasing values of $t$. In the Viterbi algorithm, we cannot apply the same idea, because we need the entire table for backtracking the most likely sequence of hidden states. We can, however, reduce the space consumption in other ways.

One idea for the Viterbi algorithm is to use a checkpoint algorithm [19], where we divide the input sequence into $\sqrt{T}$ blocks of $\sqrt{T}$ symbols each. During the computation of the $\omega$ table, we only save the first column of each block, together with the last column of the table.
During the backtracking phase, we use the last checkpoint column to re-
calculate the block that enables us to backtrack to an entry in the checkpoint
column. We then recalculate from the previous checkpoint column and so on
until we have backtracked the entire table. During this process, we save the $\sqrt{T}$
checkpoint columns and one block of $\sqrt{T}$ columns. This gives us an improved
space complexity of $O(N \cdot \sqrt{T})$ and even though we compute the table twice,
we still get a time complexity of $O(N^2 \cdot T)$.

**Checkpoints for the restricted Viterbi** For the graph representation of
the dynamic programming table dependencies, we note that even though we
move from two dimensions to four dimensions, we still have a topological order-
ing of the nodes that allows us to save checkpoints as described above. Instead of
columns in our two-dimensional $\omega$ table, we save checkpoint $t$-cubes, i.e. check-
points containing all cells in our table for a specific $t$ value. Using these cubes
as checkpoints, we can apply the same method as for the Viterbi algorithm.
For the restricted Viterbi, we retain the time complexity of $O(T \cdot N^2 \cdot u \cdot |Q|^2)$.
A $t$-cube takes up $O(N \cdot u \cdot |Q|)$ space and so applying the idea to the restricted
Viterbi gives us the improved space complexity $O(\sqrt{T} \cdot N \cdot u \cdot |Q|)$.

### 3.2.2 Approximate linear space

Instead of looking at guaranteed improvements to worst case space consump-
tion, another approach is to look at methods which guarantee no improvement,
but perform very well in practice. For such improvements to be viable, they
should perform better in practice than the improvements that make guarantees.
One approach is the *approximate linear space (ALS)* method of Churbanov et al. [4],
where the idea is to only save information that may contribute to finding
a most likely sequence of hidden states.

Recall the definition and calculation of the $\omega$ table and hidden state sequence
as described in section 2.2.3. Now, a slightly different approach to backtracking
the \( \omega \) table to find the hidden state sequence is to store backtracking pointers in a separate table \( \Psi \). We can calculate these pointers when we calculate the corresponding entry in the \( \omega \) table. This enables us to only save the latest two columns of the \( \omega \) table as we fill out the table, as can be seen in figure 3.2. However, it does not improve the space complexity since we still need to store the table of backtracking pointers. We can fill out the \( \Psi \) table with base case

\[
\Psi_1(z_1) = 0
\]

and recursion case

\[
\Psi_t(z_t) = \arg \max_{z_{t-1}} \omega_{t-1}(z_{t-1})a_{z_{t-1}, z_t} \text{ for } t = 2, \cdots, T
\]

Instead of any recalculation, we can now simply find the most likely hidden state path \( Z^* \) as

\[
z_T^* = \arg \max_{z_T} \omega_T(z_T)
\]

\[
z_t^* = \Psi_{t+1}(z_{t+1}) \text{ for } t = T - 1, \cdots, 1
\]

Now, if we look at the entries \( \Psi_t(z_t) \) for some value of \( t \) and \( z_t \), their entries hold the value of a previous hidden state entry which gave rise to the maximum of the given entry. The main observation here is that only some of the previous hidden state entries give rise to a maximum in the next column. The backtracking information of those that do not give rise to any maximum do not need to be saved. If we apply this idea recursively, we can avoid storing a lot of the backtracking information.

Churbanov et al. \cite{4} observe that backtrack paths typically converge to the most likely state path and because of that, they achieve a space consumption linear in \( T \). This observation is of course dependent on the HMM topologies in use. In section 4.4 we examine the improvement of the space consumption in the context of our simple gene prediction HMM model.

**Implementing the ALS method**  As our main approach is to improve the practical space consumption, we have chosen to implement the idea of Churbanov et al. \cite{4}.
Implementing the backtracking table with a multidimensional array based approach makes it impossible to take advantage of the observation that we do not need to save all the backtracking information as described above. Instead, if we model the backtracking paths using linked lists, we can use the reference count to find the unreferenced nodes, which do not give rise to any maximum and so can never be a part of a most likely hidden path. The idea is to create a linked list node for each entry that would have been in the table and make the backtracking pointers be actual pointers and not just integers representing the previous hidden state. Let $\psi_t(z_t)$ denote the linked list node corresponding to visiting hidden state $z_t$ at time $t$. We do not have a table, but we keep a reference to the latest two columns of the nodes as we calculate our $\omega$ table for increasing values of $t$. As we create and link these nodes, we know the value of both $z_t$ and $t$, but since, in the end, we are not referencing the nodes from a table, we have no way to read off the value of $z_t$ or $t$. We can calculate the value of $t$ from the number of pointers in the linked list since a node $\psi_t(z_t)$ always points to a previous node $\psi_{t-1}(z_{t-1})$ or NULL for $t = 0$. To know which hidden state a node represents, we need to store the hidden state number in the node.

Recall that $\psi_t(z_t)$ denotes the linked list node corresponding to visiting hidden state $z_t$ at time $t$. Let $\psi_t(z_t) \rightarrow \text{prev}$ denote the node's reference to its previous node and $\psi_t(z_t)[\text{val}]$ denote its hidden state value. We can create and link the linked lists with base case

$$
\psi_1(z_1) \rightarrow \text{prev} = \text{NULL} \\
\psi_1(z_1)[\text{val}] = z_1
$$

and recursion case.
\[
\psi_t(z_t) \rightarrow \text{prev} = \psi_{t-1} \left( \arg \max_{z_{t-1}} \omega_{t-1}(z_{t-1}) a_{z_{t-1}, z_t} \right) \quad \text{for } t = 2, \cdots, T \\
\psi_t(z_t)[\text{val}] = z_t \quad \text{for } t = 2, \cdots, T
\]

Denote the linked list nodes of the most likely hidden state path \( z^*_1, \cdots, z^*_T \) as \( \text{node}_1, \cdots, \text{node}_T \). We have a reference to the last two columns and so have a reference to \( \psi_T(z_T) \). We can then find the most likely hidden state path \( Z^* \) simply by following the pointers and reading off the value of the hidden state

\[
z^*_T = \arg \max_{z_T} \omega_T(z_T) \\
\text{node}_T = \psi_T(z^*_T) \\
\text{node}_t = \text{node}_{t+1} \rightarrow \text{prev} \quad \text{for } t = T - 1, \cdots, 1 \\
z^*_t = \text{node}_t[\text{val}] \quad \text{for } t = T - 1, \cdots, 1
\]

Now, whenever the reference count of a linked list node decreases to zero, we know that it does not give rise to any maximum and so can never be a part of a most likely hidden path. We can then deallocate the object from the heap and recursively decrease the reference count of all nodes that the deallocated object references and so forth. When this is implemented in a managed language with a garbage collector, we do not have to keep manual reference counts. Instead, the nodes are automatically garbage collected along the way as their internal reference counts reach zero.\(^3\)

---

\(^3\)We have verified this in our own implementation using the Java VisualVM profiler.

Figure 3.3: Storing backtracking information using linked lists. Dots represent nodes and lines represent pointers. Nodes that are not referenced through pointers from the two \( \psi \) columns can be garbage collected. These nodes are depicted with smaller dots and dashed lines for pointers.
We calculate the same amount of cells as in the Viterbi algorithm, so the time complexity remains the same. As we deallocate nodes, we expect a practical space reduction, but in the worst case, all nodes give rise to a maximum and so no node objects are deallocated. Therefore the space complexity also stays the same.

**Implementation for the restricted Viterbi**  As with the checkpoint algorithm, this method is also applicable to the restricted Viterbi algorithm. For the table based backtracking approach, backtracking the $t$ dimension gives us a $(z_t, k, q)$-triple for each $t = T, T - 1, \cdots, 1$ from where we can read off the most likely hidden state sequence $Z^*$.  

Let us return to the linked list approach that enables us to save space. We start by reformulating $\psi$ for the restricted Viterbi.  First, to be able to make the notation suffice, we introduce the $\delta_t$ function that returns a $(z_{t-1}, k, q')$ pair

$$
\delta_t(z_t, k, q) = \arg \max_{z_{t-1}, q' \in Q'(z_t, q)} a_{z_{t-1}, z_t} \tilde{\omega}_{t-1}(z_{t-1}, k, q')
$$

(3.13)

We use the notation $\pi_1(\delta_t(z_t, k, q))$ to select the first component, $z_{t-1}$, of the pair and $\pi_2(\delta_t(z_t, k, q))$ to select the second component, $q'$, of the pair.

Now we can calculate the base case

$$
\psi_1(z_1, k, q) \rightarrow \text{prev} = \text{NULL}
\psi_1(z_1, k, q)[\text{val}] = z_1
$$

(3.14)

and recursion case

$$
\psi_t(z_t, k, q) \rightarrow \text{prev} = \begin{cases} 
\psi_{t-1} \left( \pi_1(\delta_t(z_t, k, q)), k, \pi_2(\delta_t(z_t, k, q)) \right) & \text{if } q \notin A \\
\psi_{t-1} \left( \pi_1(\delta_t(z_t, k-1, q)), k-1, \pi_2(\delta_t(z_t, k-1, q)) \right) & \text{if } q \in A \land k > 0 \\
\text{NULL} & \text{if } q \in A \land k = 0 
\end{cases}
$$

(3.15)

Remember that we restrict the number of regular expression occurrences to be $k \in [l, u]$ where $l$ is the lower bound and $u$ is the upper bound. We find our first node by locating the $(z_T, k \in [l, u], q)$ triple that has the maximum $\tilde{\omega}$ value. Again, we denote the linked list nodes of the most likely hidden state path $z_1^*, \cdots, z_T^*$ as $\text{node}_1, \cdots, \text{node}_T$.

$$
\text{node}_T = \psi_T \left( \arg \max_{(z_T, k \in [l, u], q)} \tilde{\omega}_T(z_T, k, q) \right)
$$

The most likely hidden state path can, as with the basic Viterbi, be found by following the pointers and reading off the hidden state values.
We calculate the same amount of cells as in the table based restricted Viterbi algorithm so the time complexity remains the same. Again, as we deallocate nodes along the calculation, we expect a practical space reduction. In the worst case, all nodes give rise to a maximum and so no node objects can be deallocated. Because of that, the space complexity also stays the same.

In algorithm 6 we present pseudocode for the restricted Viterbi algorithm which uses the ALS method to improve the space consumption. The base case on lines 1-14 corresponds very closely to the base case of the standard restricted Viterbi algorithm. The difference is that because we store backtracking information along the way, we only need table values for the two most recent values of $t$ and so we move from a four dimensional table to a three dimensional table as we leave out the $t$-dimension. The two most recent $t$-cubes are stored in $\hat{\omega}_t$ and $\hat{\omega}_{t-1}$ and their corresponding linked list nodes used for backtracking is stored in $\psi_t$ and $\psi_{t-1}$ respectively.

The method NEW-NODE($z$) creates a new linked list node with value $z$ and previous pointer set to NULL. As we initialize $\hat{\omega}$ entries on lines 10 and 13, we also create corresponding linked list nodes. Entries where $\hat{\omega}$ has value $-\infty$ have no need for a corresponding linked list node, since it will never be pointed to.

Algorithm 6 RESTRICTED-VITERBI-ALS (1/2)

Require: $N, \Pi, A, B, \text{regexp}, X$

Ensure: $\hat{\omega}_t$ holds the final $t$-cube of a complete table, $\psi_t$ holds linked lists for backtracking the corresponding $\hat{\omega}$ entries

1: $\text{dfa} \leftarrow \text{CONSTRUCT-DFA}(N, \text{regexp})$
2: $Q \leftarrow \text{states}[\text{dfa}]$
3: for $z \leftarrow 0 \text{ to } N - 1$ do
4: \quad $\text{val} \leftarrow \log(\Pi[z]) + \log(B[z][X[0]])$
5: \quad for $q \leftarrow 0 \text{ to } |Q| - 1$ do
6: \quad \quad if TRANSITION-EXISTS($\text{initial}[\text{dfa}], z, q$) then
7: \quad \quad \quad $\hat{\omega}_t[z][1][q] \leftarrow \text{val}$
8: \quad \quad \quad $\psi_t[z][1][q] \leftarrow \text{NEW-NODE}(z)$
9: \quad \quad else
10: \quad \quad \quad $\hat{\omega}_t[z][0][q] \leftarrow \text{val}$
11: \quad \quad \quad $\psi_t[z][0][q] \leftarrow \text{NEW-NODE}(z)$

42
The recursion case of the ALS restricted Viterbi algorithm (lines 15-50) resembles a combination of the standard restricted Viterbi algorithm and its corresponding backtracking algorithm. This is natural, since we calculate and store both the $\hat{\omega}$ values and the $\psi$ backtracking information in the same pass. Whenever we increment the value of $t$, on line 16, $\hat{\omega}_{t-1}$ is set to point to $\hat{\omega}_t$ and $\hat{\omega}_t$ is initialized with a new array. On line 17, the same is done for $\psi$. An alternative approach would be to use modulus for indexing, which we will
see in section 3.4 on multiple emission hidden Markov models. Lines 18-46 are very similar to the normal restricted Viterbi algorithm. When we find the maximum on lines 24-46, we also save the backtracking information, i.e. what \((z, k, q)\) entry gave rise to the maximum. On line 47, we calculate and store the maximum in the \(\hat{\omega}\) entry as in the standard restricted Viterbi. Now, if the \(\hat{\omega}\) value is \(-\infty\), it will never itself give rise to a maximum. Otherwise, on line 49, we create a linked list node and, on line 50, update it to point to the linked list node of the entry which gave rise to its maximum.

Note that this pseudocode does not manage reference counts and therefore assumes a managed language with a garbage collector, like Java. When the \(\psi_{t-1}\) array is set to point to a new set of nodes, all the old nodes, which do not give rise to any maximum, will get a reference count of zero. These nodes will be garbage collected by the virtual machine at some appropriate point in time.

Algorithm 7 RESTRICTED-VITERBI-ALS-BACKTRACK

Require: \(N, \hat{\omega}_t\) holds the final \(t\)-cube of a complete table, \(\psi_t\) holds linked lists for backtracking the corresponding \(\hat{\omega}\) entries

Ensure: \(trace_z[0], \ldots, trace_z[T-1]\) is a most likely sequence of hidden states

1: \(\text{max} \leftarrow -\infty\)
2: for \(z \leftarrow 0\) to \(N - 1\) do \(\triangleright\) Base case
3: \(\text{for } k \leftarrow l\) to \(u\) do
4: \(\text{for } q \leftarrow 0\) to \(|Q| - 1\) do
5: \(\text{val} \leftarrow \hat{\omega}_{t}[z][k][q]\)
6: \(\text{if } \text{val} > \text{max} \text{ then}\)
7: \(\text{max} \leftarrow \text{val}\)
8: \(\text{max}_z \leftarrow z\)
9: \(\text{max}_k \leftarrow k\)
10: \(\text{max}_q \leftarrow q\)
11: \(\text{trace}_z[T-1] \leftarrow \text{max}_z\)
12: \(\text{node} \leftarrow \psi_{t}[\text{max}_z][\text{max}_k][\text{max}_t]\)
13: for \(t \leftarrow T - 2\) to \(0\) do \(\triangleright\) Recursion case
14: \(\text{node} \leftarrow (\text{node} \rightarrow \text{prev})\)
15: \(\text{trace}_z[t] \leftarrow \text{val}[:\text{node}]\)

The backtracking presented in algorithm 7 is simple. Recall that we define the final \(t\)-cube as the entries which have the largest \(t\)-value in our table. Now, when the space improved restricted Viterbi terminates, \(\hat{\omega}_t\) holds the final \(t\)-cube, which we need for finding the maximum entry to backtrack from. Also, \(\psi_t\) holds the corresponding linked list nodes. On line 1-12, we find the maximum entry and its corresponding linked list node. On lines 13-15 we follow the backtracking pointers and read off the hidden states to produce the trace.

