Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

The on-line Viterbi algorithm

(Master's Thesis)

Rastislav Šrámek

Advisor:
Tomáš Vinař, Ph.D.

Bratislava, 2007

I hereby declare that I wrote this thesis by myself,
only with the help of the reference literature, under
the careful supervision of my thesis advisor.

. . . .

Acknowledgments

I would like to thank Tomáš Vinař and Broňa Brejová for above-standard and patient guidance as my advisors.

Abstract

Hidden Markov models (HMMs) are probabilistic models that have been extremely successful in addressing problems in bioinformatics, error-correcting codes, natural language processing, and other important areas. In many of these applications, the Viterbi algorithm is used to find the most probable state path through the model generating sequences that can be extremely long. Known algorithms either use at least $\Omega(n)$ memory in the best case and always find the most probable state path, or use $o(n)$ memory in the worst case, but do not guarantee correct result.

We introduce a modification of the Viterbi algorithm which is guaranteed to find the most probable state path and uses anywhere between $O(1)$ and $O(n)$ memory, depending on the model and the input sequence. For a simple class of models, we show that the expected memory needed to process a sequence is $O(\log(n))$. We present results from experiments with more complicated models for the problem of gene finding in bioinformatics that indicate similar expected memory requirements.

KEYWORDS: hidden Markov models, Viterbi algorithm, information theory, gene finding

Contents

Abstract **7**

1 Introduction **3**

2 Previous results **7**

2.1 Hidden Markov models 7

2.1.1 Forward and backward probabilities 7

2.1.2 The most probable state path 10

2.1.3 Posterior decoding 11

2.1.4 Hidden Markov model training 13

2.1.5 Higher order Markov models 15

2.2 Case study: Gene finding 15

2.3 Convolutional code decoding 17

2.4 Improving the Viterbi algorithm 19

2.4.1 Correct algorithms 20

2.4.2 On-line algorithms 21

3 The On-line Viterbi Algorithm **27**

3.1 Running time analysis 28

3.2 Space analysis 29

3.2.1 Best case space analysis 29

3.2.2 Worst case space analysis 31

4 Two-state hidden Markov models **35**

4.1 Symmetric models 35

4.2 General models 41

5 Multi-state hidden Markov models **43**

6 Conclusion **47**

Abstrakt **51**

Chapter 1

Introduction

Hidden Markov models are probabilistic generative models that have hidden states and in each state generate observable emission. The transitions between hidden states are governed by a Markov chain and the emissions from each state are governed by a distinct probability distribution. The principal use of hidden Markov models is sequence annotation. On input, we have a sequence of observations generated by a hidden Markov model and we would like to know its annotation, the concrete sequence of hidden states that have generated each observation. Obviously, this problem usually does not have an exact and correct solution. Instead of finding the actual annotation, or state sequence, that have generated the observed sequence, we will try to find the state sequence that would have generated the observed sequence with the highest probability.

Hidden Markov models are a tool used in a multitude of applications. They include convolutional code decoding [21], speech recognition [18], protein secondary structure prediction [13] and gene finding [4]. The problem is usually to determine, what has caused observable outcome. In the problem of convolutional coding, we determine the data that were encoded from the resulting code sequences; in speech recognition we seek the words that were said by measuring the sound frequencies.

The sequences of observations we are trying to annotate can be very long or even continuous. This is indeed the case for convolutional code decoding and speech recognition; we would like to be able to construct devices that decode convolutional codes on-line and for arbitrary lengths of time. Same applies for systems that do speech recognition. In other applications, the sequences may be finite, but still very large. For instance DNA sequences used as an input for gene finding have tens of millions of symbols.

The principal method of sequence annotation in hidden Markov models is the linear time dynamic programming Viterbi algorithm. The algorithm fills in a dynamic programming matrix. Every element of the matrix is derived from one of the elements in previous column and denotes the probability, that the most probable path to generate the prefix of the observation sequence ending in the corresponding symbol ends in the corresponding state. To find the most probable state path, the sequence annotation, the algorithm finds the maximum in the last column and finds the states, from which was this number sequentially generated.

The Viterbi algorithm in its original form is insufficient for annotation of long or continuous sequences. It needs to finish reading the complete sequence in order to output any result, which is impossible if the sequence is continuous. The situation is not much better with long finite sequences; the algorithm will require the length of the sequence times the number of states of system memory. This may be more than we can allocate. Various solutions of this problem have been proposed and are used in praxis. Some modifications of the Viterbi algorithm address the issues with continuous sequences; they use only bounded amount of memory and generate output before the whole sequence is read. These solutions however sacrifice the property of always returning the most probable state path, they may generate sub-optimal results. Different modifications lower the memory requirements substantially, but they increase the running time and need to finish reading the sequence in order to begin outputting data.

In this work, we propose the on-line Viterbi algorithm, a different modification of the Viterbi algorithm that addresses both of these problems at the same time. The "derived from" relationships between the matrix entries form a tree. If we start tracing the branches of the tree that correspond to the most probable state paths from the states we are currently processing (the last column of the dynamic programming matrix), we will often find that they coalesce in one point. The paths will be identical beyond this point and we can be sure that the paths beyond this point will be in the correct output. Our algorithm exploits this property by effectively detecting such coalescence points and outputting the unambiguous path prefixes. It is only negligibly slower than the original Viterbi algorithm.

A central part of our work focuses on the memory complexity of the on-line Viterbi algorithm. The memory it requires to process a sequence depends on the properties of the hidden Markov model and on the properties of the sequence itself. While in the best case it has ideal constant memory complexity, worst case is linear, as bad as the original algorithm. However, we proved that there exists no algorithm with same properties and better worst case complexity. Expected maximum space complexity is therefore of interest, as it determines how much memory should be enough to process a sequence of certain length.

