Speeding Up HMM Decoding Using Compression
Torben Muldvang Andersen, 20093713

Master’s Thesis, Computer Science
June 2015
Adviser: Christian N. S. Pedersen
Abstract

This master’s thesis describes the theory, implementation and evaluation of algorithms for hidden Markov model decoding. The classical algorithms used for decoding is the Viterbi algorithm and the posterior decoding algorithm. These are reformulated into series of matrix multiplications. By finding repeated substrings in the observation sequence multiplications can be avoided and thereby potential for speedup is obtained.

Sand et al. (2013) present zipHMMlib which uses byte-pair encoding to exploit repetitions. Using and efficient implementation of matrix multiplication a speedup is gained for the forward algorithm. Theory of reformulated versions of the Viterbi algorithm, the posterior decoding algorithm, and a so-called indexed posterior decoding algorithm is presented in this thesis in the context of zipHMMlib and the software library has been extended to include implementations of these algorithms.

The experiments show that a speedup can be gained in some situations. Speedups in the order of hundreds or thousands is observed for large repetitive observation sequences and small models, but in most cases the zipHMMlib version of the algorithm is between 0.5 and 10 times faster. The speedup becomes greater if the sequences are compressed once and used multiple times afterwards with e.g. different models.
I would like to thank my supervisor Christian N. S. Pedersen for help, guidance, and discussions that have been of great value for this project. For proof reading the thesis and for discussions of problems thanks go to Dan Søndergaard and Johan Sigfred Abildskov. Thanks to Mads Fabricius Schmidt for being a pleasant office buddy and making me meet at the office early (almost) every morning.

My six years at the university has been affected by many people. My education would not have been the same without them. Thanks to Dan Søndergaard with whom I have spent hundreds of hours working on projects, preparing for exams, cooking meals and drinking beers. Thanks to Christian N. S. Pedersen for some great courses making me interested in bioinformatics and machine learning. And lastly, thanks to all student organizations, especially TÅGEKAMMERET, for making the university a very enjoyable place to be.

Torben Muldvang Andersen,
Aarhus, Sunday 14th June, 2015.
Contents

1 Introduction .................................................. 1
   1.1 Outline ................................................. 2

2 Background .................................................. 3
   2.1 Hidden Markov Models ................................. 3
   2.2 Related Work .......................................... 6

3 Method ....................................................... 7
   3.1 Notation ................................................ 7
   3.2 The Original Viterbi Algorithm ..................... 8
   3.3 Viterbi as Linear Algebra ............................ 10
   3.4 Exploiting Repetitions ............................... 11
   3.5 Original Posterior Decoding ......................... 18
   3.6 Posterior Decoding as Linear Algebra ............. 19
   3.7 Problems in Exploiting Repetitions ................. 21
   3.8 Indexed Posterior Decoding ......................... 23

4 Implementation .............................................. 31
   4.1 Description of the Implementation .................. 31
   4.2 Usage .................................................. 32

5 Experiments ................................................ 35
   5.1 Data ................................................... 35
   5.2 Experimental Setup ................................... 36
   5.3 Compression Ratio .................................... 36
   5.4 Preprocessing .......................................... 38
   5.5 Viterbi ................................................ 39
   5.6 Posterior Decoding .................................... 50
CONTENTS

5.7 Indexed Posterior Decoding ......................................... 53

6 Conclusion ................................................................. 61

Bibliography ................................................................. 65
Chapter 1

Introduction

A hidden Markov model is a statistical model that during many years has proven to be useful in many areas such as speech recognition, character recognition, face recognition, and various areas of bioinformatics, where it has been used for e.g. gene finding, modeling protein structures and sequence alignment. Examples of this can be found in Rabiner (1989), Agazzi and Kuo (1993), Nefian and Hayes III (1998), Burge and Karlin (1997), and Eddy (1998).

In general, algorithms on hidden Markov models are efficient and can be used for large data sets in e.g. genome wide analysis, but the running time of the analysis is still measured in hours or days. Since the introduction of next generation sequencing the amount of available biological data has increased tremendously, and making faster algorithms for hidden Markov models is thus of great value.

Lifshits et al. (2009) and Sand et al. (2013) use compression of the input data and minor changes in the algorithms to speed up the analysis. While Lifshits et al. (2009) mainly analyze their method theoretically, Sand et al. (2013) experimentally proves that the method has a great potential for performance improvement to the forward algorithm.

The goal of this thesis is to complement the work made by Sand et al. (2013) by extending the theory and implementation to include the Viterbi and posterior decoding algorithms and by making experiments validating the performance of the implementations.
1.1 Outline

The thesis is structured as follows.

Chapter 2: Background provides an overview of the work made by Lifshits et al. (2009) and Sand et al. (2013) and the differences between these.

Chapter 3: Method contains theory of the original formulations of the Viterbi and posterior decoding algorithms along with theoretical descriptions and analyses of these algorithms using compression.

Chapter 4: Implementation gives an overview of the implementation and instructions on how to use the library.

Chapter 5: Experiments checks that the implementation of the algorithms fit the theoretical running time and shows the performance gain obtained from the compression.

Chapter 6: Conclusion provides an overview of the developed theory, the experiments and the results of these. Ideas on how to make the framework better are also given.
Chapter 2

Background

In this chapter the preliminary theory of hidden Markov models and the work by Lifshits et al. (2009) and Sand et al. (2013) is described.

2.1 Hidden Markov Models

Hidden Markov models (HMMs) were first introduced and studied in the end of the 1960s and the beginning of the 1970s. Since then they have become very popular and have proven to be both effective and efficient in many applications (Rabiner, 1989).

![Markov Chain Diagram]

Figure 2.1: An example of a Markov chain.
Think of a system that can be described as a being in one of \( N \) states at any time. Let the set of states be notated as \( \mathcal{H} = \{ h_1, h_2, \ldots, h_N \} \). In figure 2.1 an example with three states is shown. In discrete time steps denoted \( t = 1, 2, \ldots \) the system changes from one state to another. For each state there is a certain probability of making a transition to any other state. This can be represented as a matrix \( A = \{ a_{ij} \}_{1 \leq i, j \leq N} \) with \( a_{ij} \in [0, 1] \) and \( \sum_{j=1}^{N} a_{ij} = 1 \). A system like this where the next state is only dependent on the current is called a Markov process, or more precisely for discrete-time and finite state spaces: a Markov chain. The non-zero transition probabilities are shown as arrows in figure 2.1. The initial state, at time \( t = 1 \), is determined from a vector \( \Pi = (\pi_i)_{1 \leq i \leq N} \) where \( \sum_{i=1}^{N} \pi_i = 1 \) with \( \pi_i \) being the probability of starting in state \( h_i \).

Having established the above system with \( A \) and \( \Pi \) given, one might ask questions like what the probability of observing a certain sequence of states \( X_{1:T} \) during time steps \( t = 1, \ldots, T \) is. As an example assume that the model in figure 2.1 is a model of the weather with \( h_1 \) being rainy, \( h_2 \) being cloudy, and \( h_3 \) being sunny weather. The probability of a sequence of observations \( X_{1:4} = x_1 x_2 x_3 x_4 \) with \( x_i \in \mathcal{H} \) over four days, \( t = 1, 2, 3, 4 \), can easily be computed by multiplying the initial probability to the transition probabilities.

The above formality is too limited for many applications since the states often do not correspond to the actual observable events. To overcome this problem a distinction between observables and states is made. The observables are a probabilistic function of the state. Given \( O = o_1, o_2, \ldots, o_M \) being a set of \( M \) observables an emission matrix \( B = \{ b_{ij} \}_{1 \leq i \leq N} \) is defined, where \( b_{ij} \) is the probability of observing \( o_j \) when the system is in state \( h_i \). The set of states, \( \mathcal{H} \), can now be called “hidden” states since they are not observable. An example of an HMM is shown in figure 2.2. The figure is similar to figure 2.1 except for the probabilities added for each hidden state.

For modeling weather we can now use the fact that the observables (rainy, cloudy, and sunny weather) depend on the air pressure. This can be modeled as \( h_1 \) being low pressure, \( h_2 \) being high pressure, \( o_1 \) being rainy, \( o_2 \) being cloudy, and \( o_3 \) being sunny weather. As before the probability of a sequence of observations \( Y_{1:4} = y_1 y_2 y_3 y_4 \) with \( y_i \in O \) can be found, but it is a harder than before since an observation sequence can be generated in many ways by the models.

Rabiner (1989) states three key problems for using hidden Markov models in real world applications, with the first being the one just mentioned.

1. Given the observation sequence \( Y_{1:T} = y_1 y_2 \ldots y_T \), and the model \( \lambda = (A, B, \Pi) \), how we compute \( \Pr(O | \lambda) \), the probability of the observation
2.1. HIDDEN MARKOV MODELS

Figure 2.2: An example of a hidden Markov model.

sequence.

2. Given the observation sequence \( Y_{1:T} = y_1 y_2 \ldots y_T \), how do we choose a state sequence \( X_{1:T} = x_1 x_2 \ldots x_T \) which is optimal in some meaningful sense.

3. How we adjust the model parameters \( \lambda = (A, B, \Pi) \) to maximize \( \Pr(O | \lambda) \).

The first problem of finding the joint probability of an observation sequence \( Y_{1:T} \) is solved by the forward algorithm. This was implemented in zipHMMlib by Sand et al. (2013).

For solving the second problem (also called HMM decoding), many methods may exist as it depends on which optimality criterion is chosen. However, two widely used methods exist. One solution is to look at all possible sequences of states emitting the observation sequence and pick the most likely one. This problem is solved efficiently by the Viterbi algorithm. Another method is to choose the states \( x_i \) that are individually most likely. In this way the expected number of correct states is maximized, but the state sequence might not be valid, since the transition probability from \( x_t \) to \( x_{t+1} \) can be zero. This is also called the posterior decoding and is solved by forward-backward algorithm.

Different solutions also exist to the third problem. One solution is Baum-Welch training that is an iterative procedure that maximizes the probability of the observation sequence by iteratively re-estimating the parameters such that the new model is more likely to generate the observation sequence than the previous model. Another solution known as Viterbi training set the parameters of the model such that the most likely sequences of states is becoming larger in each iteration.
2.2 Related Work

Lifshits et al. (2009) present a method for speeding up the dynamic programming algorithms used with HMMs, namely the forward-backward algorithms and the Viterbi algorithm. The approach is based on finding repeated substrings in the observation sequence. Five different algorithms for finding substrings are proposed: Four Russians method, run length encoding, Lempel-Ziv parsing, grammar-based compression and byte-pair encoding. In the article the forward-backward and the Viterbi algorithms are reformulated into a series of matrix multiplications. The overall idea is that the repeated substrings correspond to repeated matrix multiplications and by finding the repeated substrings multiplications are avoided. A single experiment has been performed on DNA sequences showing a speedup of the Viterbi algorithm when using an improved Lempel-Ziv parsing without backtracking. No further experiments have been made for the other algorithms and no code has been made available.

Sand et al. (2013) presents zipHMMlib, a highly optimized HMM library for speeding up the forward algorithm. The speedup is achieved by finding repeated substrings using byte-pair encoding. Much of the theory in this paper relies on Lifshits et al. (2009), but is extended to make the computations numerically stable. Furthermore, the code is available as an open source library with bindings for both Python and R.

This thesis extends the work made by Sand et al. (2013) to also include a efficient Viterbi algorithm and a posterior decoding algorithm based on byte-pair encoding as developed theoretically by Lifshits et al. (2009). The theory will also be extended a bit to make the computation numerically stable and the experiments will be more extensive.
Chapter 3

Method

This chapter describes the theory used and developed in this thesis. The notation for HMMs introduced in section 2.1 is written out as this is used extensively in the following sections.

The first contribution of this thesis is the reformulation of the original Viterbi algorithm to fit into the zipHMMlib framework, to analyze the running time, and to make the computations numerically stable.

Next, theory for the posterior decoding is described. As only the forward algorithm has been implemented in zipHMMlib the backward algorithm is described here since it is required to compute the posterior decoding. However, it is has not been possible to develop a posterior decoding algorithm for exploiting repetitions.

The final theoretical contribution is the theory for an indexed posterior decoding. A theoretical speedup can be gained by exploiting repetitions in the entire input sequence for the forward and backward algorithms and only return the posterior decoding for a part of the input sequence.