Summary

In section 3.2 we have described two different alternatives for reducing the space consumption of the basic Viterbi algorithm. We have adapted the two
methods for use with the restricted Viterbi algorithm. For the approximate linear space method, we have also presented pseudocode along with relevant implementation considerations.

3.3 Improving the restricted forward parameter table size

An important observation when working with the restricted forward algorithm is that the maximum number of occurrences, \( k_{\text{max}} \), depends linearly on \( T \). Intuitively, a sequence of length \( 2T \) is able to contain roughly twice as many occurrences of some pattern as a sequence of length \( T \). Any reasonable upper bound \( m \) for the number of occurrences then depends at least linearly on \( T \) as well, and so the time complexity of the restricted forward algorithm may actually be written as \( O(T^2 \cdot N^2 \cdot |Q|^2) \) instead.

This quadratic dependency on \( T \) is especially undesirable, since we want to be able to increase \( T \) drastically for real gene prediction applications. Thus, we find the challenge of minimizing the size of the occurrence dimension of the forward table very important in order for the algorithms to be relevant in practice. Note that in the restricted Viterbi algorithm, this issue is not as significant, since we can directly limit the size of the occurrence dimension by choosing some reasonable value of \( u \), e.g. based on results of the restricted forward algorithm.

In our implementation of the restricted forward algorithm, section 3.1.1 we chose a relatively simple but general approach to calculate an upper bound \( m \) for the number of pattern occurrences \( k \) in a string. In retrospect this method does not work well with our particular choice of regular expression, \( r = (N.N|C_1|R_3)|((C_3|R_1).N.N) \). By considering the sequence fragment \( C_3NNC_1 \), it is evident that it is possible for two occurrences of \( r \) to be located within a distance of 1 nucleotide. In the DFA, this possibility appears as a transition between two accept states. Thus, the method for calculating \( m \) as presented in section 3.1.1 actually returns \( m \approx T \) when using the particular regular expression \( r \).

Given a particular regular expression, it is generally not hard to calculate \( k_{\text{max}} \) as a function of \( T \). For instance for our choice of model and \( r \), \( k_{\text{max}} = 0.4 \cdot T \) when ignoring rounding issues. To see this, consider a legal sequence maximizing the number of occurrences, i.e. \( \cdots NNC_1C_2C_3NNC_1C_2C_3 \cdots \), which has 2 occurrences per 5 nucleotides. Thus, if we wanted to limit our algorithm to only use \( r \) as a regular expression, we could simply calculate \( k_{\text{max}} \) as described and choose that value as \( m \). For the restricted algorithm to be versatile, however, we need a general method for calculating an upper bound of occurrences, and a better method than the simple one in section 3.1.1 would be desirable.

In this section, we will consider different approaches of improving the method from section 3.1.1 for decreasing the size of the occurrence dimension of the table. First, we will discuss how to achieve an improved theoretical upper bound of the number of occurrences. Second, we will look into details at a more practical and aggressive method for decreasing the size of the \( k \)-dimension. The
presented ideas of both parts have emerged during discussions of the subject and are not based on external literature.

3.3.1 Theoretical considerations

As mentioned above, the method for calculating $m$ used in section 3.1.1 can produce upper bounds that are much larger than $k_{\text{max}}$. The reason for this is the simplicity of the method. It assumes that the maximum number of occurrences corresponds to continuously following DFA paths of $\min_{i,j}(\text{dist}_{ij})$ steps to the next accept state, where $\text{dist}_{ij}$ is the distance from accept state $a_i$ to accept state $a_j$. However, when visiting some accepting state, it is entirely possible that it has no outgoing path of length $\min_{i,j}(\text{dist}_{ij})$ to another accepting state, which may result in an upper bound $m$ much larger than the actual $k_{\text{max}}$.

We have considered how the method can be improved, if we keep the emphasis on the DFA and ignore possible HMM contributions due to illegal transitions, as discussed in section 3.1.1. A natural goal is to calculate the maximum number of traversed accept states in a path of length $T$ starting at the initial state of the DFA, since such a path corresponds to a hidden sequence with the maximum number of occurrences.

In order to find such a path, we can build a directed, weighted graph $G = (V, E)$ containing relevant information of the DFA. This relevant information consists of the accept states, the initial state, and shortest path distances between these states. An example of such a graph, for a DFA with three accept states and a separate initial state, is shown in figure 3.4.

![Figure 3.4: A graph consisting of DFA accept states $a_1$, $a_2$, $a_3$, and initial state $q_{\text{init}}$, with edge lengths corresponding to the number of DFA steps between the states. Edges are omitted where the corresponding path in the DFA passes through another accept state.](image)

The problem is now finding a path of length $T$ in this graph starting in $q_{\text{init}}$ and maximizing the number of traversed vertices. For sufficiently large $T$, it
can be shown that this problem can be reduced to a simple graph theoretical problem: finding a cycle \( c \) in a directed, weighted graph minimizing the ratio \( \frac{c_n}{c_{\text{dist}}} \), where \( c_n \) is the number of vertices in the cycle and \( c_{\text{dist}} \) is the total weight of one round in the cycle. This graph theoretical problem can be solved in time \( O(|V| \cdot |E|) \) using the algorithm presented in [11].

For the graph in figure 3.4, the edges of the cycle with the described property have been emphasized. An upper bound for the number of occurrences for the original problem then corresponds to following this cycle, resulting in \( c_n \) (here 3) occurrences per \( c_{\text{dist}} \) (here 5) symbols in the sequence.

In the general case, the results of such an algorithm are very interesting. Because we ignore illegal HMM transitions, it might not be able to calculate \( m = k_{\text{max}} \), but it is an improvement over the simple approach of section 3.1.1. However, when considering our practical area of application, theoretical approaches in general fall short. The reason for this is that \( k_{\text{max}} \) corresponds to closely packed pattern occurrences with as much overlap as possible. To consider overlapping occurrences when modelling gene boundaries is not relevant, and so continuing on a purely theoretical path is undesirable.

An approach is to alter the DFA to not allow overlapping occurrences, by resetting it every time it reaches an accept state, as described by Tataru et al. [20]. As a consequence, \( k_{\text{max}} \) would decrease and we would be able to choose as upper bound \( m = \frac{T}{|r|} \), where we define \( |r| \) as the minimum length of a sequence matching the regular expression. In our case, we have \( |r| = 3 \), and so the size of the occurrence dimension of our table would become \( m = \frac{T}{3} \).

However, depending on our application, this upper bound might also be far from realistic occurrence numbers, since it corresponds to as closely packed non-overlapping occurrences as possible.

If we want to decrease the size of the occurrence dimension of the restricted forward table further, we need to consider the problem from a more pragmatic standpoint.

### 3.3.2 The cut off method

As we have discovered in previous sections, it is possible to decrease the size of the \( k \)-dimension by theoretical approaches. This has a direct effect on the time consumption of the restricted forward algorithm.\footnote{The same cannot be said about the space consumption when we apply the simple space improvement of section 3.1.1.} We have also discovered, however, that the maximum number of occurrences, \( k_{\text{max}} \), is simply impractically large, and increasingly so for greater values of \( T \). Our approach in this section will be to try reducing the size of the occurrence dimension in the restricted forward table based on practical examinations of such tables.

As hinted in the previous section, it is highly unlikely that a random sequence of considerable length \( T \) contains the maximum possible number of occurrences of some regular expression \( r \). Actually, for a regular expression corresponding to gene boundaries in a DNA sequence, we expect the most likely number of occurrences to be so much smaller than the theoretical maximum that the size of this maximum is of no practical relevance at all.
Indeed, looking at some of our $\hat{\alpha}$ tables in practice reveals that the values of the table generally become very close to zero when $k$ exceeds some relatively small value. Examples of actual $k$-distributions can be found in figures A.1 and A.2. Note that the $k$-occurrence probabilities in these figures are systematically higher for even $k$’s than for odd $k$’s. This is due to the choices of model and regular expression\textsuperscript{5} and is not a general property.

Figure 3.5: An outline of a typical $k$-distribution. In the figure, $P(k)$ is used as shorthand for $P(O_r(Z_{1:T}) = k|X_{1:T})$.

To better visualize our ideas, a generalized outline of a $k$-distribution is sketched in figure 3.5. As is apparent, the interesting part of the distribution is characterized by having low $k$-values, and the majority of the distribution contributes with only small probabilities. A reasonable idea, then, is to simply cut off part of $k$-distribution, as depicted in figure 3.6.

Figure 3.6: A $k$-distribution which is cut off at the value $k_{cutoff}$, by which the part of the distribution drawn in dotted line is discarded.

For the restricted forward table, this approach corresponds to disregarding a part of the table which only contains negligible values. Depending on the choice of the cut off value $k_{cutoff}$, this results in a much smaller table, thus improving time consumption significantly. Note that cutting off the table at some $k_{cutoff} < k_{max}$ does not change our ability of accurately calculating the probabilities of a smaller number of occurrences, since $k$ can never be decreasing in part of a valid path through the table.

Choosing a reasonable $k_{cutoff}$ value in advance is complicated from a computational standpoint, since calculating such a threshold requires knowledge

\textsuperscript{5}The model structure entails a higher probability of ending in a coding than a non-coding state, corresponding to an uneven number of gene boundaries.
of typical structures of DNA sequences. An algorithm using heuristics would likely be both very cumbersome and inaccurate. However, we do not actually need to choose $k_{\text{cutoff}}$ in advance. When filling out the $\hat{\alpha}$ table, the topological ordering we apply orders entries by using $t$-values as the primary parameter. It is not the only possible topological ordering though; there exists another ordering which instead of prioritizing the $t$-value prioritizes the $k$-value. In other words, this topological ordering orders entries with lower $k$-values before entries with higher $k$-values before looking at other parameters. For entries with equal $k$-values, the entry with the smaller $t$-value must then be ordered first. Such an ordering will then satisfy the property that any entry value depends only on previously ordered entry values, which enables the use of the topological ordering in filling out the $\hat{\alpha}$ table. Using this ordering in filling out the table means that we first fill out all entries with $k = 0$ (starting with $t = 0$ and then incrementing $t$), then fill out all entries with $k = 1$ etc.

The only missing part is when to stop, i.e. actually determining $k_{\text{cutoff}}$. Since we fill out entries where $t = T$ in each $k$-iteration, we can keep updating $\mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k)$, the probability of emitting the observable sequence with the number of occurrences limited to the current $k$-value. The simple observation

\[
\mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k) = \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k - 1) + \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) = k)
\]

means we can keep $\mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k)$ updated, based on the previous value and the final $t$-cube entry values we fill out in step $k$. An important observation is then that we can easily calculate the value $\mathbb{P}(X_{1:T})$ by using the basic forward algorithm, and that

\[
\lim_{k \to k_{\text{max}}} \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k) = \mathbb{P}(X_{1:T})
\]

This gives us the option to stop filling out the table after we have filled out entries in the final $t$-cube summing to some fraction $f_{\text{cutoff}}$ of what would be the total sum of entries in the final $t$-cube, that is, stop as soon as the following condition is satisfied:

\[
\mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k) \geq f_{\text{cutoff}} \cdot \mathbb{P}(X_{1:T})
\]

The rest of the table is then effectively cut off, making $k_{\text{cutoff}}$ equal to the smallest $k$ value satisfying this condition. A choice of $f_{\text{cutoff}}$ therefore entails a choice of $k_{\text{cutoff}}$ and vice versa.

We call this the cut off method, and when applying the method, we will refer to the restricted forward algorithm as the cut off restricted forward algorithm. It is an aggressive approach which can potentially decrease time consumption of
the algorithm drastically, since the time consumption increases with the number of table cells. However, when cutting off some of the \( k \)-probabilities, we effectively alter the distribution of \( k \). Consequently, whatever we extract from the distribution, e.g. the expectation or a 95\% interval, should be considered approximations of the actual values. In other words, the cut off method sacrifices some precision for reduced time consumption.\(^6\) We will examine this trade-off in section 4.5. To distinguish between the restricted forward algorithm and the cut off restricted forward algorithm, we will refer to the former as the \textit{standard} restricted forward algorithm.

\textbf{Implementation}

In practice, the cut off technique implies a change to the restricted forward algorithm corresponding to nesting the for-loops differently, such that the \( k \)-loop is the outermost one. At the end of each \( k \)-iteration, we then calculate \( \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k) \) and compare it to the pre-calculated threshold value \( f_{\text{cutoff}} \cdot \mathbb{P}(X_{1:T}) \), where \( f_{\text{cutoff}} \) is provided as a parameter and \( \mathbb{P}(X_{1:T}) \) is calculated using the basic forward algorithm. According to equation \(3.16\) as long as \( f_{\text{cutoff}} \) is less than 1, \( \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) \leq k) \) will at some point (when \( k = k_{\text{cutoff}} \)) exceed the threshold value. At this point we terminate the algorithm and return the \( k \)-distribution consisting of the probabilities \( \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) = 0), \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) = 1), \ldots, \mathbb{P}(X_{1:T}, O_r(Z_{1:T}) = k_{\text{cutoff}}) \).

The optimization of space consumption by only keeping two 3-dimensional \( t \)-tables in the memory at a time naturally cannot be employed in the cut off implementation. Instead, we can optimize the space consumption by letting the two 3-dimensional tables correspond to two subsequent \( k \)-values. This is possible due to the fact that, cf. equation \(3.2\), we never need to look back more than one \( k \)-value to calculate an entry value. This change causes the space complexity of the cut off restricted forward algorithm to be \( O(T \cdot N \cdot \left| Q \right|) \) instead of \( O(N \cdot m \cdot \left| Q \right|) \) as in the case of the standard restricted forward algorithm.

A practical implementation as described seems straightforward. Indeed, it can be applied to the \( \tilde{\alpha} \) table directly. For the numerically stable \( \hat{\alpha} \) table however, it is not that simple, since our scaling technique requires knowledge of completely filled-out \( t \)-cubes, that is, it requires values of the very entries that we have established the cut off algorithm to avoid calculating. In order for the cut off approach to be practically feasible, we need another way of achieving numerical stability.

A reasonable approach is to adjust the scaling technique to fit the cut off algorithm. In order to do this, we want to define and apply scaling parameters such that after each \( k \)-iteration, all stored entry values are numerically stable and entries with the same \( t \) value have been scaled by the same known amount. We have considered two ways of modifying the scaling technique. In the following, we will refer to the scaled cut off table as \( \hat{\alpha}' \).

In the first approach, we intend to end up with a filled out cut off \( \hat{\alpha}' \)-table scaled such that each group of entries with equal \( t \)-value sums to 1, similarly

\(^6\)Note that the reduced time consumption does not show in the theoretical time complexity, since any reasonable choice of \( k_{\text{cutoff}} \) value increases linearly as \( T \) increases.
to the standard $\hat{\alpha}$-table. Contrary to the $\hat{\alpha}$-table however, we have no way of properly scaling $k$-entries during some $k$-iteration in such a way that a $t$-cube of the $\hat{\alpha}'$-table will end up summing to 1 once all its entries have been scaled in the same way. We can only perform the appropriate scaling once the whole table is filled out. Simultaneously, we need to keep entries scaled while iterating through $t$-values, or numerical instability problems will arise. An idea is then to perform the scaling as if the current $k$-iteration is the last. In other words, when calculating each group of entry values with some particular $t$- and $k$-value, we can perform the scaling such that the following invariant holds:

$$
\sum_{k'=0}^{k} \sum_{z_t, q} \hat{\alpha}_{t}(z_t, k', q) = 1
$$

(3.17)

Naturally, this requires rescaling all previously filled out $t$-entries during step $t$ in each $k$-iteration in order to keep the entries in the $t$-cube scaled by an equal amount. In practice, only the $t$-cube entries from the previous $k$-iteration need to be rescaled, since these are the only ones we use again. Each value of the probability distribution can then be calculated at the end of the corresponding $k$-iteration as the product of the current $c$-values and the sum of the entries at $k$ and $T$.