First, we studied symmetrical two-state models. We transformed the problem to a problem of random walk on integers, or the probability of ruin in a game of chance. Using results from the theory of random walks and extreme value theory, we succeeded in showing that the expected maximum space grows logarithmically with the length of the sequence.

We did not find a way to extend these results to multi-state HMMs. Instead, we implemented the algorithm and used it to solve the problem of gene finding. We found, that the maximum space complexity seems to grow logarithmically and that the algorithm uses only modest amount of memory. Indeed, it was able to process whole DNA chromosomes at once and achieved running time comparable with the Viterbi algorithm, before the Viterbi algorithm ran out of memory.

The rest of this thesis is organized as follows. In first part of Chapter 2, we introduce hidden Markov models and related important algorithms. Then we introduce the problems of gene finding and convolutional code decoding and we finish with showing modifications of Viterbi algorithm that address individual issues of the Viterbi algorithm. In Chapter

3 we introduce our on-line Viterbi algorithm and present proofs for the time complexity and best and worst case memory complexities. In Chapter 4 we prove expected memory complexity for two-state HMMs and in chapter 5 we present experimental results.

Chapter 2

Previous results

2.1 Hidden Markov models

Definition 1 *Hidden Markov model (HMM) is a 5-tuple* $M = \{S, \Sigma, T, E, \Pi\}$*, where* $S = \{S_1, S_2, \ldots, S_m\}$ *is a finite set of hidden states,* Σ *is an output alphabet,* $T, \{\{t_i(j)\}_{i=1}^{m}\}_{j=1}^{m}$ *is a matrix of transition probabilities between states,* $E, \{\{e_i(j)\}_{i=1}^{m}\}_{j=1}^{|\Sigma|}$ *is a matrix of emission probabilities,* $\Pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ *is the initial state distribution.*

Hidden Markov model generates sequences of observations over an alphabet Σ. Such generative process starts in of the m states chosen according to the initial distribution Π. At each step of this process, a new state is chosen according to the probability distribution given by the vector $\{t_k(1), \ldots, t_k(m)\}$, where k is the current state, and a symbol is emitted according to the distribution given by the vector $\{e_l(1), \ldots, e_l(m)\}$, where l is the state chosen in previous step. We call a sequence of these states a *state path*.

Definition 2 (State path) *We will call a sequence* $s_1, \ldots, s_n$ *of HMM states a viable state path if for every* $1 \leq i < n$, $t_{s_i}(s_{i+1}) > 0$.

Definition 3 *A HMM generates a sequence* $X_1, \ldots, X_n$ *if there exists a viable state path* $s_1, \ldots, s_n$ *for which* $e_{s_i}(X_i) > 0$ *for all* $1 \leq i \leq n$*. Furthermore, the probability of generating sequence* X *is* $e_{s_n}(X_n)\pi_{s_1} \prod_{i=1}^{n-1} t_{s_i}(s_{i+1}) e_{s_i}(X_i)$*, the product of the initial probability all transition and emission probabilities.*

In common applications of HMMs, the internal states of the model are not observable, thus the states are said to be *hidden*. We can observe only the emitted symbols (sequences). Our goal will be to extract some information about the internal states from the model parameters and emitted symbols.

2.1.1 Forward and backward probabilities

A simple task is to determine the probability of a sequence being generated by a model.

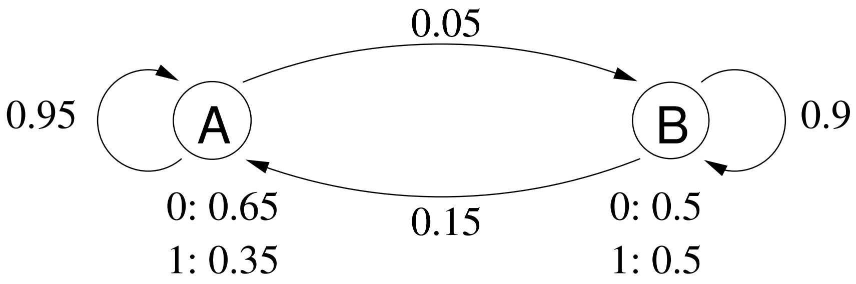


Figure 2.1:

A simple Hidden Markov model with $S = \{A, B\}$, $\Sigma = \{0, 1\}$, $T = \begin{pmatrix} 0.95 & 0.05 \\ 0.1 & 0.9 \end{pmatrix}$, $E = \begin{pmatrix} 0.65 & 0.35 \\ 0.5 & 0.5 \end{pmatrix}$

Problem 1 *Given a Hidden Markov model M and sequence $X = X_1X_2 \dots X_n$, compute the probability $P(X)$ that the sequence X was generated by M.*

We will show two methods of obtaining this probability. Even though this may seem redundant, we will use both of these methods to solve more complicated problems later.