3.1 Notation

Recall the introduction to HMM’s given in section 2.1. An HMM can generate an observation sequence \( Y_{1:T} = y_1 y_2 \ldots y_T \) with \( y_i \in O \) by traversing hidden states and emitting symbols. There exist one or more paths of hidden states \( X_{1:T} = x_1 x_2 \ldots x_T \) with \( x_i \in H \) for each observation sequence generated by the HMM. Formally a HMM is defined as

- \( H = \{ h_1, h_2, \ldots, h_N \} \), a finite alphabet of hidden states;
- \( O = \{ o_1, o_2, \ldots, o_M \} \), a finite alphabet of observables;
• a vector $\Pi = \{\pi_i\}_{1 \leq i \leq N}$, where $\pi_i = \Pr(x_1 = h_i)$ is the probability of the model starting in hidden state $h_i$;

• a matrix $A = \{a_{ij}\}_{1 \leq i, j \leq N}$, where $a_{ij} = \Pr(x_t = h_j \mid x_{t-1} = h_i)$ is the probability of a transition from state $h_i$ to state $h_j$;

• a matrix $B = \{b_{ij}\}_{1 \leq j \leq M}$, where $b_{ij} = \Pr(y_t = o_j \mid x_t = h_i)$ is the probability of state $h_i$ emitting $o_j$.

An HMM is parametrized by $\Pi, A,$ and $B$ and is denoted by $\lambda = (\Pi, A, B)$.

### 3.2 The Original Viterbi Algorithm

As mentioned in section 2.1 an observation sequence $Y_{1:T}$ can be generated in different ways since multiple hidden states can have a nonzero probability of omitting the same symbol. Given a model $\lambda$ and an observation sequence an interesting question might be: of all possible paths of hidden sequences which one is the most likely and how likely is it? This question can be solved the Viterbi algorithm.

The Viterbi algorithm finds the probability of the most likely sequence of hidden states emitting an observation sequence $Y_{1:T}$ given a model $\lambda$. This can be expressed as maximizing the probability of the observation and hidden sequences for all possible hidden sequences, that is $\max_{X_{1:T}} \Pr(Y_{1:T}, X_{1:T} = x_{1:T} \mid \lambda)$.

To compute this, consider the Viterbi variable $\delta_t(x_t)$ defined as

$$\delta_t(x_t) = \max_{x_{1:t}} \Pr(Y_{1:t}, X_{1:t} = x_{1:t} \mid \lambda).$$

This is the probability of the partial observation sequence $Y_{1:t}$ at time $t$ and the most likely sequence of states $X_{1:t}$. Since the probability of making a transition to the next state is only dependent on the current state, this formula can be solved inductively by

$$\delta_1(x_1) = \pi_{x_1} b_{x_1, y_1}$$
$$\delta_t(x_t) = b_{x_t, y_t} \max_{x_{t-1}} \delta_{t-1}(x_{t-1}) a_{x_{t-1}, x_t}. \tag{3.1}$$

This is computed efficiently by making $\delta$ a table of size $N \times T$ and by computing this table column-wise from left to right using dynamic programming. After populating $\delta$, the probability of the most likely sequence of states can be computed as $\max_{x_T} \delta_T(x_T)$, that is the largest probability in the last column.
The most likely sequence of hidden states \( V_{1:T} = v_1 v_2 \ldots v_T \), also known as the Viterbi path, can be computed similarly. Another table \( \Psi \) of the same size as \( \delta \) is computed along \( \delta \) with pointers from each state in a column to the maximizing previous state:

\[
\Psi_1(x_1) = 0 \\
\Psi_t(x_t) = \arg\max_{x_{t-1}} \delta_{t-1}(x_{t-1}) a_{x_{t-1}, x_t}.
\]

When the last columns have been computed in \( \delta \) and \( \Psi \) the last state in the Viterbi path is found using

\[
v_T = \arg\max_{x_T} \delta_T(x_T).
\]

From this state the Viterbi path \( V_{1:T} \) can be found by backtracking the pointers in \( \Psi \) using

\[
v_{t-1} = \Psi_t(v_t).
\]

### 3.2.1 Running Time

Note that if only the probability is needed, only the last computed column of \( \delta \) needs to be stored since the recursion in equation (3.1) only needs the previous column to compute the new one. Hence, the space consumption of the algorithm will be the space needed for two columns and the observation sequence, that is \( O(N + T) \). To compute a cell in \( \delta \) the algorithm maximizes over all cells in the previous column resulting in a running time of \( O(N^2 T) \).

The space consumption of this backtracking algorithm is the size of \( \Psi \) which is \( O(NT) \). Backtracking is computed in \( O(T) \) time, since it is only lookups in \( \Psi \).

A space consumption of \( O(NT) \) might be too large for some applications; especially the \( T \) factor can become large. Another way to backtrack is described by Tarnas and Hughey (1998). Using their approach the space consumption is reduced to \( O(N \sqrt{T}) \) while the asymptotic running time of the algorithm is preserved. The \( \delta \) table is split into blocks of size \( s \). While computing \( \delta \), the first column of each block is stored, while the rest is thrown away. These columns are called checkpoints. Since computing a column only requires the previous column as seen in equation (3.1), the part of the \( \delta \) table corresponding to a block can be computed using the checkpoint, and thereby \( \Psi \) can also be computed. When the last column in \( \delta \) has been computed, \( v_T \) can be found as described above. The blocks are then recomputed one at a time from right to left. When the \( \Psi \) table for a block has been recomputed it is backtracked.
as before. Hence, only one block is kept in memory at a time. If \( s = \sqrt{T} \) is chosen, the space consumption is \( O(NT/s) = O(NT/\sqrt{T}) = O(N\sqrt{T}) \). \( \delta \) is now computed two times, so the algorithm is slower, but it keeps having the same asymptotic running time of \( O(N^2T) \).

### 3.3 Viterbi as Linear Algebra

Before exploiting repetitions in the observation sequence \( Y_{1:T} \) the original Viterbi algorithm is reformulated into linear algebra. The goal is to express equation (3.1) as a series of matrix multiplications as Sand et al. (2013) and Lifshits et al. (2009) do.

For each symbol \( o_i \in O \), let \( B_{o_i} \) be the diagonal matrix having the emission probabilities of \( o_i \) on the diagonal:

\[
B_{o_i} = \begin{bmatrix}
b_{1,o_i} & & \\
& b_{2,o_i} & \\
& & \ddots \\
& & & b_{N,o_i}
\end{bmatrix}
\]

and let

\[
C_1 = B_{y_1}\pi,
\]

\[
C_{o_i} = B_{o_i}A^T
\]

where \( A^T \) is the transpose of \( A \). Each matrix \( C_{o_i} \) is of size \( N \times N \) and an entry \( C_{o_i}(k,l) \) is the probability of making a transition from state \( k \) to state \( l \) and emit symbol \( o_i \).

Equation (3.1) can now be expressed by these matrices. It is trivial to see that the base case is \( C_1 \). For the recursive case we note that

\[
\delta_t(x_t) = b_{x_t,y_t} \max_{x_{t-1}} \delta_{t-1}(x_{t-1})a_{x_{t-1},x_t}
\]

\[
= \max_{x_{t-1}} b_{x_t,y_t}\delta_{t-1}(x_{t-1})a_{x_{t-1},x_t}.
\]

Hence, \( \delta_t \) can be computed using \( C_{y_t} \) and \( \delta_{t-1} \) using the recursive formula

\[
\delta_1 = C_1
\]

\[
\delta_t = C_{y_t} \odot \delta_{t-1} = C_{y_t} \odot C_{y_{t-1}} \odot \cdots \odot C_1,
\]

where \( \odot \) is the max-times matrix multiplication defined as

\[
(X \odot Y)_{ij} = \max_k X_{ik}Y_{kj}.
\]
The implementation of the original Viterbi algorithm corresponds to compute this from right to left, i.e. the $\delta$ table is computed for $t = 1, t = 2, \ldots, t = T$. This will also be most efficient since $C_1$ is a vector that when multiplied by the matrix $C_2$ will result in a new vector. Hence, computing $\delta_t$ from right to left will result in $t$ matrix-vector multiplications, while computing it from left to right will result in $t$ matrix-matrix multiplications that are less efficient to compute.

Backtracking is achieved in the same way as for the original Viterbi algorithm as described in section 3.2 using an argmax-times matrix multiplication.

### 3.3.1 Running Time

The space consumption and running time of this algorithm has changed, compared to the original Viterbi algorithm. For each symbol $o_i$ in the alphabet of observables the corresponding matrix $B_{o_i}$ is created, thus requiring $O(MN^2)$ space. The $C_{o_i}$ matrices require the same amount of space: $O(MN^2)$. $C_1$ is a vector, thus only requiring $O(N)$ space. If equation (3.2) is evaluated from right to left it corresponds to a series of matrix-vector multiplications resulting in a vector of size $O(N)$. In total this requires $O(2MN^2 + 2N) = O(MN^2)$ space. If backtracking is required, the space consumption will be increased by the size of the $\Psi$ table that has size $O(NT)$. The total the space consumption with backtracking enabled is $O(MN^2 + NT)$, or if the checkpoint trick is used, $O(MN^2 + N\sqrt{T})$.

Likewise, the running time has changed. Creating the $B_{o_i}$ matrices takes $O(MN^2)$ time. Computing the $C_{o_i}$ matrices takes time $O(MN^3)$ due to the matrix multiplication. Computing $C_1$ takes $O(N^2)$ time. Again, if equation (3.2) is evaluated from right to left, it will be matrix-vector multiplications requiring $O(N^2T)$ time. Backtracking can be computed in either $O(NT)$ or $O(N^2T)$ time using the $\Psi$ table. In total this becomes $O(MN^2 + MN^3 + N^2 + N^2T) = O(MN^3 + N^2T)$.

### 3.4 Exploiting Repetitions

In this section it is shown how to exploit repetitions in the observation sequence to make the above algorithm run faster using byte-pair encoding.

#### 3.4.1 Byte-Pair Encoding

Byte-pair encoding is a simple data compression method. The most common pair consecutive bytes in the data is replaced by a byte that does not exist
in the data. This is repeated until either all new bytes are used or the most common pair does not appear frequently in the data. The use of bytes can easily be replaced by used of characters or integers. For example, the input sequence 01012012 would first be encoded into 33232 by substituting 01 with 3; in the next iteration it would be encoded into 344 by substituting 32 by 4 etc.

3.4.2 Using Byte-Pair Encoding to Speed Up Viterbi

The Viterbi algorithm achieves a speedup using byte-pair encoding in the following way. At the beginning of an iteration of the encoding process let \(o_{i,j} \in O \times O\) be the most frequently occurring pair of symbols in \(Y_{1:T}\) and let \(n_{o_{i,j}}\) be the number of occurrences. \(o_{i,j}\) is substituted by a new symbol \(o_{M+1}\) and the length of \(Y_{1:T}\) is thereby reduced by \(n_{o_{i,j}}\). All \(n_{o_{i,j}}\) occurrences of \(C_{o_i} \odot C_{o_j}\) in equation (3.2) can be replaced by the new matrix

\[
C_{o_{M+1}} = C_{o_i} \odot C_{o_j}.
\]

Hence, the number of matrix multiplications is also reduced by \(n_{o_{i,j}}\). The byte-pair encoding continues with more iterations until \(n_{o_{i,j}}\) becomes too small to give a speedup. This is discussed later. The result of this is a new sequence \(Y'_{1:T'}\) over the new alphabet

\[
O' = \{o_1, o_2, \ldots, o_M, o_{M+1}, o_{M+2}, \ldots, o_{M'}\} = \{o_1, o_2, \ldots, o_M, (l_1, r_1), (l_2, r_2), \ldots, (l_{M'-M}, r_{M'-M})\},
\]

where \(l_i, r_i \in \{o_1, o_2, \ldots, o_{M+i-1}\}\).

This method is split up into two separate computations. First the sequence is encoded as just discussed. As the encoding of the sequence is independent of the HMM the encoded sequence can be saved to disk for later use and this computation can be called preprocessing. The second part consist of the actual Viterbi algorithm and is split into two stages. The first stage is the computation of \(C_{o_i}\) for \(i = 1, \ldots, M\) and then \(C_{o_j}\) for increasing \(i = M + 1, \ldots, M'\) by \(C_{o_j} = C_{l_i} \odot C_{r_i}\). In the second stage \(\delta_T\) is computed by

\[
\delta_T = C_{y_{T'}} \odot C_{y_{T'-1}} \odot \cdots \odot C_{y_2} \odot C_1.
\]

An illustration of this method is given in figure 3.1. As seen, when the computation has finished only a subset of the columns of \(\delta\) has been computed. As an example consider the observation sequence \(Y_{1:10} = 1010001001\) over the alphabet \(O = \{0, 1\}\) from figure 3.1. This is being compressed by first replacing the pair 10 by a new symbol 2, and the replacing 20 by 3. Hence,
3.4. EXPLOITING REPETITIONS

![Diagram](image)

**Figure 3.1:** Exploiting repetitions in the observation sequence to speed up the Viterbi algorithm. The rectangles correspond to matrices and vectors. To fit equation (3.4) the observation sequence is written from right to left.

the original observation sequence of length 10 becomes compressed to a sequence $Y'_{1:5} = 33021$ of length 5. Now, for each symbol in the new alphabet $O' = \{0, 1, 2, 3\}$ the $C_{o_i}$ matrices and the $C_1$ matrix are constructed. These corresponds to the rectangles with a thick line. The solid lines correspond to the matrix multiplications performed, while the dotted lines is the work saved due to redundancy. Finally, columns of $\delta$ is computed by performing matrix-vector multiplications from right to left. The gray columns correspond to the amount of work saved in the Viterbi computation.