We have implemented this scaling method and verified that it produces expected results while removing the numerical stability problems. However, another way of scaling the entries exists that gets rid of the time-consuming rescaling of entries and updating of $c$-values. This approach makes use of the basic forward algorithm, which we already use to get the observable sequence probability. Implementing the basic forward algorithm, one is faced with numerical stability issues as well, and the obvious solution is to use the same scaling technique as described for the restricted forward algorithm in section 3.1.1. In other words, a forward algorithm using scaling for achieving numerical stability would calculate $c$-values such that

$$
P(X_{1:t}) = \prod_{t'=1}^{t} c_{t'}
$$

The crucial observation is that the $c$-values of the forward and the restricted forward algorithm are actually the same. This is not surprising, since the restricted forward algorithm is just a more fine-grained version of the forward algorithm, keeping track of DFA states and number of pattern occurrences. Unscaled entries of equal $t$-value therefore sum to $P(X_{1:t})$ in both algorithms. Thus, we already have an inexpensive way of calculating the $c$-values of the standard restricted $\hat{\alpha}$-table. At first sight, this might seem irrelevant, since $P(X_{1:t})$ is not equal to the sum of $t$-entries in a complete cut off $\hat{\alpha}'$-table, and so the $c$-values cannot be used to scale the $t$-entries into summing to 1. However, they actually do not need to sum to 1; all that is needed is enough scaling to achieve numerical stability while being able to recreate the actual probabilities.
Consider using the forward \(c\)-values (equal to the restricted forward \(c\)-values) for scaling all entries in the \(\hat{\alpha}'\)-table. Since the unscaled value of any entry for obvious reasons is the same as in the standard table, the scaled value will then become the same as well. In other words, the \(\hat{\alpha}'\)-table along with its \(c\)-values will become an exact copy of the \(\hat{\alpha}\)-table, except for the fact that part of the table has been cut off. This way, we can skip \(c\)-value calculation and rescaling altogether in the cut off restricted forward algorithm. This approach could be used in the restricted forward algorithm as well, although the benefits are not nearly as apparent.

We have implemented this scaling method and verified that the \(k\)-distribution it calculates is indeed merely a cut off version of the distribution calculated by the previous implementation of the restricted forward algorithm (cf. section 3.1.1). Due to its simplicity and smaller practical time consumption, this is our preferred scaling method for the forward cut off algorithm.

Pseudocode for the complete forward cut off algorithm is shown in algorithm 8.

**Algorithm 8 K-CUTOFF-FORWARD (1/2)**

**Require:** \(N, \Pi, A, B, \mathit{regexp}, X, f_{cutoff}\)

**Ensure:** Return the occurrence probability distribution cutoff at the fraction \(f_{cutoff}\)

1: \(\text{dfa} \leftarrow \text{CONSTRUCT-DFA}(N, \mathit{regexp})\)
2: \(Q \leftarrow \text{states}[\text{dfa}]\)
3: \(c \leftarrow \text{FORWARD-CVALUES}(N, \Pi, A, B, X)\)
4: \(\text{totalCProd} \leftarrow 1\)  \(\triangleright \) Calculate occurrence probability
5: \(\text{for } t \leftarrow 0 \text{ to } T - 1 \text{ do}\)
6: \(\text{totalCProd} \leftarrow \text{totalCProd} \cdot c[t]\)
7: \(f_{\text{current}} \leftarrow 0\)
8: \(k \leftarrow 0\)
9: \(\hat{\alpha}_k \leftarrow \text{new zero-initialized table}\)
10: \(\text{while } f_{\text{current}} < f_{\text{cutoff}} \text{ do}\)
11: \(\hat{\alpha}_{k-1} \leftarrow \hat{\alpha}_k\)
12: \(\hat{\alpha}_k \leftarrow \text{new zero-initialized table}\)
13: \(\text{for } z \leftarrow 0 \text{ to } N - 1 \text{ do}\)
14: \(\text{if } \text{TRANSITION-EXISTS}([\text{dfa}], z, q) \text{ then}\)
15: \(\text{else if } k = 0 \text{ & } !\text{accept}[q] \text{ then}\)
16: \(\text{else if } k = 0 \text{ & } !\text{accept}[q] \text{ then}\)
17: \(\hat{\alpha}_k[0][z][q] \leftarrow \text{val}\)
18: \(\hat{\alpha}_k[0][z][q] \leftarrow \text{val}\)

In addition to the standard restricted forward input parameters, we require \(f_{\text{cutoff}}\) as a parameter so we are able to terminate when a satisfactory part of the \(k\)-distribution has been computed. Initially, in line 3, the scaling values of the basic forward algorithm are calculated and saved in an array named \(c\).
c. **FORWARD-CVALUES** is simply a slightly modified version of the basic forward algorithm which returns its calculated \( c \)-values. Since we now know the \( c \)-values, we can later just scale each entry right away when calculating it.

When iterating though the table cells, we have the \( k \)-loop as the outermost loop as mentioned. In practice, we implement it as a while-loop (line 10), such that we can stop when the specified fraction of the distribution is calculated. Contrary to the standard implementation, and as mentioned previously, we now keep two 3-dimensional \( \tilde{\alpha}_k \) tables in memory, each corresponding to a part of the \( \tilde{\alpha} \) with a certain \( k \)-value.

**Algorithm 8** K-CUTOFF-FORWARD (2/2)

```plaintext
c1: for \( t \leftarrow 1 \) to \( T - 1 \) do ▶ \( t > 0 \)
c2: for \( z_t \leftarrow 0 \) to \( N - 1 \) do
nc3: for \( q \leftarrow 0 \) to \( |Q| - 1 \) do
nc4: if \( \text{accept}[q] \& \ k = 0 \) then
nc5: \( \tilde{\alpha}_k[t][z_t][q] \leftarrow 0 \\
nc6: \text{continue}
nc7: outSum \leftarrow 0
nc8: \( Q' \leftarrow \text{FIND-PREDECESSORS}(z, q) \)
nc9: for \( z_{t-1} \leftarrow 0 \) to \( N - 1 \) do
nc10: inSum \leftarrow 0
nc11: for all \( q' \) in \( Q' \) do
nc12: if \( \text{accept}[q] \) then
nc13: inSum \leftarrow inSum + \( \tilde{\alpha}_{k-1}[t-1][z_{t-1}][q'] \)
nc14: else
nc15: inSum \leftarrow inSum + \( \tilde{\alpha}_k[t-1][z_{t-1}][q] \)
nc16: outSum \leftarrow outSum + inSum \cdot A[z_{t-1}][z_t]
nc17: \( \tilde{\alpha}_k[t][z_t][q] \leftarrow \text{outSum} \cdot B[z_t][X[t]]/c[t] \quad \text{Scaled value} \\
nc18: scaledLastSum \leftarrow 0
nc19: for \( z \leftarrow 0 \) to \( N - 1 \) do
nc20: for \( q \leftarrow 0 \) to \( |Q| - 1 \) do
nc21: scaledLastSum \leftarrow scaledLastSum + \( \tilde{\alpha}_k[T-1][z][q] \)
nc22: \( f_{\text{current}} \leftarrow f_{\text{current}} + \text{scaledLastSum} \\
nc23: \text{occurrenceProbabilities}[k] \leftarrow \text{totalCProd} \cdot \text{scaledLastSum} \\
nc24: k \leftarrow k + 1
nc25: \text{return occurrenceProbabilities}
```

At the end of each \( k \)-iteration, in line 38-41, we calculate the sum of the \( k \)-entries with \( t = T \). This sum is then used for updating the currently filled out fraction of \( f_{\text{current}} \) of the \( k \)-distribution, and for setting the \( k \)'th occurrence probability. Once \( f_{\text{current}} \) exceeds \( f_{\text{cutoff}} \), the algorithm returns the cut off \( k \)-distribution.

**Summary**

In section 3.3 we have considered different approaches for improving the size of the occurrence distribution of the restricted forward algorithm table. Theoret-
ical approaches were considered, but ultimately the practical cut off approach proved most effective. We have described implementation choices for the cut off restricted forward algorithm, and we will examine its induced trade-off between accuracy and time consumption in section 4.5.

3.4 Multiple emission hidden Markov models

The main idea in this section has emerged in discussions with our thesis advisor, and the content is not based on external literature.

In the hidden Markov model formalism we have presented in this thesis, every hidden state emits one symbol and so the sequence of observables are produced from a sequence of hidden states of the same length. In the case of gene prediction, we have some extra knowledge of the typical structure of the models. We model structure like start codons, stop codons, coding regions, non-coding regions and reading frames. As an example let us return to the very simple gene prediction model on figure 2.7. Coding regions are sequences of codons and so have lengths dividable by three as is apparent also in the simple model, where coding regions in forward and reverse direction are modelled with states $C_1, C_2, C_3$ and $R_1, R_2, R_3$ respectively. Now, the three different states allow us to distinguish the frequency of the nucleotides in coding regions with respect to their position in the codon, i.e. first, second or third nucleotide in the codon. However, we do not have probabilities for possible codons, only for the separate nucleotides. So there is room for a more fine grained model.

Another observation is that the transition between hidden states are deterministic inside the part of the model that models these coding regions, e.g. if we transition to state $C_1$ it is certain that we transition to $C_2$ and $C_3$ in the next two steps since those transitions have probability one in our model. The time and space complexity of the forward and Viterbi algorithms depend on the number of hidden states, $N$, in the model, so it is desirable to reduce the number of states with these deterministic transitions.

To address the above problems, we consider a more general form of hidden Markov model where every hidden state, $z$, can emit a fixed, constant number of observable symbols, $\phi_z$. We call this kind of model a multiple emission hidden Markov model (MEHMM). In this model, the states with deterministic transitions can be merged into a single state and so the total number of hidden states, $N$, will become smaller. For example, in our simple model, the codon represented by $C_1, C_2, C_3$, can be merged into a resulting state, $C$, which will emit three symbols and so has $\phi_C = 3$.

For every hidden state $h_i$, we have an emission probability for every combination of emission symbols of length $\phi_{h_i}$. The size of the emission table, $B$, will vary according to the values of $\phi_{h_i}$. The first dimension of the table will be of length $N$, and for every choice of hidden state $h_i$, we have an additional $\phi_{h_i}$ dimensions of length $M$, i.e. $M \times \cdots \times M$, where $M$ is the number of output symbols. Revisiting the example, our resulting state $C$ will get $M \times M \times M = 4 \times 4 \times 4 = 64$ free parameters as opposed to $4 + 4 + 4 = 12$ free
parameters in the three states $C_1, C_2, C_3$. These extra free parameters allow us a separate probability for every possible codon, making the model more fine grained.

If we let the maximum $\phi$ value be denoted by $L = \max_{h_i} \phi_{h_i}$, the space complexity of the emission table $B$ will then become $O(NML)$. We define a multiple emission hidden Markov model formally as follows:

A multiple emission hidden Markov model is a hextuple $\lambda = (H, O, \Pi, \phi, A, B)$, where

$H$ is a finite set of hidden states $h_1, h_2, \ldots, h_N$

$O$ is a finite alphabet of output symbols $o_1, o_2, \ldots, o_M$

$\Pi$ is a vector $(\pi_1, \pi_2, \ldots, \pi_N)$ where $\pi_i$ is the probability of the hidden sequence starting in state $h_i$

$\phi$ is a vector $(\phi_1, \phi_2, \ldots, \phi_N)$ where $\phi_i$ is the number of output symbols emitted by hidden state $h_i$

$A$ is a $N \times N$ matrix where each entry $a_{i,j}$ is the probability of a state transition from $h_i$ to $h_j$

$B$ is a vector of size $N$ containing $M \times \cdots \times M$ matrices, where each $B$ entry $b_{i,j_1,\ldots,j_w}$ is the probability of emitting output symbols $o_{j_1}, \ldots, o_{j_w}$ from the state $h_i$ and where $w = \phi_{h_i}$.

Note that the sequence of hidden states no longer needs to be of length $T$, since hidden states can output multiple output symbols. For MEHMMs, we will use the terminology such that the hidden state corresponding to some observable state $x_t$ is denoted $z_t$. In other words, the subscript of both $x$ and $z$ relates to the corresponding observable sequence position, and $t$-values denote positions in the observable sequence only.

Because an MEHMM is a generalization of an HMM, we can express an HMM in terms of our MEHMM definition. For HMMs, the $N$ length $\phi$ vector should contain the value 1 in all positions, which corresponds to every state emitting one observable symbol. This will result in the emission table getting the fixed dimensions as for HMMs and the $z$ subscripts to also denote index positions into the sequence of hidden states, $Z$. The algorithms introduced here and in the following sections can therefore be seen as extended versions of already presented algorithms.

**Implementation**  Obviously, the implementation of the emission table must differ from that of HMMs. In MEHMMs, it consists of an array of length $N$, where each entry contains a $\phi$-dimensional array. To handle arrays of variable dimensions, we have created a wrapper-class, which behaves like a multidimensional array by internally storing a one-dimensional array and then translating between indices in the multidimensional array and an index in the one-dimensional array using Horner’s method\(^7\).

\(^7\)Horner’s method for polynomial computation can also be used for computing indices, e.g. $i = i_1 + \text{dim}_1(i_2 + \text{dim}_2(\cdots))$
3.4.1 The forward and Viterbi algorithms

To give an introduction to the ideas behind our implementation of the restricted algorithms in MEHMM context, we will start off by adjusting the basic forward and Viterbi algorithm for use with MEHMMs.

The forward algorithm Let us repeat the definition of the forward \( \alpha \)-parameter (cf. chapter 2):

\[
\alpha_t(z_t) = P(x_1 \cdots x_t, z_t)
\]

In the MEHMM case, this probability can be split up into individual probabilities of having emitted the first \( t - \phi_{z_t} \) symbols of \( X \) and being in any state \( z_{t-\phi_{z_t}} \) and then making the transition into \( z_t \) and emitting \( x_{t-\phi_{z_t}+1} \cdots x_t \). Formally,

\[
\alpha_t(z_t) = P(x_{t-\phi_{z_t}+1} \cdots x_t | z_t) \sum_{z_{t-\phi_{z_t}}} \alpha_t(z_{t-\phi_{z_t}}) P(z_t | z_{t-\phi_{z_t}})
\]

Thus, for calculating the forward parameter in the MEHMM case, we can again make use of previous columns of the table. Filling out the first table entries must be done a bit differently, though, as illustrated in figure 3.7.

![Figure 3.7: Filling out an \( \alpha \)-table for an MEHMM with \( N = 5 \) and \( \phi = [2, 1, 3, 3, 2] \). Table cells marked \( b \) are part of the base case, while cells marked \( r \) are part of the recursive case.](image)

Specifically, for some \( z_t \), if \( t \) is less than \( \phi_{z_t} \), the entry represents an impossible event: that \( z_t \) has emitted fewer than \( \phi_{z_t} \) observables. Such entries must therefore be assigned the value 0. In figure 3.7, the MEHMM has \( \phi = [2, 1, 3, 3, 2] \), which translates to one zero in the first row, no zeroes in the second row, two zeroes in the third row and so on.

In the case of a \( z_t \) entry with \( t \) equal to \( \phi_{z_t} \), the entry corresponds to having started the MEHMM in the hidden state \( z_t \) and emitted the first \( \phi_{z_t} \) observables from that state. This is of course possible and corresponds to the base case in the HMM \( \alpha \) table.

56
In the remaining case of \( z_t \) entries with \( t \) larger than \( \phi_{z_t} \), we are in the recursive case where the first \( t - \phi_{z_t} \) observables have been emitted before the state change to \( z_t \), from where the most recent observables \( x_{t-\phi_{z_t}+1} \cdots x_t \) are emitted. Like for HMMs, calculating such an entry value involves summing over all possible previous hidden states \( z_{t-\phi_{z_t}} \). However, since the \( \phi_{z_t} \)-value might differ for different \( z_t \)'s, the column that we sum over might differ as well. In figure 3.8 the values of two entries in a new column are about to be calculated, and the arrows from each entry point back at entries that could possibly contribute to the new entry values.