Definition 4 (Forward probability) *Let $\alpha_t(i) = P(s_t = i, X_1, X_2, \dots, X_t)$ be the probability that the HMM generates sequence $X_1X_2 \dots X_n$, ending the generative process in state i. We will call $\alpha_t(i)$ the forward probability of sequence $X_1X_2 \dots X_i$ ending in state i.*

The forward probability of a sequence X with model M is the sum of the probabilities of sequence X of all possible state paths. A brute-force method would be to generate all possible state paths, determine the probability of generating X for each and sum these probabilities. The problem however be solved by a simpler and faster dynamic programming algorithm. The recurrence used will be based on the fact that $\alpha_{i+1}(j)$, the probability of being in state S_j and having generated sequence $X_1X_2 \dots X_iX_{i+1}$ can be expressed as a sum of the probabilities of the state paths that generate sequence $X_1X_2 \dots X_i$ and end in the states S_i for all i in $\{1, \dots, m\}$, multiplied by the transition probability into state S_j and the emission probability of X_{i+1}. The recurrence $\alpha_{i+1}(j) = \sum_{k=1}^{m} \alpha_i(k)t_k(j)e_j(X_{i+1})$ holds for all states, which leads us to a simple dynamic programming algorithm (Figure 2.2), the forward algorithm ([14]).

The answer to problem 1 is the sum of the probabilities of generating sequence X and ending in a particular state, $P(X) = \sum_{i=1}^{m} \alpha_n(i)$.

Theorem 1 *The forward algorithm computes the forward probabilities in $O(nm^2)$ time using $O(m)$ space.*

We only need to store the vectors α_{i-1} and α_i while calculating α_i. Therefore, if we are only interested in solving problem 1 $O(m)$ memory is sufficient. However in some applications,

```
Forward($M$, $X$)
for $i = 1$ to $m$ do
    $\alpha_0(i) = \pi_i$ {Base case}
end for
for $j = 1$ to $n$ do
    for $i = 1$ to $m$ do
        $\alpha_j(i) = \sum_{k=1}^{m} \alpha_{j-1}(k) t_k(i) e_i(X_j)$ {Recurrence}
    end for
end for
return $\alpha$
```

Figure 2.2: Forward algorithm

```
Backward($M$, $X$)
for $i = 1$ to $m$ do
    $\beta_n(i) = e_i(X_n)$ {Base case}
end for
for $j = n - 1$ to $1$ do
    for $i = 1$ to $m$ do
        $\beta_j(i) = \sum_{k=1}^{m} \beta_{j+1}(k) t_i(k) e_i(X_j)$ {Recurrence}
    end for
end for
return $\beta$
```

Figure 2.3: Backward algorithm

it will be convenient to pre-calculate values of α_i, for all $1 \leq i \leq n$, the space complexity would be $O(nm)$ in this case.

Another way to solve problem 1 is by summing the probabilities that the sequence starts at state S_i and continues to generate X.

Definition 5 (Backward probability) *Let $\beta_t(i) = P(X_{t+1}, X_{t+2}, \ldots, X_n | s_t = i)$ be the probability that if the HMM is in state i at time t, it generates the sequence $X_{t+1}, X_{t+2}, \ldots, X_n$ in the following steps.*

The backward probability $\beta_t(i)$ can be computed recursively in a similar fashion as the forward probability However, the computation will progress from right to left (see Figure 2.3).

Analogically to the forward algorithm, $P(X) = \sum_{i=1}^{m} \beta_1(i)\pi_i$ is the answer to problem 1.

Theorem 2 *The backward algorithm computes the backward probabilities in $O(nm^2)$ time using $O(m)$ space*

As in the case of the forward algorithm, we will sometimes need to store all vectors β_i. In such case, the space complexity will be $O(nm)$.

2.1.2 The most probable state path

One of the most important features of hidden Markov models is an ability to deduce information about the hidden states of the model. In this section we show how to compute the most probable state path that could have generated the observed sequence. Knowing such path reveals some information about the "hidden" states of the HMM, and can help solve a variety of problems, as we will show in next sections.

Problem 2 *Given a hidden Markov model M and an observed sequence $X = X_1, X_2, \ldots, X_n$, find the most likely sequence of states $s = s_1, s_2, \ldots, s_n$ which could have generated X. We call the sequence s the* most probable state path *or* sequence annotation *and the process of obtaining the most probable path* decoding.

The Viterbi algorithm ([20], [7]), solves the problem 2. The algorithm consists of two phases. In the first phase we use simple dynamic programming to calculate the probability of the most probable state path ending in each state. In the second phase we use data gathered in the first phase to find the most probable state path.

Definition 6 *Given sequence $X_1, X_2, \ldots, X_t$, let $\gamma_t(i)$ be the probability of the most probable sequence of states $s_1, \ldots, s_{t-1}$ that ends in state i i.e., $s_t = i$.*

We can compute $\gamma_i(j)$ in a similar way as the forward probability $\alpha_i(j)$. The algorithm is summarized in Figure 2.4. The difference between the forward and Viterbi algorithms is that we are interested only in the single most probable path, not in the sum of all paths generating a particular sequence. We calculate the value of $\gamma_i(j)$ as the maximum probability of the state path terminating in state j, using previous values of γ_{i-1}. The value $\gamma_i(j)$ is therefore derived from exactly one of the previous values $\gamma_{i-1}(1), \ldots, \gamma_{i-1}(m)$. To simplify the computation in the second phase, we will remember the state $\delta_i(j)$, from which the value of $\gamma_i(j)$ is derived (see figure 2.4, line 8). This is the second-last state of the most probable state path ending in state j. We finish the first phase of the algorithm by computing the probability $P(s, X)$ of the most probable path s, which will be the maximum value in the vector γ_n.

In the second phase of the algorithm, which we call *back-tracing*, we reconstruct the sequence of states in the most probable state path. We do this by starting from the maximum value in the last vector and working our way back to the beginning of the sequence by following the back pointers which we stored as $\delta_i(j)$. We will then output this sequence of states (in reverse order) as the most probable state path (Figure 2.4, lines 11–14).