### 3.4.3 Backtracking

Obtaining the Viterbi path $V_{1:T} = v_1, v_2, \ldots, v_T$ of $Y_{1:T}$ is no longer as simple as for the original Viterbi algorithm. Using the standard backtracking methods only the Viterbi path $V'_{1:T'} = v'_1, v'_2, \ldots, v'_T$, of the compressed sequence $Y'_{1:T'}$ can be found. The path corresponds to the columns of $\delta$ in figure 3.1 that have been computed. It turns out that $V_{1:T}$ can be inferred from $V'_{1:T'}$ since $V'_{1:T'}$ is a subsequence of $V_{1:T}$ as described by Lifshits et al. (2009). To do this a set of matrices $R_{o_i}$ is kept along the $C_{o_i}$ matrices for each of the new symbols $o_{M+1}, \ldots, o_{M'-M}$ defined as

$$R_{o_{M+i}}(m, n) = \arg\max_k (C_{i_i}(m, k) \odot C_{r_i}(k, n)).$$

This definition is similar to the definition of the $C_{o_i}$ matrices from equation (3.3), but instead of storing the maximum value, the state that results in the maximum
value is stored.

Now, for each occurrence of a new symbol \( \omega_{M+i} = (l_i, r_i) \) in \( Y'_{1:T} \), we know the start state \( v_l \) and the end state \( v_r \) from \( V'_{1:T} \) such that \( v_l \) is the state immediately before \( l_i \) was emitted and \( v_r \) is the state after \( r_i \) was emitted. Hence, we need to find the most likely state where \( l_i \) is emitted. This state is easily obtained since it is stored at \( R_{\omega_{M+i}}(v_l, v_r) \). In the case where one or both of \( l_i, r_i \not\in O \) we can apply this method recursively to obtain \( V_{1:T} \). This method is illustrated in figure 3.2. The example is continued from figure 3.1. When the \( \delta \) table has been computed for the compressed observation sequence \( Y'_{1:T} \), the sequence of states \( V'_{1:T} \) can be found using the same backtracking methods as for the Viterbi algorithm not exploiting repetitions. The remaining part of the Viterbi path can be computed from right to left using the \( R \) matrices. As an example consider the computation of \( v_5 \) and \( v_6 \). When scanning through \( Y'_{1:5} \) from right to left the symbol \( Y'_5 = 3 \) is encountered at some point in time. As \( 3 \not\in O \) it is a new symbol introduced by the compression and in this case it corresponds to the pair 02. The most likely state when emitting 2 is \( v'_4 \) (corresponding to \( v_7 \)) and the most likely state before emitting 0 is \( v'_5 \) (corresponding to \( v_4 \)). The state when emitting 0 is stored in \( R_3(v_4, v_7) \), since this cell in \( R_3 \) resulted in the highest probability.

Since \( V'_{1:T} \) can be found in two different ways as discussed in section 3.2, three different variations of the Viterbi algorithm are defined:

- **Viterbi\_L** compresses the sequence and only computes the probability,
- **Viterbi\_P** also backtracks using a table of pointers,
- **Viterbi\_PM** saves memory on backtracking using checkpoints.

The same variations are defined for the original Viterbi algorithm.

### 3.4.4 Running Time

The space consumption of this algorithm is comparable to the Viterbi algorithm without compression enabled. The number of \( C \) matrices has changed from \( M \) to \( M' \) thus requiring \( O(M'N^2) \) space. The introduction of the \( R \) matrices does not change this since the \( C \) matrices are of similar size. Thus the space consumption of Viterbi\_L is \( O(M'N^2 + T') \). The table used for the simple backtracking has decreased to size \( O(NT') \). The Viterbi path has size \( T \) resulting in a total space consumption of \( O(M'N^2 + NT' + T) \) for Viterbi\_P. For Viterbi\_PM the space consumption is \( O(M'N^2 + N\sqrt{T'} + T) \).

With similar arguments the running time of computing the probability has changed to \( O(M'N^3 + N^2T') \). Using standard backtracking methods from the
original algorithm part of the Viterbi path is found in $O(N^2T')$ time and using the $R$ matrices the rest can be obtained in $O(T)$ time. Hence, the running time with backtracking enabled is $O(M'N^3 + N^2T' + T)$.

The running time of the preprocessing part is $O((M' - M)T)$, since one new symbol is introduced by scanning the input sequence for pairs. Note that the $M' - M$ factor depends on the input. For sequences with many repetitions the byte-pair encoding will continue for more iterations than for sequences with few repetitions.

An overview of the theoretical running times and space consumptions is shown in table 3.1.

### 3.4.5 Saving the Compressed Sequence and Computed Matrices

As mentioned, the observation sequence encoding is independent of the HMM and can be saved for use with different HMMs. This makes it possible to compress the sequences once and then make use of them multiple times with different HMMs. This has been implemented by Sand et al. (2013).
Table 3.1: Running time and space consumption of the different variations of the Viterbi algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running time</th>
<th>Space consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original_L</td>
<td>$O(N^2 T)$</td>
<td>$O(N + T)$</td>
</tr>
<tr>
<td>Original_P</td>
<td>$O(N^2 T)$</td>
<td>$O(N T)$</td>
</tr>
<tr>
<td>Original_Pm</td>
<td>$O(N^2 T)$</td>
<td>$O(N \sqrt{T})$</td>
</tr>
<tr>
<td>Viterbi_L</td>
<td>$O(M'N^3 + N^2 T')$</td>
<td>$O(M'N^2 + T')$</td>
</tr>
<tr>
<td>Viterbi_P</td>
<td>$O(M'N^3 + N^2 T' + T)$</td>
<td>$O(M'N^2 + NT' + T)$</td>
</tr>
<tr>
<td>Viterbi_Pm</td>
<td>$O(M'N^3 + N^2 T' + T)$</td>
<td>$O(M'N^2 + N \sqrt{T'} + T)$</td>
</tr>
</tbody>
</table>

Lifshits et al. (2009) suggest constructing the substitution table for the compression based on a set of representative sequences. Then the $C_0$ matrices could be computed beforehand and also saved to the disk. This could be useful in e.g. bioinformatics where one might have a lot of genome data from multiple individuals of the same species that are to be analyzed using a single model. In this case the observation sequences are very similar and the compression can work very well by finding most common pairs only in a subset of the observation sequences. This has however not been implemented in this thesis.

3.4.6 Numerical Stability

All matrices contain probabilities, i.e. they are between 0 and 1. Multiplying them together results in even smaller numbers. Since the value is stored as a IEEE 754 floating point format there is limited precision and the computations will quickly underflow. Sand et al. (2013) describes how to avoid underflow in terms of the forward algorithm. It turns out that it is easier to avoid for the Viterbi algorithm.

To circumvent underflow in the original Viterbi algorithm all probabilities are converted to log-space. So, instead of computing $\delta_T$, $\log \delta_T$ is computed. Using the property that $\log(AB) = \log A + \log B$ the multiplications are turned into additions, which makes the algorithm more resilient to underflow. If using non-transformed probabilities problems with underflow will typically occur for observations sequences of lengths in the order for hundreds, while for the log-transformed probabilities numerical problems will most likely never occur in practice.

The same idea can be used for the matrix based approach, where equa-
tion (3.4) is rewritten as

\[
\log \delta_T = \log \left( C_{y'r} \odot C_{y'r-1} \odot \cdots \odot C_{y2} \odot C_1 \right) \\
= \log C_{y'r} \oplus \log C_{y'r-1} \oplus \cdots \oplus \log C_{y2} \oplus \log C_1,
\]

and the C matrices are rewritten as

\[
C_1 = \log B_{y1} \pi, \\
C_{oi} = \log B_o A^T, \quad \text{for } 1 \leq i \leq M \\
C_{oM-M} = C_i \oplus C_{r,r}, \quad \text{for } 1 \leq i \leq M' - M
\]

where \( \oplus \) is defined as \((A \oplus B)_{ij} = \max_k \left( \log A_{ik} + \log B_{kj} \right)\).

### 3.4.7 Compression Stopping Criterion

Using the byte-pair compression method a sequence can be compressed until only a single character is present. However, as more symbols are added to the alphabet the most common pair of symbols will be less and less frequent as the sequence becomes smaller. This means that the first iterations of the compression will make the sequence much shorter whereas the later iterations will not compress the sequence as effectively as the first. Hence, for the first iterations of the compression a large increase in the speedup of Viterbi is expected, where the increase in speedup of later iterations will be more modest, so the compression may take more time than the time saved.

Sand et al. (2013) describe a method on how to determine when the compression should stop for the forward algorithm. The user of the program supplies an estimate \( e \) of how many times the compressed sequence is going to be used. For a low \( e \) the algorithm should not spend too much time on compressing the sequence while for a high value of \( e \), the algorithm can “afford” spending more time on compression since the time spent will be earned again when running the actual algorithm.

Furthermore, the user specifies a parameter \( N_{\min} \) that is the number of states in the model that will be used. In the compression phase it is estimated how much time will be spent on the \( e \) executions of the actual algorithm by measuring the time \( t_{\text{mm}} \) it takes to do a matrix-matrix multiplication and the time \( t_{\text{mv}} \) of a matrix-vector multiplication with matrices of size \( N_{\min} \). Hence, it can be estimated how long it will take to execute the actual algorithm. When the time it takes to make another iteration of the compression is larger than the estimate of the time saved in the execution of the algorithm, the compression is stopped.
Assume that in iteration $i$ of the compression the number of occurrences of the most common pair of symbols is $p_i$. Then the time saved on the actual algorithm for iteration $i$ is $e(t_{mv} p_i - t_{mm})$ since $p_i$ matrix-vector multiplications is avoided, but a new symbol is introduced and thereby also a matrix-matrix multiplication. Let $\text{pre}_i$ be the running time of iteration $i$ of the compression. When at an iteration $j$ $\text{pre}_j > e(t_{mv} p_i - t_{mm})$ is true, the compression is stopped.

If the compressed sequence is to be used with models of different sizes a list of parameters $(N_{\text{min}}^1, N_{\text{min}}^2, \ldots)$ can be specified and the input sequence will be compressed for each of these model sizes instead of provided a single size $N_{\text{min}}$. If no $N_{\text{min}}$ parameter is provided, the compression continues until the frequency of the most common pair is the same in two consecutive iterations.

This method is reused for the Viterbi algorithm. Instead of measuring the time it takes to do matrix-matrix multiplications and matrix-vector multiplications, the time measured for max-times multiplications is used.

This simple algorithm does not take backtracking into account since the time estimated is only for the computation of the probability. By letting the user specify whether backtracking is required the time estimation could also involve an estimate on how much time will be spent on backtracking by estimating the time it takes to allocate an array of size $T$ and infer the Viterbi path $V_{1:T}$ from the compressed path $V'_{1:T}$. This has not been implemented.

## 3.5 Original Posterior Decoding

The Viterbi algorithm described above finds the most likely path of hidden states through the HMM emitting the input sequence. Another kind of decoding is called posterior decoding. Posterior decoding finds the most likely state at the time the symbol is emitted. Formally the most likely state $p_t \in \mathcal{H}$ being in when emitting symbol $y_t$ can be written as

$$ p_t = \arg \max_{x_t \in \mathcal{H}} \Pr(x_t | Y_{1:T}, \lambda) . $$

The posterior decoding $P_{1:T}$ is a sequence of hidden states that might not be a valid path according to the transitions of the model, like the Viterbi decoding, but it is useful for some applications.

The posterior decoding is efficiently computed using the forward-backward algorithms. The forward-backward algorithms are comparable to the Viterbi algorithm. Instead of computing the probability of the most likely path they compute the joint probability of all paths, that is

$$ \Pr(Y_{1:T} | \lambda) = \sum_{X_{1:T}} \Pr(Y_{1:T}, X_{1:T} = x_{1:T} | \lambda) . $$
This is computed using two tables $\alpha$ and $\beta$.