![Figure 3.8](image)

Figure 3.8: Recursively calculating entry values in the \( \alpha \)-table for an MEHMM with \( N = 5 \) and \( \phi = [2, 1, 3, 3, 2] \).

Even though the \( t \)-value of base entries vary (cf. figure 3.7), we can still calculate one full column at a time (cf. figure 3.7) since each entry value can only depend on entry values with lesser \( t \), and so we have a topological ordering that allows a column-wise filling of the table.

From these considerations, we can formulate \( \alpha \) as follows:

\[
\alpha_t(z_t) = \begin{cases} 
0 & \text{if } t < \phi_{z_t} \\
\pi_{z_t} b_{z_t,x_1 \cdots x_t} & \text{if } t = \phi_{z_t} \\
b_{z_t,x_{t-\phi_{z_t}+1} \cdots x_t} \sum_{z_{t-\phi_{z_t}}} \alpha_{t-\phi_{z_t}}(z_{t-\phi_{z_t}}) a_{z_{t-\phi_{z_t}}z_t} & \text{if } t > \phi_{z_t}
\end{cases}
\]

**The Viterbi algorithm**

Recall the definition of the Viterbi parameter

\[
\omega_t(z_t) = \max_{Z_{1:t-1}} P(X_{1:t}, Z_{1:t})
\]

Similar to the observations on the forward parameter, we note that a recursive formulation of the calculation of this parameter will no longer depend only on entries for \( t - 1 \), but on entries in the range \( [t - L, t - 1] \) where \( L = \max_{h_i} \phi_{h_i} \).

Figures 3.7 and 3.8 also apply to the Viterbi algorithm. Figure 3.7 shows how we make an uneven split (with regard to \( t \)) into base case and recursion case depending on the value of \( \phi_z \) for the different values of hidden state \( z \). Zero-entries will of course translate to \(-\infty\) in log space. Figure 3.8 shows how different values of \( \phi_z \) make the entries depend on different previous columns.
Because of the great similarity, the same considerations as with the forward algorithm applies for the Viterbi algorithm, since we are just doing maximizations instead of sums. We arrive at the following way to calculate the \( \omega \) table column-wise for increasing values of \( t \)

\[
\omega_t(z_t) = \begin{cases} 
-\infty & \text{if } t < \phi_{z_t} \\
\pi_{z_t} b_{z_t, x_1 \ldots x_t} & \text{if } t = \phi_{z_t} \\
b_{z_t, x_1 \ldots x_t} \max_{s=1} \omega_{s-1}(z_{s-1}) a_{z_{s-1}, z_t} & \text{if } t > \phi_{z_t}
\end{cases}
\]

where \( s = t - \phi_{z_t} + 1 \).

Having thus introduced the basic algorithms for MEHMMs, we now turn to the restricted algorithms. We have developed restricted MEHMM algorithms by applying the ideas of this section to the previously implemented restricted algorithms. In the following sections we will present the theoretical and practical considerations that our implementations of restricted algorithms for MEHMMs are based upon.

### 3.4.2 The restricted forward algorithm

In this section, we will present the restricted forward algorithm for MEHMMs. Since it is very similar to the HMM version of the algorithm, we will focus on the differences between the two versions. Furthermore, an implementation of the algorithm, which makes use of the cut off technique as described in section 3.3.2, will be presented.

**Theory**

Similarly to equation 2.1, we define a restricted forward parameter for MEHMMs as

\[
\tilde{\alpha}_t(z_t, k, q) = \sum_{z_{1:t-\phi_z} : O_r(z_{1:t}) = k} P(X_{1:t}, Z_{1:t}, FA_H(r) = q)
\]

Note that, in accordance with our notation change described in section 3.4.2, the subscript of this forward parameter refers to an index in the observable sequence rather than the hidden sequence.

Similarly to equation 3.1, the base case, which here applies for entries where \( t = \phi_{z_t} \), can then be written as:

\[
\tilde{\alpha}_t(z_t, k, q) = \pi_{z_t} b_{z_t, x_1 \ldots x_t} \mathbb{1}(q_0 \in Q'(z_t, q)) \cdot \begin{cases} 
1 & \text{if } k = 0 \\
1 & \text{if } k = 1 \\
0 & \text{otherwise}
\end{cases} \quad (3.18)
\]

In the case of \( t < \phi_{z_t} \), we have that \( \alpha_t(z_t) = 0 \), since as mentioned, being in \( z_t \) while having emitted fewer than \( \phi_{z_t} \) symbols is not possible.
For the recursion case \((t > \phi z_t)\), equation 3.2 is updated to reflect that the number of symbols emitted from each hidden state now varies:

\[
\bar{\alpha}_t(z_t, k, q) = \begin{cases} 
  b_{z_t, x_t - \phi z_t + 1} \cdots x_t \sum_{z_{t-\phi z_t}} a_{z_{t-\phi z_t}, x_t - \phi z_t + 1} \sum_{q' \in Q'(z_{t-\phi z_t})} \tilde{\alpha}_{t-\phi z_t}(z_{t-\phi z_t}, k, q') & \text{if } q \notin A \\
  b_{z_t, x_t - \phi z_t + 1} \cdots x_t \sum_{z_{t-\phi z_t}} a_{z_{t-\phi z_t}, x_t - \phi z_t + 1} \sum_{q' \in Q'(z_{t-\phi z_t})} \tilde{\alpha}_{t-\phi z_t}(z_{t-\phi z_t}, k-1, q') & \text{if } q \in A \land k > 0 \\
  0 & \text{if } q \in A \land k = 0
\end{cases}
\]

(3.19)

The theoretical difference from the HMM version of the algorithm is minimal, the biggest differences being the existence of a \(\phi\)-vector and the requirement of looking back further than just the most recent \(t\)-cube in the \(\bar{\alpha}\)-table.

Thus, the \(\bar{\alpha}\) table has the same size as the HMM counterpart. As mentioned in section 3.4.1, we additionally need to save an emission table of size \(O(N \cdot M^L)\), and so the total space complexity becomes \(O(T \cdot N \cdot m \cdot |Q| + N \cdot M^L)\). However, for applications in gene prediction, \(M\) is the number of nucleotides, and we can assume that \(L\) corresponds to the length of a codon. Thus, for our purpose, the space complexity is reduced to \(O(T \cdot N \cdot m \cdot |Q|)\) for the standard restricted forward algorithm and \(O(T \cdot N \cdot |Q|)\) for the cut off restricted forward algorithm.

Regarding the time complexity, we note that looking up a value in the emission table takes a constant amount of time. The amount of work needed for calculating each table entry is therefore the same as that of the HMM restricted forward algorithm, and so the time complexity remains \(O(T \cdot N^2 \cdot m \cdot |Q|^2)\).

**Implementation**

For a naïve implementation, adjusting the HMM restricted forward algorithm to fit MEHMMs is fairly straight-forward based on the theory above.

We need to be able to look up entry values in the previous \(\max z \phi z_t \) \(t\)-cubes at any time. Thus, minimizing memory usage by storing only the current and previous \(t\)-cube is not enough. Instead, we need to store the last \(\max z \phi z_t \) \(t\)-cubes. In our implementation, we have done this by means of a size \(\max z \phi z_t \) array and using modulo \(\max z \phi z_t \) arithmetic for writing to and reading from the array. To avoid overwriting entries that we might need in such an array, it is important not to write into entries as their values are calculated, but only overwrite complete \(t\)-cubes at once.

The method that we used in section 3.1.1 for finding an upper bound for the number of occurrences needs a bit of adjustment as well. Previously, the method took the DFA and \(T\) as input, where \(T\) could be considered equal to the length of the hidden state sequence and thus the total number of DFA state transitions. In MEHMMs however, \(T\) can be much larger than the hidden state sequence length, so in order to avoid getting an excessively large upper bound on \(k_{max}\), we need to adjust the method. A way of implementing such an adjustment is to let the method take the \(\phi\)-table as a third parameter. Then, when counting a transition during the BFS, we can consider the distance equal
to $\phi_z$ instead of 1, where $z$ is the transition symbol. Hereby, we find the shortest distance to an accepting state measured in output symbols instead of hidden states.

With this modification, the method has been generalized to capture the concepts of MEHMMs. If we consider an HMM as merely an MEHMM with a $\phi$-table consisting of 1’s, the method becomes equivalent to the one described in section [3.1.1], and we can therefore replace the HMM method altogether. Furthermore note that, because of the similarities with the HMM table, we can apply the cut off method as described in section [3.3.2] to the MEHMM restricted forward algorithm directly.

Scaling the table entries can be done in the same way as for HMMs, i.e. such that
\[
\sum_{z_t,k,q} \tilde{\alpha}_t(z_t, k, q) = 1 \tag{3.20}
\]
This can be achieved in a similar way as for HMMs, by calculating $c_t$-values and scaling as we iterate through $t$-cubes. We need to keep in mind, contrary to the HMM-version of the forward algorithm, that we might have to look back several $t$-cubes in the $\tilde{\alpha}$-table. In order to keep the invariant
\[
\tilde{\alpha}_t(z_t, k, q) = \left( \prod_{t'=1}^{t} c_{t'} \right) \tilde{\alpha}_t(z_t, k, q)
\]
satisfied, we cannot “skip” $c$-values. Intuitively, we solve this by scaling the entry values by the skipped $c_t$-values before saving them into the $\tilde{\alpha}$-array. Formally, we have
\[
\tilde{\alpha}_t(z_t, k, q) = \begin{cases}
\sum_{z_t,k,q} a_{z_t-\phi_{z_t},z_t} \sum_{q' \in Q'(z_t,q)} \tilde{\alpha}_{t-\phi_{z_t}}(z_{t-\phi_{z_t}}, k, q') & \text{if } q \notin A \\
0 & \text{if } q \in A \land k = 0 \\
\sum_{z_t,k,q} a_{z_t-\phi_{z_t},z_t} \sum_{q' \in Q'(z_t,q)} \left( \prod_{t'=1}^{t} c_{t'} \right) \tilde{\alpha}_{t-\phi_{z_t}}(z_{t-\phi_{z_t}}, k, q') & \text{if } q \notin A \\
0 & \text{if } q \in A \land k = 0 \\
\sum_{z_t,k,q} a_{z_t-\phi_{z_t},z_t} \sum_{q' \in Q'(z_t,q)} \tilde{\alpha}_{t-\phi_{z_t}}(z_{t-\phi_{z_t}}, k, q') & \text{if } q \notin A \\
0 & \text{if } q \in A \land k = 0
\end{cases}
\]
(3.21)

In other words, in the case of $\phi_{z_t} > 1$, the entry values should not only be scaled by $c_t$, but with the product of $c_{t-\phi_{z_t}+1} \cdots c_{t-1}$ as well. Since these values are all known at the time, this change is implemented easily.

A cut off version of the restricted forward algorithm for MEHMMs can be implemented similarly to the HMM algorithm, using the $c$-values calculated by the basic forward algorithm. This requires an implementation of the basic
forward algorithm for MEHMMs, which, again, is just a simplified version of the restricted forward algorithm for MEHMMs.

Based on the considerations mentioned above, we have implemented the MEHMM restricted forward algorithm both with and without the cut off method. Pseudocode of the implementation of the version using the cut off method is shown in algorithm 9.

Algorithm 9 MEHMM K-CUTOFF-FORWARD (1/2)

Require: $N$, $\Pi$, $A$, $B$, $\phi$, regexp, $X$, $f_{\text{cutoff}}$

Ensure: Return the $k$ probability distribution cut off at fraction $f_{\text{cutoff}}$

1: $dfa \leftarrow \text{CONSTRUCT-DFA}(N, \text{regexp})$
2: $Q \leftarrow \text{states}[dfa]$
3: $c[] \leftarrow \text{FORWARD-CVALUES}(N, \Pi, A, B, X)$
4: $\text{totalCProd} \leftarrow 1$ \Comment{Calculate occurrence probability}
5: for $t \leftarrow 0$ to $T - 1$ do
6: \quad $\text{totalCProd} \leftarrow \text{totalCProd} \cdot c[t]$
7: $f_{\text{current}} \leftarrow 0$
8: $k \leftarrow 0$
9: $\hat{\alpha}_k \leftarrow$ new zero-initialized table
10: while $f_{\text{current}} < f_{\text{cutoff}}$ do
11: \quad $\hat{\alpha}_k - 1 \leftarrow \hat{\alpha}_k$
12: \quad $\hat{\alpha}_k \leftarrow$ new zero-initialized table
13: for $t \leftarrow 1$ to $T - 1$ do
14: \quad for $z_t \leftarrow 0$ to $N - 1$ do
15: \quad \quad if $t = \phi[z_t] - 1$ then
16: \quad \quad \quad \quad \quad $\text{val} \leftarrow \Pi[z] \cdot B[z] \cdot X[0] \cdots X[t] / c[0]$
17: \quad \quad \quad \quad \quad for $q \leftarrow 0$ to $|Q| - 1$ do
18: \quad \quad \quad \quad \quad \quad if TRANSITION-EXISTS($\text{initial}[dfa], z, q$) then
19: \quad \quad \quad \quad \quad \quad \quad if $k = 1$ & accept[q] then
20: \quad \quad \quad \quad \quad \quad \quad \quad $\hat{\alpha}_k[t][z][q] \leftarrow \text{val}$
21: \quad \quad \quad \quad \quad \quad \quad else if $k = 0$ & !accept[q] then
22: \quad \quad \quad \quad \quad \quad \quad \quad $\hat{\alpha}_k[t][z][q] \leftarrow \text{val}$

The difference from the pseudocode of the HMM version presented in algorithm 8 are related to the changes described previously in this chapter. For MEHMMs, we have to know both $t$ and $\phi[z_t]$ to determine whether an entry is a base case or recursion case. Therefore, the base case has been included in the $t$-loop (line 13), where it is distinguished from the recursion case by comparing $t$ and $\phi[z_t]$ (line 15 and 23). Note that the base case here is characterized by $t = \phi[z_t] - 1$, since the tables are zero indexed. Furthermore, we can leave out the case of $t < \phi[z_t] - 1$ since our tables are initialized with zeroes.
Algorithm 9 MEHMM K-CUTOFF-FORWARD (2/2)

23: \[ \text{else if } t > \phi[z_t] - 1 \text{ then} \]
24: \[ c_{\text{Prod}} \leftarrow 1 \]
25: \[ \text{for } i \leftarrow 0 \text{ to } \phi[z_t] - 1 \text{ do} \]
26: \[ c_{\text{Prod}} \leftarrow c_{\text{Prod}} \cdot c[t - i] \]
27: \[ \text{for } q \leftarrow 0 \text{ to } |Q| - 1 \text{ do} \]
28: \[ \text{if } \text{accept}[q] \text{ & } k = 0 \text{ then} \]
29: \[ \text{continue} \]
30: \[ \text{outSum} \leftarrow 0 \]
31: \[ Q' \leftarrow \text{FIND-PREDECESSORS}(z, q) \]
32: \[ \text{for } z_{t-\phi[z_t]} \leftarrow 0 \text{ to } N - 1 \text{ do} \]
33: \[ \text{inSum} \leftarrow 0 \]
34: \[ \text{for all } q' \text{ in } xQ' \text{ do} \]
35: \[ \text{if } \text{accept}[q] \text{ then} \]
36: \[ \text{inSum} \leftarrow \text{inSum} + \hat{\alpha}_k[t - \phi[z_t]][z_{t-\phi[z_t]}][q'] \]
37: \[ \text{else} \]
38: \[ \text{inSum} \leftarrow \text{inSum} + \hat{\alpha}_k[t - \phi[z_t]][z_{t-\phi[z_t]}][q'] \]
39: \[ \text{outSum} \leftarrow \text{outSum} + \text{inSum} \cdot A[z_{t-\phi[z_t]}][z_t] \]
40: \[ \hat{\alpha}_k[t][z_t][q] \leftarrow \text{outSum} \cdot B[z_t][X[t-\phi[z_t]] + 1 \cdots X[t]] / c_{\text{Prod}} \]
41: \[ \text{scaledLastSum} \leftarrow 0 \]
42: \[ \text{for } z \leftarrow 0 \text{ to } N - 1 \text{ do} \]
43: \[ \text{for } q \leftarrow 0 \text{ to } |Q| - 1 \text{ do} \]
44: \[ \text{scaledLastSum} \leftarrow \text{scaledLastSum} + \hat{\alpha}_k[T - 1][z][q] \]
45: \[ f_{\text{current}} \leftarrow f_{\text{current}} + \text{scaledLastSum} \]
46: \[ \text{occurrenceProbabilities}[k] \leftarrow \text{totalCProd} \cdot \text{scaledLastSum} \]
47: \[ k \leftarrow k + 1 \]
48: \[ \text{return occurrenceProbabilities} \]

The rest of the pseudocode should be familiar, with the exception of looking back several \( t \)-values in the two 3-dimensional \( \hat{\alpha} \)-tables.