In the actual implementation of Viterbi algorithm, we first take logarithms of all values of $t_i(j)$ and $e_i(j)$. This allows us to replace all multiplications by (faster) addition. Further, as probabilities in the model will be small, multiplying even few of them would cause underflow of double-precision variables. On the other hand, situations where we would need to normalize the logarithms of probabilities in order to avoid underflow are extremely rare. Another simplification is that the majority of models with m states have fewer than

```
**Viterbi**($M$, $X$)
 1: for $i = 1$ to $m$ do {Initialization}
 2:     $\gamma_0(i) = \pi_j$
 3:     $\delta_0(i) = 0$
 4: end for
 5: for $j = 1$ to $n$ do {First phase}
 6:     for $i = 1$ to $m$ do
 7:         $\gamma_j(i) = \max_{k=1}^{m} \gamma_{j-1}(k) t_k(i) e_i(X_j)$ {Recurrence}
 8:         $\delta_j(i) = \arg\max_{k=1}^{m} \gamma_{j-1}(k) t_k(i) e_i(X_j)$
 9:     end for
10: end for
11: $s_n = \arg\max_{k=1}^{m} \gamma_n(k)$
12: for $i = n - 1$ to $1$ do {Second phase (back-tracing)}
13:     $s_i = \delta_{i+1}(s_{i+1})$
14: end for
15: return $s$
```

Figure 2.4: Viterbi algorithm

$\binom{m}{2}$ non-zero state transitions. Therefore we can use adjacency list instead of the sparse adjacency matrix T. The recurrence us ed to calculate the matrix entries would thus be $\lambda_i(j) = \max_{k,t_k(j)\neq 0}\{\lambda_{i-1}(k) + \log(t_k(j)) + \log(e_j(X_i))\}$. Finally, it is sufficient to store only the matrix δ and the last column of matrix γ at any given moment, reducing the actual space requirements.

Theorem 3 *Given a hidden Markov model M and observed sequence X, the Viterbi algorithm finds the most probable state path in $\Theta(n|E|)$ time, where $|E|$ is the number of non-zero transitions, using $\Theta(nm)$ space.*

Proof. We compute $\gamma_i(j)$ for all states j and for all symbols X_i in the observed sequence. Using the adjacency list representation of T, we require in total $\sum_{j=1}^{m}\sum_{k=1}^{m}(t_k(j) \neq 0) = |E|$ comparisons to find the maximum in lines 7 and 8, $|E|$ being the number of non-zero probability transitions in the model. Using the adjacency matrix representation, we would need $\binom{m}{2} = O(m^2)$ comparisons. The back-tracing in lines 11-14 takes only $O(n)$ time, as each of the n back pointers is followed once. The running time is $\Theta(n|E|)$.

The space complexity of the Viterbi algorithm is $\Theta(nm)$, as we need to store a $n \times m$ matrix of either values or back pointers.

□

2.1.3 Posterior decoding

The Viterbi algorithms gives us the single most probable path that could have generated a sequence. In a situation, where multiple paths have high probability, the single most

```
**Forward-backward**($M$, $X$)
$\alpha$ = **Forward**($M$, $X$)
$\beta$ = **Backward**($M$, $X$)
**for** $j = 1$ to $n$ **do** {Loop}
    **for** $i = 1$ to $m$ **do**
        $\epsilon_j(i) = \alpha_j(i)\beta_j(i)e_i(X_i)^{-1}$ {$e_i(X_i)$ is in both $\alpha_j(i)$ and $\beta_j(i)$}
        $t_i = \arg\max_k \epsilon_j(k)$
    **end for**
**end for**
**return** $t$
```

Figure 2.5: Forward-backward algorithm

probable path can differ from the rest and, in fact, can be less relevant if we want to find out the states, through which a number of high-probability states pass, or the path that has parts shared by more high-probability state paths [3].

The problem 2 can be modified, and we can get a different solution, perhaps more relevant in some situations.

Problem 3 *Given hidden Markov model M and sequence $X = X_1, X_2, \ldots, X_n$, find for each position $1 \leq i \leq n$ the state t_i that most likely generated symbol at that position. The process of finding such states is called* posterior decoding*.*

Definition 7 (Posterior state probability) *Let $\epsilon_t(i) = P(X_1, X_2, \ldots, X_n, s_t = i)$ be the probability that the HMM generates sequence $X = X_1, X_2, \ldots, X_n$ and passes through state i at time t.*

To find out the posterior state probability for all states, we will use forward and backward probabilities, defined on pages 8 and 9. The forward probability $\alpha_i(j)$ is the probability of generating the sequence $X_1, \ldots, X_i$ and ending in state j, the backward probability $\beta_i(j)$ is the probability of generating the sequence $X_i, \ldots, X_n$ given that the HMM starts in state j. We will use the output of the forward and backward algorithms (Figures 2.2 and 2.3) in the forward-backward algorithm (Figure 2.5) The output for the forward-backward algorithm is a sequence of states t, such that state t_i is the most probable state the hidden Markov model is in, when it generates the sequence X.