For the original forward algorithm a table, $\alpha$, with entries

$$a_t(x_t) = \Pr(Y_{1:t}, X_t = x_t \mid \lambda) = \sum_{x_{t-1}} \Pr(Y_{1:t}, X_{1:t} = x_{1:t} \mid \lambda)$$

is computed. As seen, this is very similar to the Viterbi algorithm. Instead of computing the maximum over all previous states the sum is computed. $\alpha$ is computed recursively column by column from left to right like the Viterbi algorithm using

$$\begin{align*}
\alpha_1(x_1) &= \pi_{x_1} b_{x_1, y_1} \\
\alpha_t(x_t) &= b_{x_t, y_t} \sum_{x_{t-1}} \alpha_{t-1}(x_{t-1}) a_{x_{t-1}, x_t}.
\end{align*} \tag{3.5}$$

Whereas the forward algorithm computes the table from left to right, i.e. it computes the joint probability of emitting $y_t$ after having emitted $Y_{1:t-1}$, the backward algorithm computes the joint probability of emitting $y_t$ and then emitting $Y_{t+1:T}$. This corresponds to computing the table from right to left. The backward table is called $\beta$. The recursion can be written as

$$\begin{align*}
\beta_T(x_T) &= (1, 1, \ldots, 1), \\
\beta_t(x_t) &= b_{x_t, y_t} \sum_{x_{t+1}} \beta_{t+1}(x_{t+1}) a_{x_{t+1}, x_t}.
\end{align*} \tag{3.6}$$

The rightmost column is filled with 1’s since the initial state is assumed given.

After having computed the forward and backward tables it is now possible to compute the posterior decoding $P_{1:T}$ given by

$$p_t = \arg \max_{x_t \in \mathcal{H}} \Pr(x_t \mid Y_{1:t}) = \arg \max_{x_t \in \mathcal{H}} \frac{\alpha(x_t) \beta(x_t)}{\Pr(Y_{1:t})}.$$ 

Since $\Pr(Y_{1:t})$ is the same for all $t$ it can be treated as a constant result in the formula

$$p_t = \arg \max_{x_t \in \mathcal{H}} \alpha(x_t) \beta(x_t). \tag{3.7}$$

### 3.6 Posterior Decoding as Linear Algebra

The main part of the posterior decoding algorithm described in the previous section is computing the forward and backward tables. Once this is computed,
it is trivial to compute the posterior decoding. The forward and backward algorithms can be expressed as linear algebra similarly to the Viterbi algorithm.

The theory, implementation of, and experimentation with the forward algorithm is made by Sand et al. (2013). In this thesis the backward algorithm has been implemented to compute the posterior decoding.

Expressing the backward algorithm in terms of matrix multiplications is essentially the same as the forward algorithm as the two algorithms are very similar as seen in section 3.5. Let the emission matrices, $B_{o,i}$, be identical to the ones described in section 3.3. We can then set the $C$ matrices to

$$C_{o,i} = B_{o,i} A^T,$$
$$C_T = (1, 1, \ldots, 1).$$

(3.8)

Now $\beta_t$ is computed using $C_{y,t}$ and $\beta_{t+1}$ as

$$\beta_t = C_{y,t} \beta_{t-1} = C_{y,t} C_{y_{t+1}} \ldots C_T,$$

(3.9)

which is similar to the forward algorithm in Sand et al. (2013) and the Viterbi algorithm in section 3.3, except that the matrix multiplications no longer are max-times multiplications but normal matrix multiplications.

To compute the posterior decoding equation (3.7) is used. Hence, the matrix formulation is only used to compute the forward and backward tables, while the computation of the posterior decoding itself has not changed from the original formulation.

The asymptotic running time and space consumption is equal to the running time and space consumption of the Viterbi algorithm since the algorithms only differ on the type of matrix multiplication used.

### 3.6.1 Numerical Stability

In this algorithm probabilities are multiplied and as for the Viterbi algorithm numerical issues occur. For the Viterbi algorithm this was solved by working in logarithmic space.

Sand et al. (2013) describes how to make the computations in the forward algorithm numerically stable. First, as the $C$ matrices in equation (3.8) are defined by matrix multiplications they tend to underflow since each matrix consists of entries in the interval $[0, 1]$. To overcome this problem in the forward algorithm the matrices are normalized, that is they are scaled by the sum of the entries in the matrix. This can be reused in the backward algorithm as well, as the $C$ matrices are computed in the same way.
For the computation of the forward table, $\alpha$, the entries in the columns underflow when $T$ becomes large, since the computation is defined by multiplication of probabilities. A similar approach is used to overcome this problem. Each column is normalized by scaling with the sum of the entries. This makes the forward table computation numerically stable. Since the magnitudes of the entries in the forward table and backward table entries are comparable, it is reasonable to use the scalars from the forward table in the computation of the backward table in equation (3.9). Using the same scalars is also required for the result of the posterior decoding to be correct, since the use of different scalars for columns in the forward and backward tables will give arbitrary results.

### 3.7 Problems in Exploiting Repetitions

Sand et al. (2013) exploits repetitions for the forward algorithm using byte-pair encoding similarly to the description in section 3.4. Using the same method it is easy to also exploit repetitions in the backward algorithm.

To compute the posterior decoding of the uncompressed sequence from the forward and backward tables of the compressed sequence, either (1) the posterior decoding can be computed using the partial decoding obtained from the compressed sequence, or (2) the entire forward and backward tables of the uncompressed sequence are required. For both approaches the running time of computing one state in the posterior decoding must take strictly less than $O(N^2)$ time to gain a speedup, since one else might as well compute the forward and backward tables without compressing the input sequence. The two approaches are now discussed in turn.

1. For the Viterbi algorithm it is in section 3.4.3 described how the $R$ matrices can be precomputed to retrieve the full Viterbi path from the partial path. This method makes use of the property that the Viterbi algorithm maximizes over all previous states. From the partial path the previous and next states that maximizes the probability is known. Using this information the current state can be found in constant time.

The problem is that for the forward-backward algorithm the current state does not only depend on the previous and next states since it is not computed using the maximum over all previous states, but the sum of these. Hence, having computed the partial path does not help retrieve the full path as the full path is not dependent on partial path as the Viterbi path is.
2. Computing the full forward and backward tables is another approach. Assume that the tables have been computed for the compressed sequence. The goal is to extend these tables with the missing columns such that the tables for the uncompressed sequence is obtained. Since the full tables have \( N \) entries in each column, the running time of filling an entry must take strictly less than \( O(N) \) time to gain a speedup. A relation might exist between uncomputed columns that corresponds to the same symbol in the input sequence. Looking at figure 3.3, assume the columns \( i \) and \( j \) that correspond to the same symbol has been computed using the compressed sequence. The columns \( i + 1 \) and \( j + 1 \) that also correspond to the same symbol needs to be computed. It is an open question of this thesis whether it is possible to compute those faster than the naive approach that takes \( 2N^2 \) time.

Unfortunately no solution to this problem has been found. Hence, the posterior decoding cannot be sped up by exploiting repetitions in the observation sequence. The running time of computing the posterior decoding is then \( O(MN^3 + TN^2) \) which is asymptotically worse than the original posterior decoding algorithm that runs in \( O(TN^2) \). However, note that the alphabet size \( M \) typically is small, so it is expected that the asymptotic running times of the two algorithms in practice are comparable.
3.8. **Indexed Posterior Decoding**

In the last section it was discussed that the posterior decoding for an observation sequence cannot be computed asymptotically faster using the byte-pair compression. However, if only a part of the posterior decoding is needed instead of the entire decoding, a theoretical speedup can be obtained.

For indexed posterior decoding not only an input sequence $Y_{1:T}$ and a model $\lambda$ is given as input, but also two indexes $i, j \in [1, T]$. Instead of returning the posterior decoding $P_{1:T}$ only the part of the sequence from index $i$ to $j$ denoted $P_{i:j}$ is returned.

Recall from section 3.4.2 that the columns of the forward and backward tables, $\alpha'$ and $\beta'$, of the compressed sequence $Y'_{1:T'}$ are a subset of columns in the tables, $\alpha$ and $\beta$, of $Y_{1:T}$. Furthermore, recall that a column is computed using the column immediately to the left or right of the forward and backward algorithms respectively as seen in equation (3.5) and (3.6). Using this, the columns $i$ to $j$ of the forward and backward tables, $\alpha$ and $\beta$, corresponding to the subsequence $Y_{i:j}$ can be found using existing columns from the forward and backward tables, $\alpha'$ and $\beta'$.

An example of the above is illustrated in figure 3.4. Assume that the forward table, $\alpha'$, (and the backward table, $\beta'$) for the compressed sequence has been computed. This is shown as gray columns. To compute $\alpha$ and $\beta$ from index $i$ to $j$, the computed column at index $k$ with $k \leq i$ can be used as a starting point for the forward computation. In this example that is index 3.
that corresponds to the column at index 2 in $\alpha'$. Likewise, column $l$ with $l \geq j$ can be used as a starting point for the computation of the backward table.

**Starting Point Columns**

To find the indices $k$ and $l$ for the uncompressed sequence and the corresponding indices $k'$ and $l'$ for the compressed sequence, a mapping $M_{ac}$ from uncompressed indices to compressed indices is created. In this example the map would consist of the mappings $1 \rightarrow 1, 3 \rightarrow 2, 7 \rightarrow 3$ and $10 \rightarrow 4$. To find the index $k$ a search is performed on the map to find the largest key less than or equal to $i$. The value corresponding to $k$ is $k'$. Likewise, $l$ and $l'$ can be found by finding the smallest key larger than or equal to $j$.

To compute this mapping, another mapping $M_{sl}$ from each symbol in the new alphabet $O'$ to its original length is created. This can be computed iteratively by first inserting each original symbol $o_1, \ldots, o_M \in O$ with length 1 into the map. Next, $o_{M+1}, o_{M+2}, \ldots, o_{M'} \in O'$ is inserted in the specified order. As each new symbol $o_c \in O'$ is created using two existing symbols $o_a$ and $o_b$. The length of $o_c$ can be computed using the map to obtain the lengths of $o_a$ and $o_b$ and add these two together.

Now $M_{ac}$ is computed by scanning through the compressed sequence one symbol at a time while incrementing the corresponding index in the uncompressed sequence using the map of lengths.

The starting columns for the forward and backward algorithms have now been found. The only thing missing for the computation is the sequence for which to compute the following (for forward) or preceding (for backward) columns.

**Sequence**

The simplest solution for finding a sequence for which to compute the forward and backward tables is to store the original sequence $Y_{1:T}$ and then extract the sequence $Y_{k:l}$ from that. However, $Y_{1:T}$ can be very large in some applications. Instead of storing $Y_{1:T}$, $Y_{k:l}$ can be found by decompressing $Y'_{k':l'}$, by replacing the symbols $o_{M+1}, \ldots, o_{M'}$ by symbols from the original alphabet $O$. An example of this algorithm is shown in algorithm 1. The $\alpha$ and $\beta$ tables from $i$ to $j$ can be found by computing the columns corresponding to $Y_{k:l}$ and afterwards remove columns $[k, i)$ and $(l, n]$.

Decompressing $Y'_{k':l'}$ entirely might be inefficient though. If the compressed sequence $Y'_{1:T'}$ is highly compressed, i.e. $T' \ll T$, the length of $Y_{k:l}$ might be much larger than $Y_{i:j}$. In the worst case of $T' = 1$, the decompression would
Algorithm 1 Simple decompression algorithm.

1: procedure INDEXED_DECOMPRESSION($Y', k', l'$)
2:     $Z \leftarrow Y'_{k':l'}$
3:     for $c = M'$ down to $M + 1$ do
4:         $o_l, o_r \leftarrow \text{get_pair}(o_c)$
5:         Replace $o_c$ by $o_l o_r$ in $Z$
6:     end for
7:     return $Z$
8: end procedure

result in the uncompressed sequence $Y_{i:T}$. Hence, the indexed posterior decoding algorithm will spend a lot of time computing columns that are thrown away afterwards, and the running time is $O(M'N^3 + N^2T)$ which is not an improvement over the original posterior decoding. The decompression needs to be more clever to improve the running time.

Observe that only the symbols from index $[i, j]$ need to be from the original alphabet $O$. The symbols from $[k, i)$ and $(j, l]$ can be from the new alphabet, since each column in these parts of $\alpha$ and $\beta$ are not needed for the posterior decoding computation from $i$ to $j$. Hence, when decompressing $Y'_{k':l'}$ symbols that correspond to uncompressed symbols in the interval $[k, i)$ should not be decompressed, but left as compressed symbols. Likewise, symbols corresponding to the interval $(j, l]$ should not be decompressed. As before, symbols corresponding to uncompressed symbols that overlap $[i, j]$ should be decompressed. Is this case the partial decompression of $Y'_{k':l'}$, resulting in a new sequence $Z$ will then contain symbols from $O'$ at the beginning, then symbols from $O$ in the middle being the symbols $Y_{i:j}$, and then symbols from $O'$ again in the end.