Note that in the standard implementation, i.e. without using the cut off approach, another part of the \( \hat{\alpha} \)-table is saved, namely the last \( \phi[z], t \)-cubes. Consequently, to avoid unnecessary overhead in that case, we use modulo arithmetic to reference specific \( t \)-cubes, as mentioned in section 3.4.1. This is done in a similar way as in the MEHMM restricted Viterbi algorithm, for which we will present pseudocode in section 3.4.3.

The restricted forward algorithm as presented terminates once \( f_{\text{current}} \) exceeds \( f_{\text{cutoff}} \). At this point, the cut off \( k \)-distribution is complete and is returned.

### 3.4.3 The restricted Viterbi algorithm

The restricted Viterbi algorithm for MEHMMs is of course very much like the HMM version of it. This section will present the algorithm mainly by focusing on the differences between the MEHMM and the HMM version. Also, the
changes needed to apply the approximate linear space method for MEHMM restricted Viterbi will be presented.

Theory

The Viterbi parameter from equation 3.10 is changed to reflect our change in notation where an index into our hidden state sequence $Z$ reflects not the position in the index, but the hidden state corresponding to the observable symbol with the given index in $X$:

$$\tilde{\omega}_t(z_t, k, q) = \max\{Z_{1:t - \phi_{z_t}} : O_r(Z_{1:t}) = k\} \mathbf{P}(X_{1:t}, Z_{1:t}, \mathbf{FA}_H(r)_t = q)$$

As discussed in section 3.4.1, the base case is now $t = \phi_{z_t}$. For the same reasons as with the restricted Viterbi algorithm for HMMs, the base case is equal to the base case of the MEHMM restricted forward algorithm in equation 3.18:

$$\tilde{\alpha}_t(z_t, k, q)$$ for $t = \phi_{z_t}$

To obtain numerical stability, we work in log space and get, similar to equation 3.11 for $t = \phi_{z_t}$

$$\tilde{\omega}_t(z_t, k, q) = \log \tilde{\alpha}_t(z_t, k, q)$$

$$= \begin{cases} 
\log \pi_{z_t} + \log b_{z_t,x_1 \ldots x_t} & \text{if } q_0 \in Q'(z_t,q) \land \left((q \notin A \land k = 0) \lor (q \in A \land k = 1)\right) \\
-\infty & \text{otherwise}
\end{cases}$$

Now, the probability of being in some state $z_t$ while having emitted fewer symbols than the state emits, is of course not possible, so we get

$$\tilde{\omega}_t(z_t, k, q) = -\infty \text{ for } t < \phi_{z_t}$$

The recursion case from equation 3.12 is changed to reflect our change in notation as well as the fact that we emit a variable amount of observables from each state and maximize over different $t$-cubes according to the $\phi$ value of the hidden state. With those changes, for $t > \phi_{z_t}$ we arrive at

$$\tilde{\omega}_t(z_t, k, q) = \begin{cases} 
\log b_{z_t,x_s \ldots x_t} + \max_{z_{s-1}} \log a_{z_{s-1},z_t} + \max_{q' \in Q'(z_t,q)} \tilde{\omega}_{s-1}(z_{s-1}, k, q') & \text{if } q \notin A \\
\log b_{z_t,x_s \ldots x_t} + \max_{z_{s-1}} \log a_{z_{s-1},z_t} + \max_{q' \in Q'(z_t,q)} \tilde{\omega}_{s-1}(z_{s-1}, k-1, q') & \text{if } q \in A \land k > 0 \\
-\infty & \text{if } q \in A \land k = 0
\end{cases}$$

where $s = t - \phi_{z_t} + 1$.

Backtracking starts from the same entry as HMMs, namely $\arg \max_{(z_T, k \in [l,u],q)} \tilde{\omega}_T(z_T, k, q)$.

Backtracking follows the same scheme as the recursion case of the $\tilde{\omega}$ calculation.
in this section. So we examine the entries \( \hat{\omega}_{t-\phi z_t} \) as we are searching for the next entry in the trace. When \( t < \phi z_t \), backtracking is done. Backtracking an entry takes \( O(N \cdot |Q|) \) and in the worst case, the trace holds \( T \) entries so backtracking can be done in time \( O(T \cdot N \cdot |Q|) \) like for HMMs.

The dimensions of the \( \hat{\omega} \) table remain the same as for HMMs and even though the dimensions of the emission table increase, finding the emission probability is still a constant time table lookup. The total time complexity therefore remains the same \( O(N \cdot M^L) \). The space complexity of the MEHMM restricted Viterbi algorithm then becomes \( O(T \cdot N \cdot u \cdot |Q|^2) \) where \( M \) is the number of output symbols. For gene prediction, \( M \) will be the amount of nucleotides and \( L \) will be small, likely around the size of a codon. So in this context, the last part will be dominated and in practice we will get a space consumption of \( O(T \cdot N \cdot u \cdot |Q|) \) corresponding to the space consumption of the HMM algorithm.

**Approximate linear space** The space improvement for restricted Viterbi, described in section 3.2.2 is implemented in a similar way for MEHMMs as for HMMs. We need to update the backtracking equations to reflect that we can emit more than one symbol from each state and therefore skip columns as we backtrack according to the \( \phi \) value of the entry being backtracked from.

First, the \( \delta_t \) function from equation 3.13 must be updated so that we examine the correct table entries and transition probabilities as we search for the maximum

\[
\delta_t(z_t, k, q) = \arg \max_{z_{s-1}, q' \in Q'(z_t, q)} a_{z_{s-1}, z_t, \hat{\omega}_{s-1}}(z_s_{-1}, k, q')
\]

where \( s = t - \phi z_t + 1 \).

The base case in equation 3.14 remains the same, except the base case is no longer for \( t = 1 \), but for \( t = \phi z_t \)

\[
\psi_t(z_t, k, q) \rightarrow \text{prev} = \text{NULL} \\
\psi_t(z_t, k, q)[\text{val}] = z_t
\]

In the recursion case, where \( t > \phi z_t \), equation 3.15 must be updated so the backtracking pointers reflect that backtracking an entry might skip columns according to the value of \( \phi z_t \)

\[
\psi_t(z_t, k, q) \rightarrow \text{prev} = \begin{cases} 
\psi_{s-1} \left( \pi_1 \left( \delta_t(z_t, k, q) \right), k, \pi_2 \left( \delta_t(z_t, k, q) \right) \right) & \text{if } q \notin A \\
\psi_{s-1} \left( \pi_1 \left( \delta_t(z_t, k - 1, q) \right), k - 1, \pi_2 \left( \delta_t(z_t, k - 1, q) \right) \right) & \text{if } q \in A \land k > 0 \\
\text{NULL} & \text{if } q \in A \land k = 0
\end{cases}
\]

\[
\psi_t(z_t, k, q)[\text{val}] = z_t \text{ for } t = 2, \cdots, T
\]
where $s = t - \phi_z + 1$.

Note that we no longer know the length of the most likely hidden state sequence $Z^*$ in advance, since it is not necessarily equal to $T$. The length of the sequence is revealed as we backtrack the $\hat{\omega}$ table. Let $V$ be the length of the most likely hidden state sequence. If we denote the linked list nodes of $z^*_1, \cdots, z^*_V$ as $node_1, \cdots, node_V$ we find

$$node_V = \psi_T \left( \arg \max_{(z_T, k \in [l, u], q)} \hat{\omega}_T(z_T, k, q) \right)$$

The most likely hidden state path can, as with the normal restricted Viterbi, be found by following the pointers and reading off the hidden state values.

$$node_v = node_{v+1} \rightarrow \text{prev} \text{ for } v = V - 1, \cdots, 1$$

$$z^*_v = node_v[\text{val}] \text{ for } v = V, \cdots, 1$$

**Implementation**

As the implementations of different variations of the HMM restricted Viterbi algorithm have already been presented, we have chosen to only present the implementation of one flavour of the MEHMM restricted Viterbi algorithm, namely the approximate linear space method.
Algorithm 10 MEHMM-RESTR-VITERBI-SPACEOPT (1/2)

Require: $N, \Pi, \phi, A, B, \text{regexp}, X$

Ensure: $\hat{\omega}^{(T-1)\% (\max(\phi)+1)}$ holds the final $t$-cube of a complete table, $\psi$ holds linked lists for backtracking the corresponding $\hat{\omega}$ entries

1: $dfa \leftarrow \text{CONSTRUCT-DFA}(N, \text{regexp})$
2: $Q \leftarrow \text{states}[dfa]$
3: for $t \leftarrow 0$ to $T - 1$ do
4: for $z_t \leftarrow 0$ to $N - 1$ do
5: if $t < \phi_{z_t} - 1$ then
6: for $k \leftarrow 0$ to $u$ do
7: $\hat{\omega}_t[z_t][k][q] \leftarrow -\infty$
8: else if $t = \phi_{z_t} - 1$ then
9: $\hat{\omega}_t[z_t][k][q] \leftarrow -\infty$
10: end if
11: if TRANSITION-EXISTS(initial[dfa], $z_t, q$) then
12: for $q \leftarrow 0$ to $|Q| - 1$ do
13: if accept[$q$] then
14: $\hat{\omega}_t[z_t][1][q] \leftarrow \text{val}$
15: $\psi_t[z_t][1][q] \leftarrow \text{NEW-NODE}(z_t)$
16: else
17: $\hat{\omega}_t[z_t][0][q] \leftarrow \text{val}$
18: $\psi_t[z_t][0][q] \leftarrow \text{NEW-NODE}(z_t)$
19: end if
20: end for
21: end if
22: end for
23: end for

The pseudocode is presented in algorithm [10]. In the HMM version of the algorithm, we used two different three dimensional tables of $\hat{\omega}$ and $\psi$, corresponding to two four dimensional table where the $t$ dimension is of size two. For MEHMMs, when we calculate the $\hat{\omega}$ and $\psi$ tables for a given $t$ value, we need access to entries in the range $[t - \max(\phi), t - 1]$ where $\max(\phi) = \max z$.

Therefore, we need four dimensional tables where the size of the $t$ dimension is $\max(\phi) + 1$ similar to HMMs, where $\max(\phi) = 1$. We want to store the $\max(\phi) + 1$ most recent $\hat{\omega}$ and $\psi$ entries in the tables and similar to the forward algorithm, we use modulus to do indexing into the array, i.e. $t \mod (\max(\phi) + 1)$. Note that we use $\%$ to denote modulus in the pseudocode.

Because the switch from base to recursion case depend not only on $t$, but also on $\phi_{z_t}$ as depicted in figure [5,7] we have merged the base and recursion case into one $t$ loop (line 3), distinguishing them with conditional statements inside the loop (lines 5, 9 and 21). On lines 6-8, we fill the $\hat{\omega}$ entries where the state is incompatible with the amount of outputted observables. On lines 10-20, we handle the base case, which as we move to zero indexed arrays, becomes $t = \phi_{z_t} - 1$. The emission table lookup on line 10 is handled with a helper object incorporating Horner’s method for translating into an array with fixed dimensions. The details of this have been left out of the pseudocode to make it more clear. Otherwise the base case corresponds to the base case of the...
algorithm for HMMs. Note that since \( t < \phi_{z_t} \) in the base case, we do not need to use modulus for indexing.

**Algorithm 10 MEHMM-RESTR-VITERBI-SPACEOPT (2/2)**

21: \( \text{else} \) \hspace{1cm} \triangleright \text{Recursion case}
22: \( y \leftarrow \max \{ \phi \} + 1 \)
23: \( \text{for } k \leftarrow 0 \text{ to } u \text{ do} \)
24: \( \text{for } q \leftarrow 0 \text{ to } |Q| - 1 \text{ do} \)
25: \( \text{if } \text{accept}[q] \& \ k = 0 \text{ then} \)
26: \( \tilde{\omega}_t^{y}[z_t][k][q] \leftarrow -\infty \)
27: \( \text{continue} \)
28: \( \text{outerMax} \leftarrow -\infty \)
29: \( Q' \leftarrow \text{FIND-PREDECESSORS}(z_t, q) \)
30: \( \text{for } z_{\text{prev}} \leftarrow 0 \text{ to } N - 1 \text{ do} \)
31: \( \text{innerMax} \leftarrow -\infty \)
32: \( \text{for all } q' \text{ in } Q' \text{ do} \)
33: \( \text{if } \text{accept}[q] \text{ then} \)
34: \( \text{inner} \leftarrow \tilde{\omega}_{(t-\phi_{z_t})}[z_{\text{prev}}][k][q'] \)
35: \( \text{if } \text{inner} > \text{innerMax} \text{ then} \)
36: \( \text{innerMax} \leftarrow \text{inner} \)
37: \( \text{innerMax}_k \leftarrow k \)
38: \( \text{innerMax}_q \leftarrow q' \)
39: \( \text{else} \)
40: \( \text{inner} \leftarrow \tilde{\omega}_{(t-\phi_{z_t})}[z_{\text{prev}}][k][q'] \)
41: \( \text{if } \text{inner} > \text{innerMax} \text{ then} \)
42: \( \text{innerMax} \leftarrow \text{inner} \)
43: \( \text{innerMax}_k \leftarrow k \)
44: \( \text{innerMax}_q \leftarrow q' \)
45: \( \text{outer} \leftarrow \text{innerMax} + \log A[z_{\text{prev}}][z_t] \)
46: \( \text{if } \text{outer} > \max \text{ then} \)
47: \( \max \leftarrow \text{outer} \)
48: \( \max \leftarrow z_{\text{prev}} \)
49: \( \max_k \leftarrow \text{innerMax}_k \)
50: \( \max_q \leftarrow \text{innerMax}_q \)
51: \( \tilde{\omega}_t^{y}[z_t][k][q] \leftarrow \max + \log(B[z_t][X[t - \phi_{z_t} + 1]] \cdots [X[t]]) \)
52: \( \text{if } \tilde{\omega}_t^{y}[z_t][k][q] \neq -\infty \text{ then} \)
53: \( \psi_t^{y}[z_t][k][q] \leftarrow \text{NEW-NODE}(z_t) \)
54: \( (\psi_t^{y}[z_t][k][q] \rightarrow \text{prev}) \leftarrow \psi_{(t-\phi_{z_t})}[z_{\text{prev}}][\text{max}_k][\text{max}_q] \)

The recursion case, on lines 22-54 is very similar to the recursion case for the HMM algorithm presented in algorithm 6. On line 22, we let \( y = \max \phi + 1 \) and then, as explained, in all the code do modulo \( y \) on indexes into the \( t \) dimension of the \( \tilde{\omega} \) and \( \psi \) tables. Also, where we in the HMM version reference \( t - 1 \) entries of the tables, we need to reference \( t - \phi_{z_t} \) entries in the MEHMM version. Apart from the reference and indexing changes, lines 28-50 find the maximum for a
given entry like for HMMs. On line 51, we save the maximum in the current entry of the \( \hat{\omega} \) table. If the value is not \(-\infty\), this value might give rise to a later maximum and so we create a linked list node on line 53. The previous pointer of this node must point to the linked list node corresponding to the \( \hat{\omega} \) entry that gave rise to the maximum of this entry. This is done on line 54.