Theorem 4 *The forward-backward algorithm finds the posterior state probabilities in $O(nm^2)$ time and $O(nm)$ space.*

A potential problem with forward-backward algorithm is that the sequence $t_1, t_2, \ldots, t_n$ may not be a viable state path (Definition 2). We can only guarantee that such path is viable in Markov models that have all state transition probabilities non-zero, but these form a small and rare subset of all hidden Markov models. In order to eliminate paths that are not viable, we will use the *posterior-Viterbi algorithm* ([5]).

```
posterior-Viterbi($M$, $X$)
$\epsilon$ = Forward-backward($M$, $X$) {We will use a version of the Forward-backward algorithm that returns $\epsilon$ instead of $t$}
for $i = 1$ to $m$ do {Initialization}
    $\zeta_1(i) = \pi_i * \epsilon_1(i)$
end for
for $j = 1$ to $n$ do {Recursion}
    for $i = 1$ to $m$ do
        $\zeta_j(i) = \max_{k=1}^{m} \zeta_{j-1}(k)(t_k(i) \neq 0)\epsilon_j(i)$
        $\eta_j(i) = \arg\max_{k=1}^{m} \zeta_{j-1}(k)(t_k(i) \neq 0)$
    end for
end for
$u_n = \arg\max_{k=1}^{m} \zeta_n(k)$
for $i = n - 1$ to 1 do {Back-tracing}
    $u_i = \eta_{i+1}(u_{i+1})$
end for
return $u$
```

Figure 2.6: posterior-Viterbi algorithm

The posterior-Viterbi algorithm finds a path $v_1, v_2, \ldots, v_n$ that maximizes the average posterior probability of its states. It first calculates the posterior probabilities, then uses a slightly modified Viterbi algorithm to find such path through them (Figure 2.6).

Theorem 5 *The posterior-Viterbi algorithm will find the best possible posterior decoding in $O(nm^2)$ time and $O(nm)$ space.*

Proof The posterior-Viterbi is a combination of the forward-backward and Viterbi algorithm, first running the forward backward algorithm in $O(nm^2)$, then the Viterbi algorithm in $O(nm^2)$. Both use only $O(nm)$ space.

□

2.1.4 Hidden Markov model training

So far we have described some of the uses of hidden Markov models, but we have neglected the problem of creating them and estimating their parameters. First thing we need to do when creating a hidden Markov model is to determine its topology. A possibility would be to guess the number of states and train a model with all possible transitions (the transition graph being a complete graph on m vertices). This was however shown to be impractical and slow in practice. Usually, the hidden Markov model topology is given beforehand (designed by an expert or heuristically). We only need to determine the transition probabilities for the transitions allowed by the model topology and the emission probabilities. We will call the process of estimating the transition and emission probabilities of a hidden Markov model *training.*

In a supervised training, we receive a sequence X and the corresponding state path s that has generated X. The training is then straight-forward, but this situation is rather rare.

Problem 4 (Supervised training) *Given a hidden Markov model topology, sequence X and a state path s, find the set of transition and emission probabilities that maximizes the probability of X being generated by the state path s.*

We have a complete emission and state sequence. It is possible to simply count the number of times each emission and transition was used. We will set the transition and emission probabilities to reflect the acquired distribution of transitions and emissions used. Usually, we want to train the model on whole dataset of different emission/state sequences, not just on one. Training the model parameters on one or just few sequences will cause the model to lack generalizing property, only sequences from the training set will be generated with high probabilities. Given the dataset, a set of o training sequences X^i and corresponding state paths s^i, we will denote the count of transitions from state u to state v used in the dataset $T_u(v)$ and the number of times symbol x was emitted from state u we will denote $E_u(x)$. We will now set the transition and emission probabilities as

$$t_u(v) = \frac{T_u(v)}{\sum_{k=1}^{m} T_u(k)} \tag{2.1}$$

$$e_u(x) = \frac{E_u(x)}{\sum_{k=1}^{m} E_u(y)} \tag{2.2}$$

in order to maximize the probabilities of generating the sequences in the dataset using corresponding paths.

The problem is more complicated, when we have training sequences, but do not have access to the internal mechanism of the process we are modeling and thus, we do not know the state paths that have generated the training sequences. This is unfortunately the usual case. We want to find such transition and emission probabilities that make the generation of the training sequences most likely.

Problem 5 (Unsupervised training) *Given a hidden Markov model topology and a set of sequences $X^1, \ldots, X^o$ of lengths $n_1, \ldots, n_o$ find the set of transition and emission probabilities that maximize the probability of $X^1, \ldots, X^o$ being generated by the hidden Markov model.*

Currently, there is no known efficient algorithm to solve this problem. The Baum-Welch algorithm, a version of the expectation maximization algorithm, was proved to converge to a local maximum, which is often sufficient [1].

The Baum-Welch algorithm iteratively improves the transition and emission probabilities in order to maximize the probability of the sequences X^i being generated. We start with initial values of $t_i(j)$ and $e_k(l)$. In every iteration, we calculate the forward and backward probabilities and use them to determine the expected number of times each transition and emission event happened during the generation of sequences X^i. The newly estimated transition probabilities will be

$$t'_i(j) = \sum_{k=1}^{o} \frac{1}{\Pr(X^k)} \sum_{l=1}^{n_k} \alpha_l^k(i) t_i(j) e_j(X_{l+1}^k) \beta_{l+1}^k(j) \tag{2.3}$$

where X_i^j is the i-th symbol of j-th input sequence, $\alpha_i^j(k)$ and $\beta_i^j(k)$ are the forward and backward probabilities for i-th symbol of k-th input sequence and state j. We summed the probabilities that transition from state i to state j was used for every position in every sequence and then normalized the sum for each sequence by dividing by the probability of that sequence being generated. New emission probabilities $e'_i(j)$ will be determined in a similar manner.

$$e'_i(j) = \sum_{k=1}^{o} \frac{1}{\Pr(X^k)} \sum_{l=1, X_l=j}^{n_k} \alpha_l^k(i) \beta_l^k(i) e_i(X_i^k)^{-1} \tag{2.4}$$

We are summing the probabilities that each symbol j in the sequence was generated by state i and again normalizing by dividing by the probability of generating each sequence. As emission probability $e_i(X_i^k)$ is in both forward and backward probability, we need to divide each summand by it.