An algorithm for doing this takes as input a compressed sequence $Y'$ and the indices $i, j, k, l, k'$ and $l'$. The symbols of subsequence $Y'_{k':l'}$ is pushed to a stack $Y^s$ such that $Y'_{l'}$ is at the bottom and $Y'_{k'}$ at the top. The algorithm pops symbols one by one from $Y^s$, while the corresponding index into the uncompressed sequence $Y$ is maintained using $M_{sl}$. When a symbol $c$ that overlap $[i, j]$ is encountered $c$ is either appended to $Z$ if $c \in O$ or it is split into a pair and pushed to of $Y^s$ such that the left symbol of that pair will be popped in the next iteration. The first time a symbol $c \in O$ overlaps $[i, j]$ is encountered the index (named start_index) of that symbol in $Z$ is saved, as it is used later for computing the posterior decoding. If $c$ overlaps $[k, l]$ it is appended to $Z$. In case there is no overlap between $c$ and $[k, l]$ nothing is
computed, i.e. $c$ is thrown away. Pseudo code for this algorithm is given in algorithm 2. This algorithm is much more efficient than algorithm 1 in terms of running time.

Algorithm 2 Partially decompress the compressed sequence.

```
1: procedure indexed_partial_decompression($Y', i, j, k, l, k', l'$)
2:     $Z \leftarrow \text{nil}$
3:     Stack $Y^S \leftarrow \text{nil}$
4:     Push $Y'_k, \ldots, Y'_l$ to $Y^S$
5:     start_index $\leftarrow \text{nil}$
6:     index $\leftarrow k - (M_{sl}(Y^S.\text{top}) - 1)$
7:     while $Y^S$ is not empty do
8:         $c \leftarrow Y^S.\text{pop}()$
9:         next_index $\leftarrow$ index + $M_{sl}(c)$
10:        if (index $\leq i$ $\land$ next_index $> i$) $\lor$ (index $> i$ $\land$ index $\leq j$) then
11:            if $c \in \mathcal{O}$ then
12:                $Z.\text{append}(c)$
13:                if start_index is nil then
14:                    start_index $\leftarrow Z.\text{size}() - 1$
15:                end if
16:            end if
17:        else
18:            $c_l, c_r \leftarrow \text{get_pair}(c)$
19:            Push $c_r, c_l$ to $Y^S$
20:            continue $\quad$  /* Don’t update index */
21:        end if
22:    else if (index $\leq k$ $\land$ next_index $> k$) $\lor$ (index $> k$ $\land$ index $\leq l$) then
23:        $Z.\text{append}(c)$ $\quad$  /* $c$ overlaps $[k, l]$ */
24:    end if
25:  end while
26: return $Z, \text{start_index}$
27: end procedure
```

In the previous paragraphs the start columns from $\alpha'$ and $\beta'$, that is the columns at index $k'$ and $l'$ respectively, have been found. They are used as the first and last column of the forward table and the backward table for the partially decompressed sequence $Z$. Furthermore, the sequence $Z$ has been found. The forward and backward tables of that sequence can now be computed using the equations (3.5) and (3.6), but omitting the base cases as the
first (or last) column of the tables has already been computed. As the partially decompressed sequence \( Z \) corresponds to the interval \([k, l]\), but only the posterior decoding for the interval \([i, j]\) is requested, the posterior decoding should only be computed for that interval. This is where `start_index` is used as it is the index into \( Z \) that corresponds to the index \( i \) in \( Y_{i:j} \). Equation (3.7) is still used to compute the posterior decoding, but the computation starts at \( t = \text{start\_index} \) and ends at \( t = \text{start\_index} + j - i \).

### 3.8.1 Running Time

The analysis of the indexed posterior decoding is split into a number of steps.

1. Compute \( \alpha' \) and \( \beta' \);
2. compute the start columns;
3. compute the sequence \( Z \) corresponding to the columns;
4. compute the forward and backward tables and the posterior decoding for \( Z \).

These four steps are now analyzed in turn.

1. The running time of the forward and backward algorithms is the same as for the Viterbi algorithm. If preprocessing/compression of the input string \( Y_{1:T} \) is not included it takes time \( O(M'N^3 + N^2T') \) to compute \( \alpha' \) and \( \beta' \).

2. Computing the starting point columns requires building the map \( M_{sl} \). This takes time proportional to the number of symbols, i.e. \( O(M') \). Computing the map \( M_{uc} \) requires a scan through the compressed sequence. That takes \( O(T') \) time. Performing the search on the indices of \( M_{uc} \) to find the indices \( k, l, k' \) and \( l' \) takes \( O(T') \) time. In total this becomes \( O(M' + T') \).

3. The analysis of algorithm 2 is more complex. In the worst case \( Y'_{1:T'} \) has length 1, i.e. \( T' = 1 \). The compression of \( Y'_{1:T'} \) is illustrated as a binary tree seen in figure 3.5. This is also seen in figure 3.1 where \( T' = 5 \) and the compression corresponds to five binary trees. In this example the compression is of the observation sequence \( Y_{1:16} = o_1o_1\ldots o_1 \) and corresponds to a perfectly balanced tree. Assume that the indexed posterior decoding is requested for index \( i = 6 \) to \( j = 14 \). Since \( T' = 1 \), \( k' = 1 \) and \( l' = 1 \) algorithm 2 will begin at the root of the tree and
decompress it into two symbols, namely the children of the root. This
decompression continues for all nodes that are necessary to decompress
to obtain $Y_{i:j}$. To make the analysis easier to reason about some nodes
in the illustration has been colored. Nodes colored in red or blue
correspond to symbols that are introduced by decompression of $Y'_{1:T}$.
Nodes without coloring correspond to symbols that are not introduced
by the decompression. The running time of the algorithm is proportional
to the number of colored nodes, so to find the running time the number
of nodes need to be counted.

The colored nodes are split into two. The blue nodes correspond to
symbols that are created during the decompression and has descendants
that are not decompressed. At each level in the tree there are at most two
blue nodes. Hence, there is at most 2 times the height $h$ of the tree blue
nodes. The red notes correspond to symbols that are fully decompressed
into symbols from the original alphabet $O$. To count the number of
these observe that the $j - i$ leaves have at most $\frac{j-i+2}{2}$ parents, at most
$\frac{j-i+2}{4}$ grand parents and so on. In this example the leaves and their red
ancestors correspond to three subtrees in the tree, but in the worst case
they correspond to one balanced subtree. The size of a balanced binary
tree with $j - i + 2$ leaves is $2(j - i + 2) - 1$, which is the worst case number
of red nodes. Summing the number of blue and red nodes we obtain

$$2h + 2(j - i + 2) - 1 = O(h + (j - i)).$$

The height of tree, $h$, can be bounded. The worst case height is obtained
for sequences that compresses well, that is sequences with many repM
etitions. For unary sequences the tree has height $h = \log_2 T$ as seen
in figure 3.5. Another class of words that contain many repetitions
is Fibonacci words. They are generated in a way very similar to how
Fibonacci numbers are generated by concatenating the two previous
words, corresponding to generating the next Fibonacci number from
the two previous ones. Let $S_0$ be “0” and $S_1$ be “01”. Now define
$S_n = S_{n-1}S_{n-2}$. According to Fraenkel and Simpson (1998) it is not
known whether a class of words with more repetitions exist, so for this
analysis it is assumed that Fibonacci words are the worst case. Observe
that the number of leaves $T$ for a tree of height $h$ is the $h$’th Fibonacci
number. In general, the $n$’th Fibonacci number can be computed as

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2}\right)^n$$
rounded to the closest integer value. Using this, the height $h$ can be bounded in terms of $T$ using that

$$T = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{h}$$

$$\Rightarrow \log T = \log \left( \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{h} \right)$$

$$= \log \frac{1}{\sqrt{5}} + \log \left( \frac{1 + \sqrt{5}}{2} \right)^{h}$$

$$= \log \frac{1}{\sqrt{5}} + h \log \frac{1 + \sqrt{5}}{2}$$

$$\Rightarrow h = \frac{\log T - \log \frac{1}{\sqrt{5}}}{\log \frac{1 + \sqrt{5}}{2}} = O \left( \log T \right).$$

Hence, the number of colored nodes is no larger than $O(\log T + (j - i))$, which is also the worst case running time of algorithm 2.

4. Finally the posterior decoding is computed. This takes time proportional to the length of the decompressed subsequence. Of course, the subsequence contains uncompressed symbols from index $i$ to $j$, but as previously discussed the subsequence also contains symbols corresponding to indices $[k, i)$ and $(j, l]$ that may or may not have been decompressed. These are drawn as nodes with a thick line in figure 3.5. The number of these symbols can also be bounded in terms of $T$. Since the algorithm does not decompress symbols that corresponds to decompressed symbols in the interval $[k, i)$ and $(j, l]$, there is at most be two nodes corresponding to these intervals at each layer in the tree. Hence, the number of nodes in the intervals is bounded by the height of the tree that is $\log T$. The subsequence then has length $O(j - i + \log T)$.

Computing the forward and backward tables take time $O(N^2(j - i + \log T))$, and computing the posterior decoding using equation (3.7) also takes $O(N^2(j - i + \log T))$ time.

By adding the running times of these four steps a running time of

$$O \left( M'N^3 + N^2T' \right) + O \left( M' + T' \right) + O \left( \log T + (j - i) \right) + O \left( N^2(j - i + \log T) \right)$$

$$= O \left( M'N^3 + N^2T' + N^2(j - i + \log T) \right)$$
is obtained for the indexed posterior decoding algorithm. Note that this is a worst case running time. For many practical applications the compression will not be as good and thus $T' > 1$. In that case the decompression can be illustrated as $T'$ trees. In many cases only some of these need to be traversed and the height of them will be smaller than $\log T$. Furthermore, the $\log T$ term will most likely not dominate the $j - i$ term. As an example the human genome has length approximately $3,000,000,000$ base pairs and can be used as the observation sequence in some bioinformatics applications. As $\log_2 3,000,000,000 \approx 31$ and $j - i$ probably will be larger, the $\log T$ term does not contribute much of the running time in most cases.
Chapter 4

Implementation

This chapter describes the important aspects of the implementation. As this thesis is intended to be an extension of zipHMMlib by Sand et al. (2013), the code itself is also an extension of the library code. By the work made in this thesis the library has been extended to include an efficient Viterbi algorithm and an indexed posterior decoding algorithm exploiting repetitions, as well as a posterior decoding algorithm that does not exploit repetitions.

As zipHMMlib is written in C++, the additions made in this thesis have also been written in C++. The code is available at http://users-cs.au.dk/muldvang/thesis. The build procedure has not changed, so for instructions on how to build the project see http://github.com/mailund/ziphmm.

4.1 Description of the Implementation

In the original library the main class of the library was the Forwarder class. This class has been renamed to HMMSuite. This class is responsible for preprocessing sequences, reading and writing these from and to the disk, and for executing the forward, backward, Viterbi and posterior decoding algorithms.

Each sequence is represented as a std::vector<unsigned> in memory. Using unsigned to represent symbols makes the compression easy as, a sequence of alphabet size four consists of the symbols 0, 1, 2, and 3 and during the compression the symbols 4, 5, 6, etc. are added. The matrices of the model, Π, A, and B, and the matrices computed in the algorithms are handled by the Matrix class. In this class a matrix is represented as a single double array, where the first elements of the array is the first row of the matrix, the next elements is the second row of the matrix, etc. The Matrix class is also responsible for various operations on matrices such as converting
it into log-space or multiplying matrices. For matrix multiplications a BLAS framework is used to make these as efficient as possible. For the max-times matrix multiplication and addition simple nested for loops are used as BLAS does not support these operations. For the algorithms where a table is stored it is stored as a Matrix array.

As mentioned, the library supports reading in multiple sequences from a folder. In the original library the sum of the probabilities for each sequence is returned by the forward algorithm. It is not currently supported to read in multiple sequences if either the forward or backward tables are requested or if the Viterbi path or posterior decoding is requested.

### 4.2 Usage

In the following three sections the usage of the Viterbi algorithm, the posterior decoding algorithm, and the indexed posterior decoding algorithm from within C++ is described. Lastly the programs added to the library are discussed.

#### 4.2.1 Viterbi

In listing 1 an example of C++ program that uses zipHMMlib to compute the Viterbi decoding can be found.

The `HMMSuite.read_seq(...)` reads a sequence from a file. The user has to specify the alphabet size since the sequence may not contain all symbols from the alphabet. Furthermore, the user specifies the number of states in the model $N_{\text{min}}$ that will be used later and the number of executions $e$ of the Viterbi algorithm as discussed in section 3.4.7.