When the algorithm terminates, \( \hat{\omega}((T-1)%(\max[\phi]+1) \) holds what would have been the final \( t \)-cube of a complete table and so this is where the backtracking begins.

**Algorithm 11** MEHMM-RESTRI-VITERBI-SPACEOPT-BACKTRACK

**Require:** \( N, \phi \) and that \( \hat{\omega}((T-1)%(\max[\phi]+1) \) holds the final \( t \)-cube of a complete table, \( \psi \) holds linked lists for backtracking the corresponding \( \hat{\omega} \) entries

**Ensure:** The array \( \text{trace}_z \) holds a most likely sequence of hidden states

```latex
1: y \leftarrow \max[\phi] + 1
2: max \leftarrow -\infty
3: \textbf{for} z \leftarrow 0 \textbf{to} N - 1 \textbf{do} \quad \text{\hspace{1cm}} \text{\textgreater{} Base case}
4: \textbf{for} k \leftarrow l \textbf{to} u \textbf{do}
5: \quad \textbf{for} q \leftarrow 0 \textbf{to} |Q| - 1 \textbf{do}
6: \quad \quad val \leftarrow \hat{\omega}((T-1)%(\max[\phi]+1))[z][k][q]
7: \quad \quad \textbf{if} val > max \textbf{then}
8: \quad \quad \quad max \leftarrow val
9: \quad \quad \quad max_z \leftarrow z
10: \quad \quad \quad max_k \leftarrow k
11: \quad \quad \quad max_q \leftarrow q
12: \quad \textbf{end for}
13: \textbf{end for}
14: \textbf{end for}
15: \textbf{while} node \neq NULL \textbf{do} \quad \text{\hspace{1cm}} \text{\textgreater{} Recursion case}
16: \quad index \leftarrow 0
17: \quad trace_z[index] \leftarrow max_z
18: \quad node \leftarrow (\psi((T-1)%(\max[\phi]+1))[max_z][max_k][max_t]) \rightarrow \text{prev}
19: \quad \textbf{while} node \neq NULL \textbf{do}
20: \quad \quad index \leftarrow index + 1
21: \quad \quad trace_z[index] \leftarrow val[\text{node}]
22: \quad \quad node \leftarrow (\text{node} \rightarrow \text{prev})
23: \quad \textbf{end for}
24: \textbf{end while}
25: \textbf{for} \text{\hspace{1cm}} \text{\textgreater{} Reversing the trace}
26: \quad \textbf{while} \text{\hspace{1cm}} \text{\textgreater{} Reversing the trace}
27: \quad \quad index \leftarrow index - 1
28: \quad \quad trace_z[index] \leftarrow \text{\text{reverse}[trace_z]}
29: \textbf{end for}
30: \textbf{end while}
31: \textbf{reverse}[\text{trace}_z]
```

The base case of the backtracking is finding the maximum entry of \( \hat{\omega}((T-1)%(\max[\phi]+1) \), which is done on lines 2-11. On line 14, we store a reference to the corresponding linked list node of \( \psi \). On lines 15-18, we follow the \( \text{prev} \) pointers to find the trace, which is stored in reverse order in \( \text{trace}_z[] \). As we do not know the length in advance, we allocate a size of \( \frac{T}{\min[\phi]} \) for the trace array, since this would be the maximum size of the trace. On line 19, we reverse the array to produce the final trace.

**Summary**

We have introduced MEHMMs, which allow a more fine grained model, and where hidden states can output more than one symbol. This enables us to reduce the number of hidden states in our models. We have adapted the re-
restricted algorithms to this formalism, and we have presented implementation
details regarding our improved restricted algorithms.
Chapter 4

Experiments

The results of [20] pose an interesting alternative to the basic Viterbi algorithm. Depending on the context, we now have the opportunity of imposing relevant restrictions on the calculated hidden state sequence. The trade-off, as mentioned earlier, consists in a more advanced algorithm with greater time and space consumption. The algorithms and their applied combination of finite automata and hidden Markov models are theoretically intriguing regardless, but the associated time and space complexity may render it practically inapplicable.

In chapter [3] we have developed ideas to improve the practical resource requirements of the restricted algorithms. While the improvements do not change the theoretical time or space complexity, we still expect significant practical improvements on the time and space consumption.

In this chapter, our goal is to verify the postulated abilities of the restricted algorithms and test whether our alterations to the naïve implementation lead to significant changes in time and/or space consumption.

The implementation is written in Java, and it can be downloaded from http://cs.au.dk/~sandholm/thesis/. All experiments have been run on a computer with an Intel Core2 Duo 2.40GHz processor with 4 GB memory running 64 bit Windows 7 and Oracle's Java Virtual Machine version 1.6.

4.1 Accuracy of the restricted Viterbi algorithm

We find it appropriate to reproduce the results of [20] from an independent implementation of the restricted algorithms. If we obtain the same results as [20], we gain confidence in both the results of [20] and our implementation. By performing the same experiments as [20], we naturally expect very similar overall results, but with minor deviations due to the random element of using sequences generated by the hidden Markov model.

We examine the normalized differences between the true number of pattern occurrences and the number given by the basic Viterbi and the restricted forward algorithm, as well as the sensitivity, specificity and MCC values given by the basic and restricted Viterbi algorithm.
The restricted forward algorithm Similarly to [20], we use the HMM presented in figure 2.7 to generate a test set of 100 pairs of \((X, Z)\)-pairs for each of the \(T\)-values 500, 525, \cdots, 1500. Furthermore, we choose the same regular expression, \(r = (N N(C_1|R_3))|(C_5|R_1)N N\), corresponding to the transitions between coding and non-coding regions.

We run the restricted forward algorithm on all the generated observable sequences, each yielding a probability distribution over the number of pattern occurrences in the hidden state sequence. From each such distribution, we then calculate the upper and lower bound of its 95\% interval as well as the expectation. Grouping sequences of equal length together provides normally distributed sets of expected values and upper and lower 95\% interval bounds. For each \(T\), we thus have a normal distribution for each of these three values, from which an expected value and standard deviation can be determined. From these values we then calculate normalized differences, \(\frac{\text{estimate} - \text{true value}}{\text{true value}}\), from the true value of the occurrence count in the hidden sequences.

The normalized differences are plotted in figure 4.1 and the intervals corresponding to the standard deviation on both sides of expectation are coloured.

![Figure 4.1: Normalized differences between true number of pattern occurrences and the number of occurrences induced by Viterbi and the restricted forward algorithm](image)

Furthermore, we run the basic Viterbi algorithm on the same generated observable sequences and count the number of pattern occurrences in the returned hidden sequence, yielding a normal distribution of the expected number of pattern occurrences for each \(T\). After calculating the standard deviation and normalizing the difference to the true value, we insert the Viterbi results in figure 4.1 as well.

As expected, we see convincing similarity between figure 4.1 and the corre-
sponding figure 3 in [20]. Small deviations are visible due to the randomness of the generated sequences, but no significant discrepancies appear. We observe that the results of the restricted forward algorithm improve in relative precision as $t$ increases, which is in agreement with figures A.1 and A.2.

Our conclusion to this part of the experiment is therefore the same as the one in [20], that is, the restricted forward algorithm estimates the true number of pattern occurrences with high precision. The estimate of the Viterbi algorithm, on the other hand, is consistently too low, at around only $\frac{1}{5}$ of the true value. As mentioned in [20], the low estimates of the Viterbi algorithm are probably connected to the low model probabilities to and from coding regions, which leads to the prediction of few, but long genes. This is because the basic Viterbi will prefer not to make unlikely transitions, unless the emission probabilities "outweigh" the low transition probability. In contrast, the restricted Viterbi uses the probability distribution over the number of genes to make sure the number of predicted genes is comparable to the amount induced by the transition probabilities of the model.

However, the model generated annotations might result in the basic Viterbi algorithm performing worse than it would under real circumstances. After all, the generated sequences might contain very short genes by chance, while the typical gene length of bacteria is around 1,000bp.

Now, even though the occurrence count expected by the Viterbi algorithm seems very low, it might be the case that it is on equal footing with the restricted algorithms at predicting the coding status (coding/non-coding) of each DNA residue. To test this, a look at the restricted Viterbi results is necessary.

**The restricted Viterbi algorithm** Analogously to [20], we use average sensitivity, specificity and Matthew’s correlation coefficient (MCC) [1] for comparing the accuracy regarding coding and non-coding regions of the restricted Viterbi compared to the basic Viterbi. Sensitivity and specificity measure the algorithms ability to identify coding and non-coding regions, respectively, while MCC is an overall accuracy measurement that captures both these aspects. Sensitivity and specificity can vary between 0 and 1, while MCC can vary between -1 and 1, where 1 represents perfect prediction.

In order to perform these measurements on the restricted Viterbi, we first have to run it on the observable sequences from known $(X, Z)$-pairs with some chosen values for lower and upper occurrence bounds. Similarly to [20], and in order to make use of the information given by the restricted forward algorithm, we use the already generated sequence-pairs and choose two different approaches: one using the expectation value as both lower and upper occurrence bound, and one using the 95% interval bounds as occurrence bounds. From each of the resulting hidden state sequences, we calculate the number of true positives, false positives, true negatives and false negatives. The prediction of a coding nucleotide must be in the correct direction to be considered a true positive. Thus, we split the calculation of each of these values in two. For instance, for the number of true positives, $TP$, a value $TP_C$ is calculated considering $C_i$ as positive and $R_i$ and $N$ as negative, and $TP_R$ is calculated considering $R_i$ and $N$ as positive and $C_i$ as negative.
considering \( R_i \) as positive and \( C_i \) and \( N \) as negative. To give equal significance to genes in either direction, \( TP \) is then calculated as the average of \( TP_C \) and \( TP_R \).

After averaging the sensitivity, specificity and MCC values for each sequence length, the results are plotted in figure 4.2. Comparing the sensitivity figure to the corresponding figure in [20], we again see great similarity, as expected. The sensitivity of the three result types are all similar and relatively high, fluctuating at around 0.7. The restricted Viterbi algorithm on the 95% interval achieves the highest sensitivity.

The specificity values are very similar to the reference values as well. The two restricted Viterbi result sets are close to indistinguishable, while the basic Viterbi produces somewhat lower values. The lower specificity means that the basic Viterbi predicts coding regions better than non-coding regions. Considering the fact that the basic Viterbi predicts a lot fewer genes than the restricted Viterbi (cf. figure 4.1), this suggests that the few genes that Viterbi predicts generally are much too long, in agreement with the discussion of figure 4.1. Because of the model structure and probabilities, the hidden state sequences generated by the model contain on average three times as many \( C \)s and \( R \)s as it contains \( N \)s. So, for some prediction of a \( C \) or an \( R \) there is much higher probability of getting a true positive by chance than for some prediction of \( N \). This also contributes to the difference between the sensitivity and specificity of
the basic Viterbi.

For the MCC figure, we once again get similar results to the corresponding figure in [20]. The two restricted Viterbi graphs are closely positioned, while the basic Viterbi performs worse. The restricted Viterbi using the 95% occurrence interval seems to be performing a bit better than the one using the expected value. We assume that this small difference is connected to the fact that the run using the expectation only accepts hidden sequences with an occurrence count of exactly the expectation, which puts a serious limit on the range of possible hidden state sequences. Conversely, the run using the 95% interval has a much greater selection of hidden state sequences to choose from.

However, since the algorithms are using the same model, we expect the restricted Viterbi to prefer few, but long genes, similarly to the basic Viterbi. Its advantage is only the possibility of enforcing restrictions on the number of occurrences. Based on the behaviour of the basic Viterbi in figure 4.1, we therefore expect the restricted Viterbi to generally return hidden sequences with a number of occurrences in the lower end of the supplied occurrence interval. In other words, a too large interval would result in the restricted Viterbi inheriting the weaknesses of the basic Viterbi. In fact, if we use an interval holding all possible occurrence values, the restricted Viterbi reaches the same result as the basic Viterbi algorithm. The effect can be seen in the specificity figure, where the 95% restricted Viterbi only performs on equal footing with the expectation restricted Viterbi. Consequently, we expect the existence of a sweet spot between the two restricted Viterbi approaches, i.e., using an interval below 95%, performing better than both of them with regard to the MCC value.

Based on the similarity of our results and the results of Tataru et al. [20], we conclude that our independent implementation performs in accordance with the reference implementation, which gives us increased confidence in both the results of [20] and our implementation.

4.2 Time complexity of the restricted algorithms

In these experiments, we want to verify the theoretical time complexity of the restricted forward and restricted Viterbi algorithm. We choose not to verify the theoretical space complexity, since it is directly related to the size of the allocated table.

4.2.1 The restricted forward algorithm

Our hypothesis is that the time complexity of the restricted forward algorithm is $O(T \cdot N^2 \cdot m \cdot |Q|^2)$, where $T$ is the length of the observable sequence, $N$ is the number of hidden states in the hidden Markov model, $m$ is some chosen upper bound on the number of regular expression occurrences, i.e. $m \geq k_{\text{max}}$, and $|Q|$ is the number of states in the finite automaton. To verify our hypothesis, we measure the time consumption as a function of each term individually, e.g. we keep $N$, $m$ and $|Q|$ fixed and measure the time consumption as a function of $T$. In principle, we should perform such an experiment with many different combinations of fixed values of $N$, $m$ and $|Q|$, but this quickly leads to
a combinatorial explosion of parameter choices. Therefore we have made the pragmatic choice to only perform such an experiment once, i.e. only one choice of the three fixed parameters.

We thus want to carry out four experiments, each measuring the time consumption as a function of one of the parameters while keeping the other three fixed. This leads to considerations about parameter dependencies. The parameter $|Q|$ depends on the choice of regular expression, the parameter $N$ depends on the hidden Markov model, the parameter $T$ is the length of the observable sequence and the parameter $m$ depends on both the observable sequence length as well as the choice of regular expression. In other words, if we change the regular expression to vary the parameter $|Q|$, we also vary the parameter $m$, and if we change the observable sequence length $T$, we also vary the parameter $m$. The same applies in the other direction as if we change either the regular expression or the sequence length to vary the parameter $m$, we also vary the parameter $|Q|$ or $T$ respectively. However, to perform the four experiments, we need a way around these dependencies.

The value of $m$ decides the size of the table’s $k$-dimension, representing the number of regular expression occurrences. The table dimension needs to have size $m \geq k_{\text{max}}$ in order for the restricted forward algorithm to calculate the correct probability distribution. If, on the other hand, we are not interested in the results of the restricted forward algorithm, we can set the size of the table dimension $m$ independently of $k_{\text{max}}$. In other words, fixing the value of $m$ does not fundamentally change how the algorithm runs, so measuring the time consumption will be viable even though the algorithm no longer computes the correct probability distribution. Thus, we are able to vary only $|Q|$ or $T$ in order to verify their contribution to the time complexity. For varying $m$, we can choose to just do it directly by inputting $m$ to the algorithm and skip the normal calculation of $m$.

In this way, we are able to vary each one of the parameters solely and can proceed with the described experiments.

**Time consumption as a function of $T$** We use the HMM from figure 2.7 to generate an observable sequence for every length $T = 200, 400, \cdots, 5000$. One sequence for each length is sufficient, since as mentioned, we are not interested in the results of the algorithm, but merely its running time which is independent of the particular sequence choice. For the algorithm runs, we use the same HMM and the regular expression for gene boundaries $r = (NN(C_1|R_3))(((C_3|R_1)NN)$, as exemplified in [20], and fix the value of $m$ to 200. To even out noise from the host machine, we run the restricted forward algorithm five times on every observable sequence and then calculate the average time spent for each sequence.

We expect the time consumption to be linear in $T$ as the time complexity suggests.
Figure 4.3: The relationship between $T$ and the time consumption of the restricted forward algorithm.