The algorithm usually terminates, when the change in transition and emission probabilities between succeeding iterations is small and local maximum has been reached.

2.1.5 Higher order Markov models

In a k-th order hidden Markov model, the emission probabilities depend on the k previous emissions: $e_i(j) = P(X_i | X_{i-1} X_{i-2} \dots X_{i-k})$. The hidden Markov models described until this point were of order 0. Instead of the matrix E, a $k+2$ dimensional matrix E_k needs to be used. The first two dimensions will denote the state and emission symbol as in E, the rest k dimensions will determine the previous k symbols. Only trivial changes need to be done to here described algorithms to convert them to use emission matrix E_k instead of E. For instance in the Viterbi algorithm (Figure 2.4) all instances of $e_i(X_j)$ need to be replaced by $e_i(X_j, X_{j-1}, \dots, X_{j-k})$. The case where $j < k$ needs to be treated specially, for instance, by averaging values for the unknown sequence symbols.

2.2 Case study: Gene finding

The decoding of hidden Markov models is a powerful tool. It has been successfully used in many fields, among others bioinformatics, speech and pattern recognition and information theory. In the this section, we will focus on gene finding, one of the applications of hidden Markov models in bioinformatics, in order to bring forth a more concrete example.

Genetic information of each organism is stored in a macro-molecule called deoxyribonucleic acid (DNA). Information stored in DNA is coded by a sequence of four types of nucleotides: adenine, cythosine, guanine and thymine. In this thesis, we will view DNA as a string over the four letter alphabet $\Sigma = \{a, c, g, t\}$. The *genome* of an organism is a complete set of its DNA. It consists of a number of *chromosomes*, continuous strings of DNA. The process of determining the DNA string of an organism is called *sequencing.* DNA of a number of organisms has been sequenced to date.

The central dogma of molecular biology explains, how the information in DNA is used to synthesize proteins, the building blocks of all organisms. From this point of view, DNA logically consists of three types of subsequences: *intergenic regions*, *introns* and *exons.* Introns and exon together form genes, units that code one or more complete proteins. Exons are gene subsequences that directly translate into amino acids. Introns are subsequences that are not used at all or have structural, regulatory, or other function. Sometimes, it is possible to divide a gene into introns and exons in more than one way, synthesizing different proteins. Intergenic regions do not code proteins and are nucleotide sequences that can have regulatory or structural function, but are mostly junk DNA that is not used at all.

The central dogma is illustrated in Figure 2.7. The DNA is first transcribed into pre-mRNA, with chemical, but no logical change done. In the next step, introns are left out (spliced out) of the pre-mRNA, to form mRNA. The adjacent exons are joined together and are translated into amino acids, each amino acid being coded by three nucleotides (together called a *codon*). These amino acids then form a protein. There are 20 different amino acids synthesized by living cells, and $4^3 = 64$ possible codons. Some amino acids can be coded by different codons and some codons have structural function (like coding the beginning or end of a gene).

Given a known DNA sequence, the problem of segmenting the sequence to intergenic regions, introns and exons is called *gene finding.* When we know the location of a gene in the DNA and its introns and exons, we can determine the amino acids and thus the protein the amino acids form. One of the methods used to predict the segmentation of DNA uses HMM decoding. If we have a HMM that captures the properties of DNA with states corresponding to various parts of intergenic, intron and exon sequences that emit symbols from $\{a, c, g, t\}$, we can treat the DNA sequence as a sequence generated by our model and decode it to obtain the most probable state path. As state of the most probable state path will correspond to intron, exon or intergenic region; the decoding gives us a segmentation of the DNA.

This approach to gene-finding is implemented in a number of gene finders. For instance GENSCAN, described in [4] or ExonHunter, described in [2]. Such gene finders often use HMMs in concert with other methods to determine the DNA segmentation.

The optimal method is to process the DNA by chromosomes. However, chromosomes can be tens of millions of bases long; for example the longest human chromosome has approximately 245 million bases. Simple calculation shows that using a modest model of 100 states and remembering only the back pointers $\delta_i(j)$ (1 byte per pointer), we would need at least 24.5GB of memory in order to decode the whole chromosome using the Viterbi