If the compressed sequence will be reused later by another program, it is saved to the disk by `HMMSuite.write_to_directory(...)`. It can then be loaded using `HMMSuite.read_from_directory(...)`. `HMMSuite.read_HMM(...)` simply reads an HMM specification from a file. `HMMSuite.viterbi(...)` runs the Viterbi algorithm. A boolean `memory_save` can be specified to choose whether to save memory or not as discussed in section 3.2. If the parameter is not specified, the algorithm defaults to use memory saving. If the parameter `viterbi_path` is provided, the algorithm will backtrack to obtain the Viterbi path. If it is omitted only the probability of the most likely path is computed.
Listing 1 Compute the Viterbi path using zipHMMlib.

```cpp
#include "hmm_io.hpp"
#include "hmm_suite.hpp"

int main() {
    // Read input.
    zipHMM::HMMSuite h;
    size_t alphabet_size = 2;
    size_t min_num_of_evals = 1;
    size_t no_states = 2;
    h.read_seq("test.seq", alphabet_size, no_states, min_num_of_evals);

    // Save for future runs.
    h.write_to_directory("some_dir");

    // Read HMM.
    zipHMM::Matrix pi, A, B;
    zipHMM::read_HMM(pi, A, B, "test.hmm");

    // Run Viterbi.
    bool memory_save = false;
    std::vector<unsigned> viterbi_path;
    double probability = h.viterbi(pi, A, B, memory_save, viterbi_path);

    // Print result to stdout.
    std::cout << "Probability: " << probability << std::endl
              << "Path: ";
    for (std::vector<unsigned>::const_iterator it = viterbi_path.begin();
         it != viterbi_path.end(); ++it)
        std::cout << *it << " ";
    std::cout << std::endl;
    exit(0);
}
```

4.2.2 Posterior Decoding

As described in section 3.7, exploiting repetitions cannot speed up the execution of the posterior decoding algorithm. It might still be faster, due to matrix multiplications being faster than the original approach. The implementation finds a posterior decoding of the same length as the compressed sequence. Hence, to find the posterior decoding of the original observation sequence \( Y_{1:T} \), compression should be disabled. This can be achieved by specifying \( e = 0 \) when reading the sequence using `HMMSuite.read_seq(...)`. The `HMMSuite.posterior_decoding(...)` is a bit simpler than the Viterbi
method and takes as parameters the matrices $\pi, A$, and $B$ and a vector in which the posterior decoding is saved.

### 4.2.3 Indexed Posterior Decoding

Compression can be enabled for the indexed posterior decoding. The `HMMSuite::indexed_posterior_decoding(...)` takes as parameters the matrices $\pi, A$, and $B$, the indices $i$ and $j$, and a reference to a vector where the indexed posterior decoding is saved.
Chapter 5

Experiments

This chapter covers description and discussion of all experiments performed in this thesis. First, the data used is described followed by the experimental setup. Finally, the Viterbi and posterior algorithms are discussed in turn.

5.1 Data

Hidden Markov models have successfully been applied to various kinds of data. As this thesis is written with no specific application in mind, aside from repetitive data of course, the experiments use random data that does not contain many repetitions and Fibonacci words that is a highly repetitive kind of sequence. The two types of sequences used is now described in turn.

Random sequences are generated by uniformly choosing symbols from the alphabet. The sequences vary in length from $10^3$ to approximately $10^8$.

Fibonacci words have already been described in section 3.8.1 but recall that they are generated by concatenating the two previous words. Let $S_0$ be “0” and $S_1$ be “01”. Now define $S_n = S_{n-1}S_{n-2}$. In contrast to the random sequences, Fibonacci words contain many repetitions due to the recursive definition and it is expected that they compress well.

The models used are random, fully connected models, i.e. there is a transition from a state to any other state. The emission probabilities are random too. The alphabet size has been kept constant at four. In general, the larger the alphabet, the worse compression ratio is expected. The size of the models, i.e. the number of states, varies from 2 to 512.
5.2 Experimental Setup

The experiments have been run on a PC with an Intel Xeon W3550 3.07 GHz CPU and 4 GB RAM running GNU/Linux 3.13. As zipHMMlib uses the ATLAS as BLAS implementation as default, this has also been used for the experiment. ATLAS version 3.10.2 was used. GCC 4.8.2 with optimization flag O3 was used for compiling the code.

Each point in the plots in this chapter corresponds to the mean of multiple runs. For all experiments, the executions of a program with a set of parameters has been repeated five times and the mean computed to even out fluctuations in the running time. Furthermore, in the case of random sequences, 10 random sequences of the same length has been generated and the mean of the measure calculated since, especially for short sequences, the compression ratio may vary a lot. The standard deviation has been calculated and is shown as error bars in the plots to visually indicate the fluctuations in running time.

The running time in all experiments has been measured using time.h that provides a time resolution of nanoseconds for measuring real time.

Many of the figures in this chapter verify the theoretical running of the algorithms. For verifying running time one of the factors of the running time has been varied while the remaining has been kept constant. The actual running time of the algorithm is divided by varied factor. If the points in the plot then are constant, the running time satisfies the theoretical bound. However, since some of the algorithms read in data etc. before the computation starts, the points will typically form a decreasing curve instead of being constant.

5.3 Compression Ratio

The running time of the Viterbi algorithm and the indexed posterior decoding algorithm implemented in zipHMMlib is highly dependent on the size of the compressed sequence. As mentioned earlier 3.4.7 it is possible to compress the sequence to a single character, even though this might not be a good idea in terms of performance. To see how well the different kinds of data compresses when used by zipHMMlib the compression ratio for all the sequences described above have been measured. The compression ratio is defined as \( \frac{\text{original size}}{\text{compressed size}} \).

Recall that the amount of compression depends on the estimate of how many times the algorithm is run after compressing the sequence. For this experiment estimates of \( e = 1 \) and \( e = 500 \) have been provided to the program.

As seen in figure 5.1 the compression ratio grows linearly for Fibonacci words as the sequence length increases, whereas it is almost constant for
random sequences. The compression of Fibonacci words is much better than the compression of random sequences as expected. This suggests that the performance of the Viterbi algorithm will be very dependent on the input sequences. To verify that Fibonacci words are very repetitive, unary sequences have also been compressed, and as seen the compression of these two types is very comparable.

It is also seen that the compression is dependent of the \( e \) parameter. For Fibonacci words the compression ratio is approximately a factor of 10 lower when \( e = 1 \) compared to \( e = 500 \). For random sequences of alphabet size four only a single iteration of the compression is run when \( e = 1 \) and the sequence is only compressed by a small percentage. For \( e = 500 \) more iterations of the compression is made and thereby a better compression ratio is obtained.

For this experiment only, random sequences of an alphabet of size 2 were also tested. It is expected that the random sequences with an alphabet of size 2
will compress better than the sequence with an alphabet size 4, since the most common pair of symbols will occur more often. This is also seen in the plot for $e = 500$. For random sequences of alphabet size four the compression ratio is approximately four, while it is almost eight for sequences with alphabet size 2. For $e = 1$ more than one iteration of the compression is made for random sequences of length 2 and the compression becomes better than for random sequences of length 4. This suggests that using the library with e.g. protein sequences that has an alphabet size of 20 might not result in a good performance when zipHMMlib is used.

To keep the experiments simple and limit the number of plots, an alphabet size of four has been used for the rest of the experiments. This corresponds to DNA sequences, which HMMs are often used with in the field of bioinformatics.

### 5.4 Preprocessing

As described in section 3.4.5 the compression of the input sequence is computed prior to executing the Viterbi or forward-backward algorithms and can be saved to disk. This is useful if the algorithms are used with different models, as the compression of the sequence does not depend on the model.

The preprocessing has a theoretical running time of $O((M' - M)T)$ as stated in section 3.4.4. In figure 5.2 the running time divided by the length of the sequence, $T$, is shown for random sequences and Fibonacci words of varying length.

Recall that the preprocessing iterates over the observation sequence and finds the most common pair of symbols, introduces a new symbol, and replaces the pair with the new symbol.

As seen the running time is not linear in $T$ for random sequences. To explain this, assume that the sequence is compressed using $M' - M$ iterations with a new symbol introduced in each iteration. As seen in the previous section the observation sequence only becomes a bit, say $\epsilon_i$, smaller in iteration $i$. Thus, an approximate running time of the compression of random data can be written as

$$O(T + T - \epsilon_1 + \epsilon_1 + \epsilon_2) + \cdots + T - (\epsilon_1 + \epsilon_2 + \cdots + \epsilon_{M' - M - 1})) = O ((M' - M)T).$$
Since the compression ratio is dependent on $T$ as seen in the previous section and the size of the new alphabet $M'$ grows with the compression ratio, $M'$ is also dependent on $T$. Hence, when $T$ grows $M'$ also grows i.e. they are not independent. What is seen in the figure is not surprising; the running time of the preprocessing for random sequences is superlinear in $T$ and not linear as the theoretical running time suggests.

However, for Fibonacci words the running time is linear in $T$. The new alphabet size $M'$ is linearly dependent on $T$ as seen in the previous section. Due to the recursive structure of Fibonacci words the length of the observation sequence is almost halved each time a new symbol is introduced. The running time can be written as

$$O(T + \frac{T}{2} + \frac{T}{4} + \frac{T}{8} + \cdots + \frac{T}{2^{M'-M}}) = O(T).$$

Hence, the first iteration of the preprocessing takes time proportional to $T$, the next to $T/2$, the next to $T/4$, etc. Thus, even though $O'$ is dependent on $T$ the recursive structure makes the running time linear in $T$ and independent of $O'$ and $O$.

5.5 Viterbi

This section covers the experiments made in context of the Viterbi algorithm. First, the theoretical asymptotic running time of the algorithm is verified
to make sure that the running time of the algorithm behaves as expected. Secondly, an implementation of the original Viterbi algorithm is compared to the zipHMMlib implementation.

5.5.1 Verification of Theoretical Running Times

In this section the theoretical running times are verified. The running time has been measured for three variations of the Viterbi algorithm corresponding to the backtracking discussed in section 3.4.3. They are named as follows.

- **Viterbi** compresses the sequence and only computes the probability,
- **Viterbi** also backtracks,
- **Viterbi** also saves memory on backtracking.

The next two sections verifies that the implementation follows the theoretical running times.

**Sequence Length**

The implementation of the Viterbi algorithm supports both computing and not computing the Viterbi path. There is a difference in the theoretical running time since computing the path is linear in $T$. For a list of running times for the algorithms see table 3.1.

In figure 5.3 the running time divided by the length of the compressed sequence $T'$ is shown for input sequences of varying lengths. The experiment has been made for both random sequences (5.3a) and Fibonacci words (5.3b). As expected the fraction is close to constant with a slightly decreasing curve for Viterbi in both cases. For random sequences the running time for Viterbi and Viterbi is in practice linear in $T'$. This is due to the compression ratio being only approximately four, and that the backtracking is computed quickly using the $R$ matrices. For Fibonacci words the compression ratio is much better and the Viterbi and Viterbi algorithms are clearly not linear in $T'$.

Both backtracking methods have a theoretical running time linear in the length of the original sequence $T$. To verify this the running time is divided by $T$ for sequences of varying lengths and shown in figure 5.4. As seen, it is indeed linear in $T$.

**Model Size**

The theoretical running times of all three variations of the Viterbi algorithm is cubic in the number of states $N$. To verify this an experiment with random
5.5. \textit{VITERBI}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{The running time of Viterbi algorithms divided by the compressed sequence length $T'$. Viterbi$_L$ is linear in $T'$ whereas Viterbi$_P$ and Viterbi$_{PM}$ are superlinear.}
\end{figure}

(a) For random data the running time is in practice linear in $T'$ as the compression ratio is approximately four.

(b) For the very repetitive Fibonacci words, $T' = 2$. This exploits that the backtracking algorithms are far from linear in $T'$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{The running time of Viterbi as a function of the original sequence length $T$. All three variations are linear in $T$.
\end{figure}

(a) Random sequences.

(b) Fibonacci words.
sequences of length 10,000 has been run for models with between 2 and 512 states. Fibonacci words have not been used in this experiment since the running time does not change asymptotically with the compression ratio. The running time divided by $N^3$ is shown in figure 5.5.

It is expected that the points in the plot will form a decreasing curve. This is however not the case. The running time seems to be constant until a model size $N = 16$. After that the running time divided by $N^3$ increases until $N = 256$ where it again stabilizes. It is expected that this is due to memory alignment of matrices in the implementation. Recall that for Viterbi the max-times matrix multiplication is implemented naively using for-loops. As each cell in the resulting matrix is computed using a row from one matrix and a column from the other cache misses are likely to occur since the memory alignment of the two matrices is the same as described in section 4.1.

To check this an experiment measuring the running time and the percentage of L1, L2, and L3 cache misses have been conducted. The cache misses have been measured using PAPI. The result of this is shown in figure 5.6. Note that the running time is not identical to figure 5.5 as this experiment has been made with a shorter observation sequence. As seen the percentage of L1 cache misses increase from zero to approximately 50% when the running time starts to increase. When cache misses also occur for the L2 cache the running time becomes even worse. By this experiment it is concluded that the alignment of the data in the matrices is the cause of the running time not to behave as expected. It is concluded that the running time is indeed proportional to $N^3$ as it becomes constant in figure 5.6 for large models. The number of cache misses can be reduced by changing the implementation of the max-times matrix multiplication or by keeping two copies of each matrix in memory, each with a different memory layout, and use the appropriate one for multiplications.