We have plotted the results in figure 4.3. On the horizontal axis is the sequence length $T$ while the vertical axis holds the time consumption in milliseconds divided by $T$. If the time consumption is linear in $T$, we expect such a plot to yield points resembling a horizontal line. This is indeed what the graph shows. We have used the least squares method to compute a linear regression fit of the points, and as the figure shows, the slope of the linear regression fit is very close to zero. For the choices of parameters and hardware we have made, the restricted forward algorithm takes approximately 10.5ms to compute all $\alpha$ table entries for one value of $t$. We judge by the results of this experiment that the time consumption of the restricted forward algorithm is linear in $T$.

**Time consumption as a function of $m$**  In this experiment we use the same HMM as in the last experiment. We first generate an observable sequences of length $T = 1000$ and then run the restricted forward algorithm on the observable sequence for every $m = 10, 20, \cdots, 500$, using $r = (\text{NN}(C_1|R_3))|((C_3|R_1)\text{NN})$. To even out noise from the host machine, we measure the running time of 5 different runs per value of $m$, and we calculate the average time spent for every $m$.

We expect the time consumption to be linear in $m$ as the time complexity suggests.
The results have been plotted in figure 4.4. On the horizontal axis is the value of $m$ while the vertical axis holds the time consumption in milliseconds divided by $m$. Again, if the time consumption is linear in $m$, we expect such a plot to yield points resembling a horizontal line. The graph shows some noise in the beginning, but as $m$ grows larger, the points start to resemble a horizontal line. This is fine, since the time complexity only implies that for sufficiently large $m$, the time consumption is bounded by $cm$ where $c$ is some constant. One possible explanation of the notably larger time consumption for small values of $m$ is that the relative time to do initialization work will be longer when $m$ is small. We have used the least squares method to compute a linear regression fit of the points where $m \geq 190$, and if we only include those points, the slope of the linear regression fit is again very close to zero. For the choices of parameters and hardware we have made, the restricted forward algorithm takes approximately 52ms to compute all $\alpha$ table entries for one value of $k$. We conclude that the time consumption is linear in $m$.

**Time consumption as a function of $N$** In this experiment we want to vary $N$ while keeping $T$, $m$ and $|Q|$ constant. To do this, we have implemented code to auto generate simple HMMs. They are not useful for gene prediction, but they have some structure in common with our simple model. We generate HMMs for different values of $N$ by incrementally adding three states representing probabilities for some coding regions. The structure of the generated HMMs can be seen in figure 4.5. Emission probabilities for forward and reverse coding states are randomly chosen such that they sum to 1 for each state.
Figure 4.5: Generated HMM with $N = 10$ and $N = 13$ (including dashed objects). Non-labeled arcs from state $N$ sum to 0.1 and are equally distributed.

We generate a random sequence of length $T = 1000$. We then generate HMMs of sizes $N = 7, 10, \cdots, 100$ and perform five runs for each HMM, letting $m = 50$ and using the simple regular expression $r = NC_1$ so $|Q|$ remains constant. Finally, we calculate the average time spent on running the restricted forward algorithm on the differently sized HMMs.

We expect the time consumption to be quadratic in $N$ as the time complexity suggests.

![Graph showing the relationship between $N$ and the time consumption of the restricted forward algorithm.](image)

Figure 4.6: The relationship between $N$ and the time consumption of the restricted forward algorithm.

The result of the experiment can be found in figure 4.6. On the horizontal axis is $N$ while the time consumption in milliseconds divided by $N^2$ is on the vertical axis. As $N$ grows, the points resemble a horizontal line. Again, initialization overhead might explain why the time consumption is relatively larger for small $N$. The linear regression fit for $N \geq 57$ is shown in the plot.

$$f(N) = a \times N + b$$

$a = -0.00149327$

$b = 6.17353$

linear regression fit

![Graph showing linear regression fit](image)
and again the slope is very close to zero. We conclude that the time consumption is quadratic in $N$.

**Time consumption as a function of $|Q|$**  In this experiment we return to using the HMM from figure 2.7 from which we generate an observable sequence of length $T = 1000$. We set $m$ to 50, run the restricted forward algorithm five times for every $|Q| = 3, 5, \cdots, 59$ and calculate the average time spent at each $|Q|$. We vary $|Q|$ by choosing and generating regular expressions that give us the desired amount of states when the automaton is constructed.

We expect the time consumption to be quadratic in $|Q|$ as the time complexity suggests.

![Figure 4.7: The relationship between $|Q|$ and the time consumption of the restricted forward algorithm.](image)

The result is shown in figure 4.7. The horizontal axis holds $|Q|$ while the time in milliseconds divided by $|Q|^2$ is on the vertical axis. With the same arguments as in the previous experiments, we find that the plot and linear regression support our hypothesis that the time consumption is quadratic in $|Q|$.

**Summary of results**  Our experiments support our hypothesis that the time complexity of the restricted forward algorithm is $O(T \cdot N^2 \cdot m \cdot |Q|^2)$.

### 4.2.2 The restricted Viterbi algorithm

Recall that the time complexity for the restricted Viterbi algorithm is $O(T \cdot N^2 \cdot u \cdot |Q|^2)$, which we would like to verify as well. Like for restricted forward, we split the experiment into four subexperiments, each varying one of the parameters while keeping the other three constant and measuring time consumption.
The same considerations about the algorithms correctness while keeping one parameter from a co-varying parameter pair constant apply. Because of the similarities between restricted forward and restricted Viterbi, all subexperiments yielded very similar results to the ones presented for the restricted forward algorithm. We will therefore not repeat the same discussions, but merely present the plots (along with linear regression fits) from this experiment in appendix B.

4.3 Improvement of the time consumption

As described in section 3.1.3, precomputing possible previous states for each combination of transition symbol and state in the DFA can be expected to improve the time consumption of the restricted algorithms. Furthermore, the observation that $|Q'|$, in the case of gene prediction models, is constant and not linear in $|Q|$, gives us an expectation that the precomputation of possible previous DFA states saves a factor $|Q|$ in the time consumption.

We implement both algorithms with this supposed speed-up, and in this experiment, we want to examine whether it indeed improves the time consumption as we expect. Similarly to the $|Q|$ experiment from section 4.2, we want to vary $|Q|$ by varying the regular expression, and keep other parameters fixed, ignoring incorrect results of the algorithm. Since the restricted forward and restricted Viterbi algorithms are alike regarding the computation of possible previous DFA states, we are content with just performing the experiment on one of them. We choose the restricted Viterbi algorithm.

Similarly to previous experiments, we use the HMM from figure 2.7 from which we generate an observable sequence of length $T = 1000$. We set $l$ to 0 and $u$ to 50, run the improved restricted Viterbi algorithm five times for every $|Q| = 3, 5, \ldots, 59$ and calculate the average time spent at each $|Q|$. We vary $|Q|$ by choosing and generating regular expressions that give us the desired amount of states when the automaton is constructed.

Based on our considerations, we expect the time consumption to be linear in $|Q|$.
The result is shown in figure 4.8. The horizontal axis holds $|Q|$ while the time in milliseconds divided by $|Q|$ is on the vertical axis. The points resemble a horizontal line, which indicates that the time consumption is linear in $|Q|$. As the figure shows, the slope of a linear regression fit for $|Q| > 18$ is fairly close to zero. The restricted forward algorithm shows similar results when precomputing predecessors. We judge by the results of this experiment that the time consumption of this implementation of the restricted algorithms is linear in $|Q|$ when using the regular expressions as previously discussed. All subsequent implementations of the algorithms use this speed-up.

4.4 Space improvement for the restricted Viterbi algorithm

While interesting enough, we do not want to examine whether we achieve a space consumption having approximate linearity in $T$. Instead, in line with the other improvements, we want to measure whether the optimization is worthwhile, i.e. is the trade off between time and space preferable in the context of gene prediction? Also, we want to ensure that we actually save space.

To do this, we compare the time and space consumption of the ("standard") restricted Viterbi algorithm against the approximate linear space ("ALS") restricted Viterbi algorithm introduced in section 3.2.2.

The space complexity of the restricted Viterbi algorithm is $O(T \cdot N \cdot u \cdot |Q|)$ and while several things such as the transition and emission tables contribute to the space consumption, it is the $\hat{\omega}$ table that is the significant contributor. For the purpose of examining the space consumption, we only measure the size.
of this table and its counterparts of the same size such as the data structures holding backtracking information.

While we might not need to save as many linked list nodes in our backtracking data structure as we have table entries in the \( \hat{\omega} \) table, a linked list node takes up more space than its \( \hat{\omega} \) entry counterpart. In the standard restricted Viterbi, each entry in the \( \hat{\omega} \) table of size \( T \cdot N \cdot (u + 1) \cdot |Q| \) holds an 8 byte double. In the ALS restricted Viterbi, we create a linked list node for every entry in the normal \( \hat{\omega} \) table. Each node holds a 4 byte pointer for backtracking, a 4 byte integer for the hidden state and 8 bytes for object housekeeping (in Java). Note that even though we use an integer to hold the hidden state, in practice, a byte will typically be enough. Using an integer, we get a total of 16 bytes for every linked list node, so a linked list node takes up double the space as an entry in the \( \hat{\omega} \) table. In the ALS restricted Viterbi, because we store separate backtracking information, the \( \hat{\omega} \) table is constant in the \( t \) dimension and so in practice only three dimensional, e.g. of size \( 2 \cdot N \cdot (u + 1) \cdot |Q| \) for HMMs. Therefore, we only take the backtracking data structure into account when measuring space for the ALS restricted Viterbi, cf. the previous discussion.

We use the HMM from figure 2.7 to generate five observable sequences for every length \( T = 200, 400, \cdots, 5000 \). We use regular expression for gene boundaries \( r = (NN(C_1|R_3))|(C_3|R_1)NN \) and set a reasonable upper bound of \( u = T/20 \) for every length. First we measure the time consumption by running the standard restricted Viterbi and the ALS restricted Viterbi on each sequence. For every length, we calculate the average time consumption for each algorithm. We plot the time consumption of the ALS restricted Viterbi relative to the normal restricted Viterbi, i.e. \( \frac{\text{avg. ALS time}}{\text{avg. standard time}} \) for every length.

To measure the space consumption, we cannot rely on the garbage collector for precise measurements. Therefore, we have instrumented the ALS restricted Viterbi algorithm with code that measures the reachable nodes of the backtracking data structure. Every time we increment \( t \), so the nodes referenced by the \( \psi \) table changes, we follow the linked list pointers to compute the set of nodes that are reachable at this point in time. Nodes that are reachable cannot be garbage collected, so by keeping track of the maximum size of this set, \( \text{max}[\text{reachable}] \), we know the maximum number of linked list nodes that must be stored in the heap at the same time during the execution of the algorithm. The reason we measure the maximum number of reachable nodes with five different observable sequences is to account for the fact that different observable sequences might give rise to a different number of reachable nodes during execution. We know the number of cells in the \( \hat{\omega} \) table in the normal restricted Viterbi algorithm, \( \text{size}[\hat{\omega}] = T \cdot N \cdot (u + 1) \cdot |Q| \). We want to plot the space consumption of the ALS restricted Viterbi relative to the space consumption of the normal restricted Viterbi. As discussed, a linked list node takes up double the space of a \( \hat{\omega} \) entry, so for every length we plot the relative space consumption as \( \frac{\text{avg. max}[\text{reachable}]}{\text{size}[\hat{\omega}]} \) for every length.

We expect the time consumption of the ALS restricted Viterbi to be comparable to the normal restricted Viterbi, possibly with a small overhead because of the housekeeping required to manage the linked lists in the backtracking data
structure. We expect the space consumption of the ALS restricted Viterbi to be significantly smaller than the normal restricted Viterbi, since backtracking paths are expected to converge to the most likely hidden state path as discussed in section 3.2.2.

![Relative ALS time and space consumption](image)

Figure 4.9: Time and space consumption of ALS restricted Viterbi relative to the standard restricted Viterbi.

The results can be seen in figure 4.9. As expected, we see a great reduction in space consumption for the ALS restricted Viterbi compared to the standard restricted Viterbi. Even for low $T$ values, we use less than 10% space in the ALS restricted Viterbi compared to the standard restricted Viterbi. As we increase $T$, the relative space consumption becomes even lower and seems to settle around 5% of the standard restricted Viterbi. We believe that the reason the relative space consumption improves for growing values of $T$, is because the expected convergence towards the most likely hidden state sequence gives us a relatively more effective improvement for bigger values of $T$. The time consumption of the ALS restricted Viterbi is comparable to the normal restricted Viterbi. The induced overhead from managing backtracking information seems not to make such a big impact on the time consumption. For some values of $T$, the ALS restricted Viterbi was even faster than the normal restricted Viterbi. The reason may be that even though we have overhead in managing backtracking information in the ALS restricted Viterbi, we also spend time doing regular backtracking in the standard restricted Viterbi.

As discussed earlier, the ALS method makes no guarantees to improve the space consumption. Still, we conclude that the approximate linear space method for the restricted Viterbi algorithm is indeed a worthwhile improvement as the space consumption is greatly reduced while the time consumption is not notably increased as our results show.
4.5 The cut off method for the restricted forward algorithm

In this section, we will examine the results of applying the cut off method of section 3.3.2 to the restricted forward algorithm.

Our interest in testing the cut off method is mainly concerned with whether or not it represents a worthwhile trade-off. As discussed in section 3.3.2, for an appropriately chosen cut off fraction $f_{\text{cutoff}}$, we expect a slight loss in the precision of the occurrence distribution, but a significant improvement in the practical time consumption. Thus, we would like to examine two different properties of the cut off restricted forward algorithm compared to its standard counterpart: time consumption and occurrence distribution precision. Space consumption might be interesting to compare as well, but because of the simple and effective space improvement of both algorithms, time consumption is the biggest issue here. Note that in the experiment, we are not particularly interested in the absolute values of the distribution and time consumption, but more so in relative difference between the two methods.

For the experiment, we use the HMM from figure 2.7 and the regular expression for gene boundaries $r = (NN(C_1|R_3))((C_3|R_1)NN)$. We use the HMM to generate observable sequences for every length $T = 200, 400, \ldots, 5000$. Again, to even out noise from the host machine, we generate five different sequences of each length. We run the restricted forward algorithm with and without the cut off technique on every observable sequence, and for every sequence length, we calculate the average time spent for both methods. The occurrence distribution for each algorithm run is saved as well. We then calculate the average of these three values for each sequence length. In order to compare these, we calculate the percentage-wise fraction $\frac{\text{cut off estimate}}{\text{standard estimate}}$.

Since the cut off method only cuts probabilities of large $k$-values off the distribution, we expect estimates based on the cut off distribution to generally be lower than the corresponding estimates based on the standard distribution. We therefore expect the calculated percentages to be lower than 100%. However, by using a cut off fraction of 99.9 % for all algorithm runs, we expect the percentages to be in close proximity to 100 %.
Distribution percentage of standard forward algorithm

Figure 4.10: 95% bounds and expected value of number of occurrences of the cut off restricted forward algorithm relative to those of the standard restricted forward algorithm.

We have plotted the results in figure [4.10]. On the horizontal axis is the sequence length $T$ while the vertical axis holds the cut off estimate percentages of the standard estimates, $\frac{\text{cut off estimate}}{\text{standard estimate}}$. The line at 100% can be thought of as the values of the standard restricted forward algorithm. As expected, the cut off values are all a small fraction lower than the standard values. The occurrence expectation lies close to, or actually a bit higher than, 99.9% as one might anticipate given the cut off fraction of 99.9%. The lower 95% bound is generally very close to the values of the restricted forward algorithm, while the upper bound is somewhat lower. This is not surprising either, since cutting the 0.1% highest $k$-values off the distribution naturally has greater impact on the upper bound of a 95% interval than on the lower bound, which is less affected by such a change.