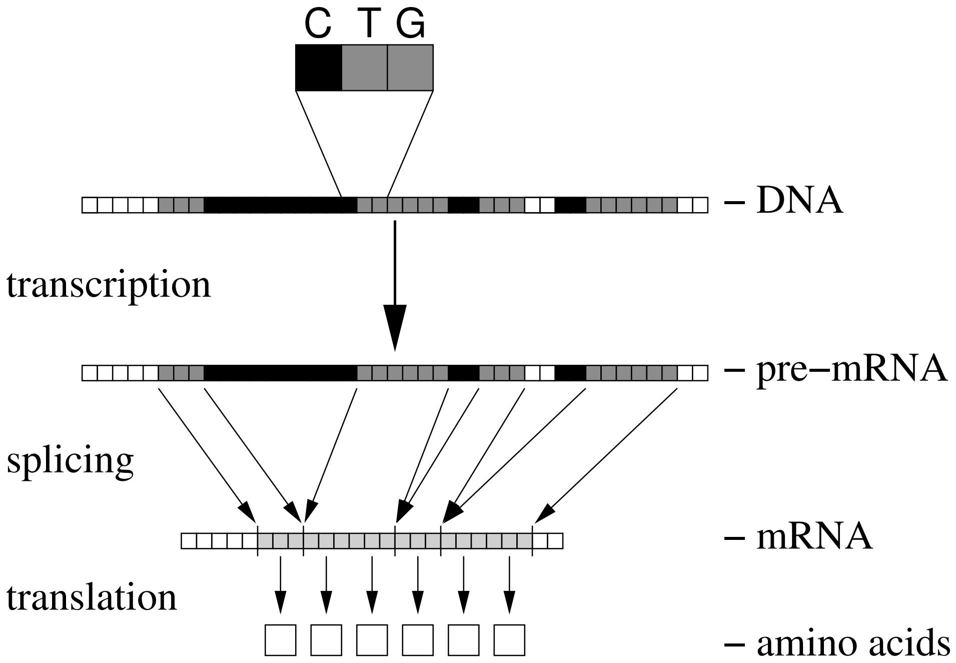


Figure 2.7: Transcription, splicing and translation of DNA

or posterior-Viterbi algorithm. This is clearly impractical, we will need to process each chromosome as more separate pieces. If we split a chromosome, the split may occur in the middle of a gene, thus lowering our chance of detecting it. In order to overcome this problem, we need a algorithm with lower space requirements than the Viterbi algorithm.

The main focus of this thesis is on the memory complexity of hidden Markov model decoding. We will look at known methods for improving this complexity in sections 2.4.1 and 2.4.2. In order to explain the algorithms in section 2.4.2, we need to introduce one more field of application for hidden Markov models.

2.3 Convolutional code decoding

The Viterbi algorithm was originally conceived for the decoding of convolutional codes [20]. Here, we present a brief and simplified version of convolutional code coding and decoding. For a more comprehensive explanation of convolutional codes see [15].

Convolutional codes are error-correcting codes, often used to increase reliability of various communication lines, such as mobile phone networks, satellite communication and deep space data links. The codes are (m, n)- codes, m bits of input being encoded by n bits ($m < n$). The ration m/n is called a code *rate*. Each bit of the encoded sequence depends on the previous k bits of the input sequence, the value k is the *order* of a convolutional code. The encoding is realized by shift-register with a set of k registers (Figure 2.8). Initially set to zero, at each step, each register takes the value of the register to its left, the left-most (first) register takes the new input bit. The output is generated by n (mod

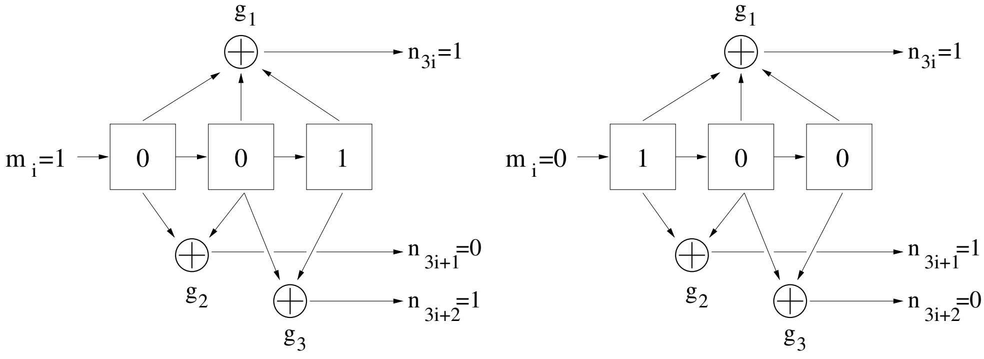


Figure 2.8: Two consecutive steps of a convolutional coder

2)-adders, each taking as input those registers that correspond to its generator polynomial. The convolutional coder in Figure 2.8 represents a $(1, 3)$ code, with generator polynomials $g_1 = x + x^2 + x^3$, $g_2 = x + x^2$ and $g_3 = x^2 + x^3$. For every bit of input i_i, the three bits of output, o_{3i}, o_{3i+1}, o_{3i+2} are generated. It is possible to construct (m, n) codes using multiple $(1, n)$ codes [21]. During the transmission, error e is added to the transmitted sequence o, resulting in the sequence $t = o + e$.

One of the ways to decode the transmitted sequence t in order to obtain the input sequence i that generated it with highest probability is to use the Viterbi algorithm [21]. We will create a hidden Markov model, corresponding to the convolutional coder. The states of the hidden Markov model will represent the states of the encoder. In above mentioned example, $S = \{000, 001, 010, \ldots, 111\}$, all the possible states of the 3 registers. In a general case, $S = \{0, 1\}^k$.

We define the transition matrix T as follows

$$t_i(j) = \begin{cases} 0.5 & \text{if transition between } S_i \text{ and } S_j \text{ is possible} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

A transition between states S_i and S_j is possible if S_j is a right shift of S_i with 0 or 1 in the first register. For example, transition between 010 and 101 is possible, but transition between 010 and 100 is not possible. Every state has two possible outgoing transitions, because either 0 or 1 can be put in the first register from the input, and two possible incoming transitions because 0 or 1 could have dropped out of the last register.

The alphabet is $\Sigma = \{0, 1\}^n$, all possible n bit outputs, $\{000, \ldots, 111\}$ in the forementioned example. In the absence of transmission errors, each state S_i can generate exactly one n-bit output $O_i = o_{3i}o_{3i+1}o_{3i+2}$. In order to be able to decode sequences with errors, we need to consider the probability of bits flipping during transmission. In a simple case, where each bit flips independently with the probability p, we would calculate the emission matrix E as

$$e_i(j) = p^{h(o_i,j)}(1 - p)^{n-h(o_i,j)} \tag{2.6}$$

where $h(o_i, j)$ is the Hamming distance of the output of state i and a symbol $j \in \Sigma$. The value $e_i(j)$ is the probability of the symbol o_i being transformed into j by the data channel.