Recall that the running time is cubic in $N$ due to the computation of the $C$ matrices. The multiplication of these matrices, i.e. the computation of the Viterbi or forward-backward algorithm, takes time proportional to $N^2$ as stated in section 3.4.4. When the size of the new alphabet $O'$ is small compared to the length of the compressed sequence, the $N^3$ term vanishes in practice when compared to the $N^2$ term. Hence, it is expected that the running time of the matrix multiplication (and the backtracking) is proportional to $N^2$ by not compressing the sequence. This is shown in figure 5.7 where $|O'| = 4$ and $T = 10,000$.

This ends the experiments concerning the theoretical running times of the Viterbi algorithm itself. It has been shown that the running time in practice in terms of the length of the observation sequence is very dependent on the structure of the sequence, but it still fits the theoretical worst case running time.
5.5. VITERBI

Figure 5.5: The running time of Viterbi as a function of the number of hidden states in the model $N$. 

Figure 5.6: Cache misses and running time of Viterbi$_{PM}$ for varying number of hidden states $N$. The percentage of cache misses has been calculated as the number of cache misses divided by the number of cache accesses.
Figure 5.7: The running time of Viterbi as a function of the number of hidden states, \( N \), in the model. If the input sequences are not compressed the running time of all three variations of the Viterbi algorithm is quadratic in \( N \).

In terms of the model size the physical hardware becomes visible. However, when the models become large enough the running time stabilizes and fits the theoretical running time.

5.5.2 Comparing zipHMMlib Viterbi to the Original Viterbi Algorithm

The zipHMMlib Viterbi algorithm is compared to an implementation of the original Viterbi algorithm. Recall that the compressed sequence from the preprocessing can be saved to disk for later use. However, for many problems one only need to run Viterbi once for a sequence, where the most likely path of hidden states is requested for an observation sequence and a model. Whereas for e.g. Viterbi training, the sequences is used multiple times. As described in section 3.4.7 the preprocessing uses an estimate \( e \) of how many times the Viterbi algorithm is run to compress the sequence appropriately. To make a fair comparison between the two algorithms the preprocessing has been included in the running time for these experiments. To experiment with this, experiments have been made with \( e = 1 \) and \( e = 500 \) executions of Viterbi. In the end of this section is it seen that \( e = 500 \) roughly corresponds to measure
the running time without preprocessing as the preprocessing time is a minor fraction of the total running time.

The algorithms are compared to their respective counterpart of the original algorithm, e.g. for Viterbi, the original algorithm without backtracking has been used. In the following plots a speedup factor is shown on the y-axis. The speedup factor is calculated as the running time of the original algorithm (with or without backtracking enabled) divided by the running time of algorithm being measured, including the time it takes to do the preprocessing. Hence, any speedup factor larger than one means that the algorithm being measured is faster than the original algorithm.

Random Data

The first experiments have been made using random sequences.

In figures 5.8a and 5.8b the speedup factor for the Viterbi algorithm is shown for sequences of varying length with $e = 1$ and $e = 500$, respectively. A model with 16 states was used for these experiments.

As expected, computing the Viterbi path limits the speedup factor since the running time is linear in $T$ instead of $T'$ as it is for only computing the probability. Viterbi$_P$ and Viterbi$_{PM}$ gain a similar speedup. This is expected as the main part of the memory saving in Viterbi$_{PM}$ essentially consists in computing Viterbi twice. Since this is computed for both the zipHMMlib implementation and the original implementation, the speedup becomes the same as for Viterbi$_P$. For $e = 500$ the speedup is better than for $e = 1$ due to the compression ratio. For small sequences however, the speedup factor is approximately equal for $e = 1$ and $e = 500$ for Viterbi$_P$ and Viterbi$_{PM}$. This is due to the zipHMMlib Viterbi algorithm computing the $C_{o_{ij}}$ matrices before starting the actual Viterbi computation. As the sequences become longer this computation becomes a smaller fraction of the running time and thereby the running time increases at the beginning for $e = 500$.

For $e = 1$ the new alphabet size $O'$ is very small and the $C_{o_{ij}}$ matrices are quickly computed. Hence, the running time is almost independent of this computation and the speedup becomes close to constant for increasing sequence lengths. For $e = 1$ the speedup factor is very large for computing only the likelihood, but it decreases quickly as the sequence length is increased. This is partly due to the compression being better for small sequences as seen in section 5.3, but is does not explain that the speedup factor is so large. However, the running time also varies a lot as seen from the error bars so to find any further explanation more experiments should be made.
Instead of presenting the speedup factor as a function of the sequence length, the model size is used in the next experiment. While the size of the model is varied the length of the observation sequence is kept constant at length 10,000. Recall that the running time of Viterbi is \( O(M'N^3 + N^2T') \). The cubic term might become a problem as \( N \) grows. The speedup factor as function of model size \( N \) is plotted in figures 5.9a and 5.9b for \( e = 1 \) and \( e = 500 \), respectively.

For \( e = 1 \) the speedup factor is growing in the beginning as the number of states increases. This is due to the matrix representation of the algorithm making it more efficient. However, for larger matrices, the cubed \( N \) term becomes a limiting factor and the speedup factor drops below 1. This is also seen for \( e = 500 \). For \( e = 500 \) however, the best result is obtained for very small models. This is due to the new alphabet size \( O' \) being bigger when \( e = 500 \). Hence, the \( M'N^3 \) term of the running time becomes more dominant and the speedup becomes smaller even for small increases of the model size.

**Fibonacci Words**

Speedups is gained for many cases using random data, but it is expected that much greater speedups will be obtained using repetitive sequences. Thus, experiments similar to the ones for random data have been conducted. In figures 5.10a and 5.10b the speedup factor is shown for sequences of varying
Figure 5.9: The speedup factor of the Viterbi algorithms for varying model sizes.

As seen, the speedup factors are greater than for random data. It is however very dependent on the value of $e$. For $e = 1$, even though the sequence is being compressed a lot as seen in section 5.3, the speedup factor is limited
since a lot of time is still spent on compression. For $e = 500$ however, the speedup factor keeps increasing for rather long sequences for the Viterbi$_L$ algorithm, but as for random data the speedup converges to a constant for long sequences. Below it is shown that this is due to the algorithm using more time on compression. For random data the computation of the Viterbi path limited the speedup factor. This limitation becomes even larger for Fibonacci words. This is seen in figure 5.10b, where the computation of the Viterbi path makes the computation orders of magnitudes slower than just computing the probability. However, the speedup factor for the Viterbi$_P$ and Viterbi$_{PM}$ algorithms perform better on Fibonacci words than on random sequences.

**Future Work**

To get an impression of where the bottleneck of the algorithm is, the running time of the preprocessing and the execution of the Viterbi algorithm has been measured. To keep the number of graphs in this chapter at a reasonable level, the experiment has only been made for the Viterbi$_L$ and Viterbi$_{PM}$ algorithms. Similar results can be obtained for the Viterbi$_P$ algorithm. Again, for $e = 1$ the preprocessing has been computed and then Viterbi has been executed once. For $e = 500$ the preprocessing has been computed followed by 500 executions of Viterbi. The result is seen in figure 5.11.

For the $e = 1$ experiments it is seen that a large part of the running time is spent on the preprocessing. However, as the number of executions is increased the time of the preprocessing becomes lower for random data, while it still accounts for the largest part of the running time for Fibonacci words. For $e = 500$ the time spent on preprocessing is very low compared to the actual computation of the Viterbi algorithm except for the case of Fibonacci words and the Viterbi$_L$. Since Fibonacci words compress well this is expected. As the compressed sequence is very short as seen in section 5.3 the running time of the actual Viterbi computation will not depend on $T$ and thus it will be close to constant for all sequence lengths. However, for the compression to make the sequence that short more iterations are required for long sequences than for short sequences. Hence, the running time of the compression will increase for increasing sequence lengths and become a larger fraction of the running time.

In terms of performance gains, making the preprocessing more efficient will give a speedup gain when the number of executions of the Viterbi algorithm is low and primarily for smaller sequences or if the input sequences are very repetitive and the Viterbi path is not requested. Higher efficiency in the Viterbi algorithm or in the effectiveness of the compression will make the running time smaller especially for large values of $e$. 
5.5. VITERBI

![Figure 5.11: Fraction of the running time spent on preprocessing and the Viterbi algorithm for random sequences and Fibonacci words of varying length.](image)

5.5.3 Summary

This ends the section of experiments for the Viterbi algorithm. In conclusion the compression of sequences does speed up the execution time of the Viterbi algorithm in many cases. The best results are of course found for repetitive data with the Viterbi algorithm run multiple times and the Viterbi path not computed. Speedup factors in the order of hundreds is found, so for these kinds of problems this method has a great potential. If the Viterbi path is requested the speedup factors becomes much lower. The maximum speedup seen in the experiments is below 20. In other scenarios with random data a more modest speedup is obtained. For random data and request of the Viterbi path speedup factors below 1 is obtained. Hence, the zipHMMlib Viterbi is slower than the original Viterbi algorithm in some cases. The algorithm performs very badly on large models. This is partly due to the alphabet size
becoming large when the sequence is compressed, but it is also due to the matrix multiplication algorithm. As seen, cache misses occur when working with large models, so the algorithm will be faster for larger models if a better algorithm for multiplying matrices were implemented.

5.6 Posterior Decoding

This section contains all experiments performed on the posterior decoding algorithm. As described in section 3.7 no efficient way of exploiting sequence repetitions has been found. Nevertheless, experiments have been made to see whether a minor speedup can be obtained by computing the posterior decoding in the matrix multiplications framework in zihmmLib. In contrast to the previous section with multiple variations of the Viterbi algorithm, only a single variation makes sense here: the input sequence is not compressed and the path is computed.

The section is split into two parts. The first part compares the actual running time to the theoretical running time to verify that the implementation of the algorithm fits the theoretical running time. The second part compares it to the original implementation of posterior decoding.

5.6.1 Asymptotic Running Time

As in section 5.5.1 the theoretical compared to the actual running time experiments have been made using random data. The theoretical running time is \( O(MN^3 + TN^2) \). Since \( M \) is very small compared to \( T \), it is expected that the first term vanishes. This has already been seen for the Viterbi algorithm in section 5.5.1, figure 5.7. Hence, for this experiment the theoretical running time \( O(TN^2) \) is assumed. This of course only makes sense if \( M \) is not too large, but as seen in the plots the assumption of a theoretical running time of \( O(TN^2) \) does make sense for these experiments.

First, the running time compared to the sequence length is shown in figure 5.12. A model with 16 states has been used. As expected, the running time is decreasing going towards constant.

Secondly, in figure 5.13 the running time for an increasing number of states is shown. A random sequence of length 10000 has been used. It is seen that the running time is indeed quadratic in the number of states in practice.

It is concluded that the actual running time of the algorithm follows the theoretical.
5.6. POSTERIOR DECODING

Figure 5.12: The running time of posterior decoding is linear in the size of the input sequence.

Figure 5.13: The running time of posterior decoding is quadratic in the number of states.
5.6.2 Comparison to the Original Algorithm

Now the algorithm is compared to the original algorithm. In figure 5.14 the speedup factor as function of the sequence length is shown. Random sequences and a model with 16 states was used for the experiment. In all tested cases the algorithm is a bit faster than the original implementation. There is a tendency that the speedup factor becomes larger as the input sequence grows. This is due to the initial setup of the algorithm that slows down the algorithm a bit, which was also seen in figure 5.12, where the factor between running time and observation sequence length is larger for small sequences than for large sequences. Hence, as the sequence length is increases the initial setup time becomes relatively smaller and the speedup factor increases. Speedup factors between 1 and 5 are gained in this experiment. This might change if the number of states in the model is different.

Looking at the speedup factor as a function of the number of states in figure 5.15, it is seen that for small models it is below 1. This means that the algorithm is slower than the original posterior decoding algorithm. However, as the number of states in the model increases the speedup factor quickly becomes larger than 1, due to the BLAS framework computing matrix multiplication efficiently.

Figure 5.14: The speedup factor for sequences of varying lengths.
5.7. INDEXED POSTERIOR DECODING

5.6.3 Summary

These experiments end the section of the posterior decoding. Overall the algorithm has a running time that is comparable to the original implementation of posterior decoding with observed speedups between approximately 0.5 and 6. For scenarios with large models and sequences zipHMMlib is faster whereas the original algorithm is better for small models. This is partly due to the BLAS framework being very efficient at multiplying large matrices and partly due to sequence not being compressed resulting in the MN^3 term of the running time vanishing. This is different from the Viterbi algorithm that performs badly on large models due to an inefficient matrix multiplication algorithm and a larger alphabet.