Furthermore, it seems evident that all three percentages increase with the sequence length $T$. In other words, the longer the sequence, the smaller the precision disadvantage from using the cut off method becomes. From examining $k$-distributions for different sequence lengths, we believe this is due to the fact that the significant part of the distribution becomes relatively more narrow as we increase $T$. An example of this effect can be seen by comparing the two figures in appendix [A]. A more narrow distribution decrease relatively more rapidly for $k$-values greater than the expectation, and so more of the discarded part of the distribution is located close to the significant part of the distribution.

For most practical purposes, the loss of around 0.1% precision in the distribution can easily be tolerated. The big question, then, is whether the cut off method brings along the expected improvements of the time consumption. To test this, we have for each sequence length calculated the cut off algorithm’s
average time consumption as percentage of the average consumption of the standard algorithm.

Figure 4.11: Time consumption of the cut off restricted forward algorithm relative to the standard restricted forward algorithm.

The results are plotted in figure 4.11 and show clear improvements as expected. The cut off technique decreases the time consumption to between 5 and 10% of the standard algorithm. The decreased time consumption is closely connected to the decrease in the size of the $k$-dimension of the table, which is shown in figure 4.12. This is not surprising, since the time consumption is linearly dependant on the table size.
We observe that the percentages generally decrease for increasing sequence length, looking to settle at around 5%. We believe that the decreasing percentages are due to the narrowing of the significant part of the distribution (cf. appendix A) for increasing $T$. Intuitively, a more narrow distribution means that $k_{cutoff}$ corresponding to some chosen cut off fraction moves relatively closer to the expected occurrence count and farther from $k_{max}$.

Note that the result of the comparison depends on the implementation choices of the standard restricted forward algorithm. Our version uses the simple way of calculating $m$ as described in section 3.1.1. In other words, the percentages of our time consumption and $k$-distribution size plots would generally be higher, if we had implemented and compared to a version of the standard algorithm using the actual $k_{max}$ as an upper bound. Since $k_{max} = 0.4 \cdot T$ in our case (see section 3.3), the percentages in such a comparison would be a bit more than doubled, i.e. settling around 12 % in the figure.

We can thus safely state that the time consumption of the cut off restricted forward algorithm is significantly lower than the restricted forward algorithm. To illustrate the improvement further, we have added figure 4.13 showing the actual time consumption averaged at each sequence length for our implementations of the two methods. The magnitude of the improvements induced by the cut off method is evident, and using the method might be a big step towards practical feasibility.
To conclude, we judge the cut off method to represent a time/precision trade-off that is more than worthwhile for practical purposes.

4.6 MEHMM time and space complexity

As discussed in section 3.4, a major benefit of using the more fine-grained and general MEHMMs over the regular HMMs is fewer hidden states. A smaller number of hidden states leads to smaller tables of the forward and Viterbi parameter, which given the same time complexity and time spent per entry is tantamount to smaller time and space consumption.

In this experiment, we wish to verify the time complexity of the restricted MEHMM algorithms and compare their time consumption with the restricted HMM algorithms. Since adjusting restricted forward and Viterbi for MEHMMs involves substantial changes to the algorithms, cf. section 3.4, it seems appropriate to repeat all previous HMM experiments for MEHMMs. With regard to the proportions of this section, however, we have decided to leave out such lengthy testing. Instead, we are going to verify that the algorithms have the same time complexity as the HMM algorithms. Verifying equal time complexities of the HMM and MEHMM algorithms is an important step in the process of showing that MEHMMs can replace HMMs in our practical applications. We will perform similar experiments as in section 4.2 while utilizing the time consumption improvement tested in section 4.3.

To diversify, we here perform the experiments on the Viterbi algorithm instead of the forward algorithm. Similarly to section 4.2, we are going to vary one parameter while keeping the others constant and ignore the fact that the algorithm consequently may produce inaccurate results.
For comparing the time consumption to the HMM restricted Viterbi, we are going to perform each experiment on both the HMM and MEHMM version of the algorithm. The use of arrays of variable dimensions leads to some overhead when translating between index arrays and an internal index using Horner’s method as described in section 3.3. Furthermore, a bit of overhead is added through other implementation details, e.g. the modulo arithmetic when dealing with the $\max_\phi t$-cubes in memory at a given time. We therefore expect some added overhead in the MEHMM algorithm, but we expect this to be fairly insignificant in comparison to the rest of the algorithms.

**Time consumption as a function of $T$** For both the HMM and MEHMM algorithms, we use a model with 7 states. Before running the algorithms, we generate one observable sequence for each length $T = 200, 400, \cdots, 5000$. The choice of model used to generate such sequences is not important. For the HMM algorithm, we use the regular expression for gene boundaries $r = (NN(C_1|R_3))((C_3|R_1)NN)$, and for the MEHMM algorithm, we use a similar regular expression $r = (NN(C_1|R_1))((C_1|R_1)NN)$, where $C_1$ is a specific forward coding state and $R_1$ is a specific reverse coding state, each having a $\phi$ value of 3. We fix the values of $l$ and $u$ to 0 and 200, respectively, and run the restricted Viterbi algorithm for the HMM and the MEHMM on every generated observable sequence, and then calculate the average time spent for each sequence.

We have earlier seen that the time consumption of the HMM restricted Viterbi is linear in $T$. We expect the time consumption of the MEHMM to be linear in $T$ as well. Because of the added overhead as described above, we expect the MEHMM Viterbi to be slower than the HMM Viterbi for equally sized $\tilde{\omega}$-tables.
We have plotted the results in figure 4.3. As expected, the graph shows both the HMM and the MEHMM algorithm to be linear in $T$. We have computed a linear regression fit of the MEHMM points, and as the figure shows, the slope of the linear regression fit is very close to zero. The MEHMM algorithm takes approximately 6.2 ms to compute all $\omega$ table entries for one value of $t$, while the HMM algorithm uses around 10% less for the same task. This slightly higher constant associated with the MEHMM time consumption is consistent with our expectations.

Note that the constant of the HMM restricted Viterbi time consumption is not equal to the one presented in appendix B, because we have since then improved its time consumption cf. section 4.3.

We should be careful about concluding on whether it is worth using MEHMMs based on the figure. In practice, the MEHMM will have much fewer states than the HMM. Thus, the MEHMM algorithm’s smaller table size might make up for the difference in constants.

We judge by the results of this experiment that the time consumption of the restricted Viterbi algorithm for MEHMMs is linear in $T$.

**Time consumption as a function of $u$**  For examining the time consumption as a function of $u$, we first generate an observable sequence of length $T = 1000$. For running the algorithms, we use the same models and regular expressions as in the last experiment. We run both restricted Viterbi algorithms five times for every $u = 10, 20, \cdots, 500$ and calculate both algorithms’ average time spent for each $u$-value. $l$ is 0 for all runs.

We expect the time consumption of both the HMM and MEHMM algorithm to be linear in $u$ as the time complexity suggests. Similarly to the the $T$
experiment, we also expect the constant of the MEHMM algorithm to be a bit higher than that of the HMM algorithm.

\[
\begin{align*}
\text{MEHMM} & \quad + \quad \text{HMM} \quad \times \\
\end{align*}
\]

\[
f(u) = a \cdot u + b \\
a = -0.000936412 \\
b = 31.3474
\]

Figure 4.15: The relationship between \( u \) and the time consumption of the MEHMM and HMM restricted Viterbi algorithms.

The results have been plotted in figure 4.15. On the horizontal axis is the value of \( u \) while the vertical axis holds the time consumption in milliseconds divided by \( u \). As expected, both series of points do indeed resemble a horizontal line. The slope of the linear regression fit of the MEHMM sample points is again very close to zero. For the choices of parameters and hardware we have made, the MEHMM restricted Viterbi algorithm takes approximately 31.3 ms to compute all \( \alpha \)-table entries for one value of \( k \). Comparing to the \( T \)-plot, the \( u \)-plot shows a similar relationship between the MEHMM constant and the HMM constant.

Based on the figure, we conclude that the time consumption of the restricted Viterbi algorithm for MEHMMs is linear in \( u \).

**Time consumption as a function of \( N \)** Similarly to the \( N \) experiment in section 4.2, we want to vary \( N \) while keeping \( T, m \) and \( |Q| \) constant. Again, we must be able to generate both HMMs and MEHMMs of a desired size. For the HMMs, we use the same technique as depicted in figure 4.5, while we generate the MEHMMs in a similar way, as shown in figure 4.16.
Like the generated HMMs, emission probabilities for forward and reverse coding states are randomly generated such that they sum to 1 for each state, and we generate MEHMMs for different values of $N$ by incrementally adding a single state. As mentioned earlier, these models are not generated with the purpose of using them for gene prediction, but merely to test how the model size affects the running time of the algorithms.

We generate a random sequence of length $T = 1000$. We then generate an HMM and an MEHMM of each of the sizes $N = 7, 10, \cdots, 100$ and perform five runs for each of the models, letting $l = 0$, $u = 50$ and using the simple regular expression $r = NC_1$ so $|Q|$ remains constant. Finally, we calculate the average time spent on running the restricted forward algorithm on the differently sized models, grouped into HMMs and MEHMMs.
The result of the experiment can be found in figure 4.6. On the horizontal axis is $N$ while the time consumption in milliseconds divided by $N^2$ is on the vertical axis. As $N$ grows, the points of both model types resemble a horizontal line, just as expected. The relatively large time consumption for small $N$ can be explained with initialization overhead. The MEHMM linear regression fit for $N \geq 22$ is shown in the plot and again the slope is very close to zero. Again, the difference between the constants of the two algorithms is small.

We judge by the results of this experiment that the time consumption of the restricted Viterbi algorithm for MEHMMs is quadratic in $N$.

**Time consumption as a function of $|Q|$** In this experiment we return to using the HMM from figure 2.7 from which we generate an observable sequence of length $T = 1000$. We set $l$ to 0, $u$ to 50, run the restricted forward algorithm five times for every $|Q| = 3, 5, \cdots, 59$ and calculate the average time spent at each $|Q|$. We vary $|Q|$ by choosing and generating regular expressions that give us the desired amount of states when the automaton is constructed.

We expect the time consumption to be linear in $|Q|$, based on the experimental results of section 4.3.
The result is shown in figure 4.7. The horizontal axis holds $|Q|$ while the time in milliseconds divided by $|Q|$ is on the vertical axis. With the same arguments as in the previous experiments, we find that the plot and linear regression support our hypothesis that the time consumption of the MEHMM restricted Viterbi algorithm is linear in $|Q|$, even though the general time complexity remains quadratic in $|Q|$.

**Discussion of results** As explained earlier, we have decided to limit the experiments in this section to the MEHMM and HMM restricted Viterbi algorithms. We have performed the same experiments on the forward algorithms with similar results: the MEHMM restricted forward algorithm has the expected time complexity and a slightly higher constant than the HMM counterpart, due to extra overhead.

Contrary to what one might conclude based on a quick glance on the plots in this section, the presented results supports using MEHMMs instead of HMMs for most practical purposes. This is because the plots compares running times of the two algorithm with equal parameters, including $N$. In practice however, an MEHMM has fewer states than the corresponding HMM. For our simple way of generating corresponding HMMs and MEHMMs, as depicted in figure 4.5 and figure 4.16, the size of the MEHMM approaches one third of the corresponding HMM size for increasing $N$-values. Of course, this leads to a time and space consumption of around one third of the HMM algorithms. The added overhead of around 10% as shown in the plots is of little consequence in comparison. For more realistic and complex models however, we might need to translate an HMM to an MEHMM where it is not possible to reduce the number of states by that large a fraction, which may not render using the MEHMM much faster.
We would expect that the smaller size of the MEHMM would still outweigh the added overhead, though, and especially so for gene prediction, where HMMs are bound to contain groups of three states to account for codons.

To conclude, the results of the experiments in this section show MEHMMs to be a valid replacement for HMMs. The forward and Viterbi algorithms adjusted for MEHMMs run in unchanged time complexity. A bit of overhead is added, but we believe it to be outweighed by an improvement in the model size for most realistic models.

4.7 Summary of experimental results

In this chapter, we have performed experiments verifying the accuracy and time consumption of the restricted forward and Viterbi algorithms. Furthermore, we have examined the impact of our implemented improvements to the restricted forward and Viterbi algorithms for HMMs and MEHMMs. We have seen that each of our improvements are significantly beneficial to the practical resource consumption.
Chapter 5

Conclusion

In this thesis we have examined the restricted hidden Markov model algorithms, which are still under development.

First, we verified similarly to Tataru et al. [20] that, using simple models, the restricted algorithms perform better than the Viterbi algorithm.

We wanted to proceed in the direction of testing the restricted algorithms on more advanced models and real bacterial genomes, but found that the time and space consumption was a major issue. Therefore we instead turned our focus to improve the time and space consumption of the algorithms. Consequently, we did not have enough time to examine our improvements on realistic data.

In section 3.1.3 a preferable trade off between time and space managed us to save time by precomputing and storing automaton transition information. In section 3.2 we managed to greatly reduce to space consumption of the restricted Viterbi algorithm by only storing backtracking information that can be part of the most likely hidden state path. In section 3.3.2 we greatly reduced the time and space consumption of the forward algorithm by relaxing the accuracy by a very small amount. In section 3.4 we introduced MEHMMs to allow higher precision models using fewer hidden states. With only a small time consumption overhead, the reduction in the number of hidden states is expected to result in both a time and space consumption reduction.

While the theoretical time and space complexity of the algorithms remain unchanged, our results confirm that we have managed to greatly reduce the practical time and space consumption of the restricted algorithms. Our results show that the greatest contributors to this reduction are the Viterbi approximate linear space method and the forward cut off approach.

5.1 Future work

The question remains whether the improvements are enough to make the restricted algorithms viable in a realistic gene prediction context. Therefore, the next step we would take would be designing advanced MEHMMs and train them with real genomic data to examine the accuracy, as well as the time and space consumption in such a scenario. What also remains is to compare the restricted algorithms against other methods which have already been shown
to improve the prediction accuracy in other contexts. One such example is the 1-best method \cite{13}, which are shown to perform equal to or better than the Viterbi algorithm. Another is the posterior-Viterbi \cite{8}, which combines the posterior and Viterbi algorithms, and are shown to outperform other algorithms when predicting the topology of beta-barrel membrane proteins.
Bibliography


Appendix A

Occurrence distribution examples

The following two figures present examples of occurrence distributions from our practical examinations. Both of them are based on restricted forward algorithm runs using the model in figure 2.7 and the regular expression \( r = (NN(C_1|R_3))((C_3|R_1)NN) \). In the figures, \( P(k) \) is used as shorthand for \( P(O_r(Z_{1:T}) = k|X_{1:T}) \).

To emphasize the significant part of the distributions, we have set the maximum value of the \( k \)-axis to 25 % of the maximum possible number of occurrences, \( k_{\text{max}} \). For \( k \)-values larger than the shown interval, the probabilities are all very close to zero, and for \( k \)-values larger than \( k_{\text{max}} \), the probabilities are zero.

![Figure A.1](image)

Figure A.1: An occurrence distribution for a randomly generated sequence of length \( T = 1000 \). \( k_{\text{max}} \) for the distribution is 400.
Figure A.2: An occurrence distribution for a randomly generated sequence of length $T = 4000$. $k_{max}$ for the distribution is 1600.
Appendix B

Time complexity of the restricted Viterbi algorithm

Figure B.1: The relationship between $T$ and the time consumption of the restricted Viterbi algorithm.
Figure B.2: The relationship between $u$ and the time consumption of the restricted Viterbi algorithm.

Figure B.3: The relationship between $N$ and the time consumption of the restricted Viterbi algorithm.
Figure B.4: The relationship between $|Q|$ and the time consumption of the restricted Viterbi algorithm.

$f(|Q|) = a * |Q| + b$

a = -0.00160275
b = 48.2772
Appendix C

Division of work

This thesis has mostly been written in cooperation. Also, the main ideas have been developed together. To differentiate our work, we have chosen sections in the thesis, where one person is primarily responsible for presenting the content. It has been divided as follows

- Daniel: sections 2.1, 3.1.1, 3.3, 3.4.2, 4.5
- Jakob: sections 2.3, 3.1.2, 3.2, 3.4.3, 4.4