5.7 Indexed Posterior Decoding

In this section experiments on the indexed posterior decoding are carried out. Recall that indexed posterior decoding also takes two indices i and j as input and only returns the posterior decoding for the subsequence Y_{i:j}.

Recall from section 3.8.1 that the algorithm has a theoretical running time of O(M'N^3 + N^2T' + N^2(j - i + log T)), assuming the input sequence has already been compressed. This is very similar to both the Viterbi algorithm and the posterior decoding algorithm. Therefore, the verification of the theoretical
running in practice has been omitted, since most of the implementation uses the same code as the two previously tested algorithms. However, the three terms of the running time are still discussed in turn below.

The first term $M'N^3$ comes from the computation of the $C$ matrices. This has already been experimentally verified for the Viterbi algorithm in section 5.5.1. Since the only difference between the computations of the matrices for the Viterbi algorithm and the forward-backward algorithm is matrix multiplication vs. max-times multiplication and numerical stability, the theoretical running time will also hold for the indexed posterior decoding.

The second term $N^2T'$ comes from computing the forward and backward tables for the compressed sequence of length $T'$. In section 5.6.1 this was already verified for the posterior decoding algorithm.

Finally, the $N^2(j - i + \log T)$ term comes from the decompression of the compressed subsequence that is linear in $j - i + \log T$ and from computing the posterior decoding for the subsequence. The running time of the posterior decoding has already been shown to match the theoretical running time. The decompression is trivial and as the plots is this section look as expected experiments have not been made for this.

5.7.1 Comparison to the Original Algorithm

The indexed posterior decoding algorithm has been compared to the original posterior decoding algorithm. For the original algorithm the entire posterior decoding has been found and the state sequence from index $i$ to $j$ has been extracted.

As for the Viterbi algorithm experiments, the comparison to the original implementation is first made for random data and then for Fibonacci words. As for the Viterbi algorithm the algorithm is analyzed by varying the input sequence length and the model size, but as the length of the partially decompressed the subsequence $Z$ also has an impact on the running time the value of $j - i$ is also varied.

Random Data

For the first experiment the distance between the two indices $i$ and $j$ has been varied for random input sequences of length 10,000 and a model with 16 states. For this experiment the “extra” symbols corresponding to the indices $[k, i)$ and $(j, l]$ can be left out as there is at most $2 \times \log 10,000$ of these according to the analysis in section 3.8.1, which is a small number compared to $j - i$.
5.7. INDEXED POSTERIOR DECODING

\[ e = 1 \quad e = 500 \]

**Figure 5.16:** The running time of indexed posterior decoding for varying distances between \( i \) and \( j \). Random sequences has been used.

Depending on how well the data compresses, the value of the length of the compressed sequence \( T' \) will be larger or smaller than the length of subsequence \( Z \). For random data that only has a compression ratio of approximately 4 as seen in section 5.3 the number of “extra” symbols will be small. Hence, when \( j - i \) is small the length of \( Z \) will also be small compared to \( T' \). Therefore, it is expected that \( T' \) will dominate the running time and small increases of \( j - i \) will not make major changes to the overall running time. However, for large values of \( j - i \), the value will be close to or larger than \( T' \) and the value will have a larger impact on the running time, thus increasing the running time.

This is seen in figure 5.16 where the running time is nearly constant for small values of \( j - i \), but starts to grow when \( j - i \) gets closer to \( T \), since the forward and backward tables then are computed for a large part of the uncompressed observation sequence.

In figure 5.17 the running time is compared the original algorithm. A speedup is gained when running the algorithm multiple times, but for a single run the algorithm is slower than the original algorithm. The speedup becomes smaller when \( j - i \) approaches \( T \). That is expected from the running time seen in figure 5.16 that was discussed in the previous paragraph.
Figure 5.17: The indexed posterior decoding algorithm compared to the simple algorithm for random data for varying distances between \(i\) and \(j\).

For the second experiment the length of the input sequence has been varied from small sequences of length \(10^3\) to approximately \(10^7\). As already seen in sections 5.5.2 and 5.6.2 the longer the input sequence, the better speedup for the Viterbi algorithm and the posterior decoding. It is expected that the indexed posterior decoding will be similar to the posterior decoding algorithm. The result of this experiment is shown in figure 5.18. The result is as expected. Again, the indexed posterior decoding with \(e = 1\) is slower than the original algorithm for small sequences. For the indexed posterior decoding with \(e = 500\) however, speedups of up to factor 15 are seen in this example.

Finally, the number of states in the model has been varied. Sequences of length 10000 were used and the posterior decoding has been computed for a posterior decoding of length 200. As for the Viterbi algorithm experiments in section 5.5.2 it is expected that the algorithm is most efficient for "medium sized" models. As seen in figure 5.19 that is also the case. The result is very similar to the results seen in figures 5.9a and 5.9b.
5.7. INDEXED POSTERIOR DECODING

**Figure 5.18:** The indexed posterior decoding algorithm compared to the simple algorithm for random data for varying input sequence lengths.

**Figure 5.19:** The indexed posterior decoding algorithm compared to the simple algorithm for random data for varying model sizes.
Fibonacci Words

The results of the experiments using random data showed speedups in some cases whereas in other cases slowdowns were observed. As seen for the Viterbi algorithm better results were obtained using Fibonacci words which is also expected for the indexed posterior decoding.

The experiments from the previous section is repeated with Fibonacci words instead of random sequences. It is expected that this will provide larger speedups. The result of these experiments is shown in figures 5.20, 5.21, and 5.22. As seen the curves in the plots are similar to those of random data except that the speedup factor is much larger.

5.7.2 Summary

The indexed posterior decoding algorithm also gives speedups. As for the Viterbi algorithm the speedup depends heavily on the value of e. For e = 1 slowdowns often occur and the maximum speedup is a factor four. For e = 500 no slowdowns is observed and speedups of up to a factor 15 is observed. Compared to the Viterbi algorithm the indexed posterior decoding works better with larger models since the BLAS framework multiplies matrices in a more efficient way. However, the $MN^3$ factor of the running time limits
5.7. *INDEXED POSTERIOR DECODING*

The indexed posterior decoding algorithm compared to the simple algorithm for a Fibonacci word for varying substring lengths.

![Graph showing speedup factor vs. j - i](image)

**Figure 5.21:** The indexed posterior decoding algorithm compared to the simple algorithm for a Fibonacci word for varying substring lengths.

The indexed posterior decoding algorithm compared to the simple algorithm for Fibonacci words of varying lengths.

![Graph showing speedup factor vs. sequence length T](image)

**Figure 5.22:** The indexed posterior decoding algorithm compared to the simple algorithm for Fibonacci words of varying lengths.
the speedup when models become too large. For Fibonacci words a better speedup is seen again. If the length of the requested decoding $j - i$ is much smaller than the observation sequence length $T$ a speedup in the order of hundreds is observed.
Chapter 6

Conclusion

Sand et al. (2013) implements a zipHMMlib for working with hidden Markov models. It includes an efficient implementation of the forward algorithm that obtains a speedup compared to an implementation of the original algorithm by exploiting repetitions in the input sequence.

In this master’s thesis the zipHMMlib library has been extended to include implementations of the Viterbi, the backward and two variants of the posterior decoding algorithms. This includes description of the theory behind the algorithm, efficient implementations of these, and experiments that evaluates the performance compared to the standard implementation.

By exploiting repetitions in the observation sequence the running time of the Viterbi algorithm for computing the probability of the most likely path has changed from $O(N^2 T)$, with $N$ being the number of states and $T$ being the length of the observation sequence, to $O(M'N^3 + N^2T')$ where $T'$ is the length of the compressed observation sequence and $M'$ is the new alphabet size. If the Viterbi path is required the running time has changed to $O(M'N^3 + N^2T' + T)$. This requires that the observation sequence has been preprocessed. This takes $O((M' - M)T)$ time with $M'$ being the new alphabet size that is dependent on the data and a user provided estimate of how many times the Viterbi algorithm is executed afterwards.

The experiments for the Viterbi algorithm show that a possibly very large speedup can be gained. However, it is very dependent on how well the data compresses, i.e. how many repetitions the observation sequence contains, and how many times the user estimates that the algorithm is executed after the compression. Furthermore, since the algorithms are cubic in the number of states a speedup is not obtained for large models and large alphabet sizes. The experiments show that a speedup between 0.5 and 20 can be a typical result.
depending on how the algorithm is used.

The backward algorithm has been implemented as part of the implementation for computing the posterior decoding. It has not been possible to find a method for exploiting repetitions in the observation sequence, so in practice it becomes a battle between constants whether the original version or the zipHMMlib version of the posterior decoding algorithm is the fastest. The running time of the original algorithm is $O(N^2T)$ while the zipHMMlib version has a running time of $O(MN^3 + N^2T)$. In the experimental setup used in this thesis the constants are in favor of the zipHMMlib implementation, when the observation sequence length and model size are large enough. In general, a speedup between 0.5 and 6 can be expected using the zipHMMlib posterior decoding.

Exploiting repetitions is however possible for posterior decoding if only a part of the posterior decoding is requested. In that case the running time becomes $O(M'N^3 + N^2T' + N^2(j - i + \log T))$ with $j - i$ being the length of the requested substring. A speedup between 0 and 60 can be expected for most cases.

zipHMMlib can be extended beyond what has been implemented in this thesis. Below is listed a number of improvements that can make the library better.

- In zipHMMlib a parallelized version of the forward algorithm is implemented. This is described in detail in Sand (2014). This can be implemented similarly for the Viterbi and posterior decoding algorithms. For Viterbi the idea is to parallelize equation (3.4) as the max-times multiplication can be computed in any order. One processor computes equation (3.4) from right to left while the remaining processors multiply matrices from the entire equation. Hence, compared to the sequential version the first processor does not need to multiply by matrices already multiplied by other processors. The first processor will compute matrix-vector multiplication while the remaining processors will compute matrix-matrix multiplications. Hence, this parallelization scheme will work best for small models. Note that $\delta_t$ is not computed for all $t \in [1, T]$, since only the first processor makes matrix-vector multiplications. Hence, backtracking becomes harder. One solution to this is given in Sand (2014), where the missing $\delta_t$ vectors are computed and then the path is backtracked. Another idea is to compute extra $R$ matrices for each of the matrix-matrix computations. When the subset of the $\delta_t$ matrices and the path corresponding to these have been computed, the
ones missing can be found using the extra $R$ matrices using the same technique as in section 3.4.3.

- At the current state zipHMMlib compresses the sequences using a single compression stopping criterion as discussed in section 3.4.7. Since a BLAS framework is used for the implementation of the forward and backward algorithms and a naive max-times matrix multiplication is used for Viterbi multiple compression criterions could be used. Also, the criterion could depend on whether the Viterbi path is requested or not. One solution is to compute multiple compressed sequences from a single sequence such that the best compression is used for any algorithm.

- A minor improvement of the indexed posterior decoding algorithm would be to save the uncompressed sequence along the compressed sequence. In that case no decompression would be needed. However, preliminary experiments suggest that this will not improve the running time significantly.

- As seen in section 5.5.1 the max-times matrix multiplication results in cache misses for large models. This problem can be solved by either implementing a better max-times multiplication algorithm that takes the underlying hardware into account or by storing two copies of each matrix with different memory alignment of the data.

- At the current state the indexed posterior decoding computes the forward and backward tables, $\alpha'$ and $\beta'$, and then the indexed posterior decoding itself for a pair of indexes $i$ to $j$. If the indexed posterior decoding is wanted for multiple pairs of indexes $i_1$ to $j_1$, $i_2$ to $j_2$, etc. $\alpha'$ and $\beta'$ are recomputed each time. These could be saved in memory once computed and reused for each of the pair of indexes.

- When having an indexed posterior decoding algorithm it would also make sense to implement an indexed Viterbi algorithm. This should be faster than computing the entire Viterbi path.

- Finally, Sand et al. (2013) have provided bindings to the Python and R programming languages. This has not yet been done for the implementations made in this thesis.

In most cases the added algorithms to zipHMMlib perform better than the original algorithms. The algorithms exploiting repetitions should not be used with large models, but this is also normally preferred as the original
algorithms have a running time proportional to the square of the number of states. In a few cases major speedups in the order of hundreds are obtained, but in most cases the speedups are more modest between factors 1 and 10. In some cases the speedup factor is below 1 which correspond to a slowdown.

As the running time of the algorithms on large data set in some applications can be measured in hours or days small speedups of a few factors can be very useful. The original algorithms are very simple and thus it can also hard to improve on those. In this thesis it is shown that compression of the input can speed up the algorithms, but that it is only by a small factor in many cases.
Bibliography