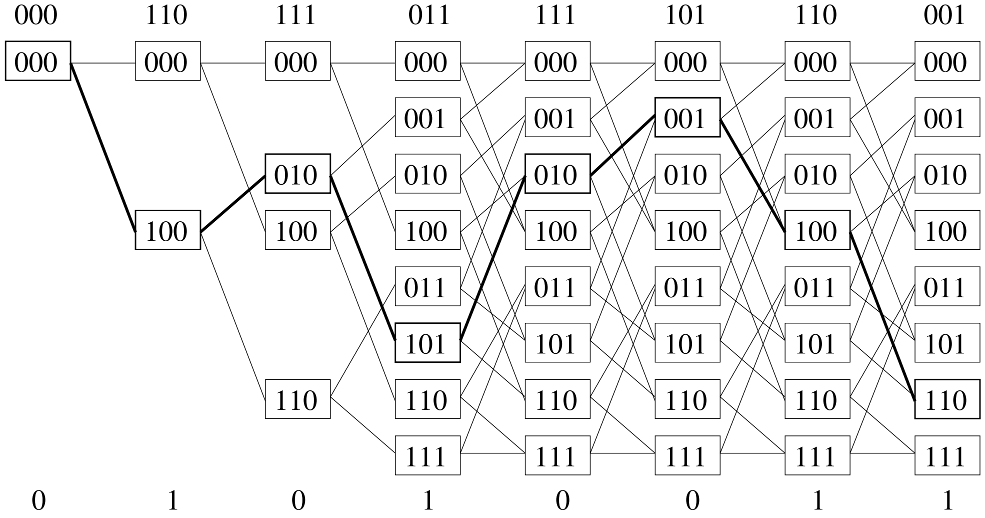


Figure 2.9: A path through trellis, with input and output sequences.

For example, given $p = 0.1$, the probability that 101 will be transformed to 001 will be $0.1 \cdot 0.9^2 = 0.081$. One possible way of graphical representation of the coding and decoding of convolutional codes using the Viterbi algorithm is the *trellis* (Figure 2.9).

The trellis consists of all possible state paths of the encoder that could have generated given transmitted sequence t. All sequences have the same trellis, as the probability of the output of the encoder o being transformed by the data channel to any other sequence of the same length is non-zero. The encoding of each sequence can be represented as a single path through the trellis (Figure 2.9). Although a trellis diagram can be used to represent sequence generation and decoding for any HMM, it is especially useful to represent convolutional code decoding where transition matrix is very sparse.

On the receiving side, we can use the Viterbi algorithm to find the most probable state path that generated the transmitted data [20]. Having the most probable state path, we can simply transform it into the corresponding input sequence by considering only the first register of each state. This would be the most probable decoding, providing that the probability p and the error model we used were correct. Devices used to decode convolutional codes in this way are called *Viterbi decoders*. They use various modification of the Viterbi algorithm, as the original algorithm needs to keep the whole (and finite) sequence in memory during the decoding. This would be a problem when the decoder should run in real-time on continuous sequences.

2.4 Improving the Viterbi algorithm

As we have shown in the previous section, the original Viterbi algorithm for finding the most probable state path, has two kinds of problems in many applications. Its memory requirements are too high, and we need to read the complete sequence to start decoding. There are several known algorithms that improve on the original Viterbi algorithm, we will divide them according to two criteria.

Definition 8 *We will call an algorithm solving the most-probable state path problem* on-line *if it returns parts of the solution before it finishes reading the input. We call other algorithms* off-line*.*

Definition 9 *An algorithm solving the most-probable state path problem is called* correct *if it always returns the most probable state path. We will call other algorithms* heuristic*.*

In this section we describe known modifications of the Viterbi algorithm that attempt to address the above mentioned issues. All of these algorithms are either off-line or heuristic. In the rest of the thesis, we then present an algorithm that is both on-line and correct, and its analysis.

2.4.1 Correct algorithms

Trivial

We can trivially lower the space complexity to $\Theta(n + m)$ while increasing the time complexity to $\Theta(n^2|E|)$. We will not store any columns of the matrices γ and δ except the current one and recompute each column from the beginning during the back-tracing run. Such algorithm would be extremely impractical on longer sequences. In the rest of this section, we show several algorithms that present more practical instances of this trade-off between memory and running time.

Checkpointing

Checkpointing algorithms, first introduced by Grice et al. [10], are based on the following idea. We divide the input sequence into K blocks of n/K symbols. During the forward pass, we store only $\gamma_{\ell(n/K)+1}$ for $0 \leq \ell < K$, the first column of each block. We will call these matrix columns *checkpoints*. After finding the maximum in the last column as in the Viterbi algorithm (Figure 2.4), we start the back-trace by recomputing the matrix columns $\gamma_{(K-1)(n/K)+2}$ to γ_n and back-tracing them (Figure 2.10). Now we can discard the columns $\gamma_{(K-1)(n/K)+2}$ to γ_n and replace them with recomputed columns $\gamma_{(K-2)(n/K)+2}$ to $\gamma_{(K-1)(n/K)}$ and continue the back-tracing. We start the re-computation of each block in the matrix from its first column, which we have stored. The pseudocode of the algorithm for a simpler case where $K|n$ is shown in Figure 2.11.

Theorem 6 *The checkpointing Viterbi algorithm can find the most probable state path in $\Theta(n|E|)$ time and $\Theta(n + m\sqrt{n})$ space.*

Proof. The algorithm recomputes every column of the matrix γ (except the K first columns in every block) twice. The complexity of the back-trace is the same as in the original Viterbi algorithm. Therefore asymptotic running time remains the same. As for space complexity,

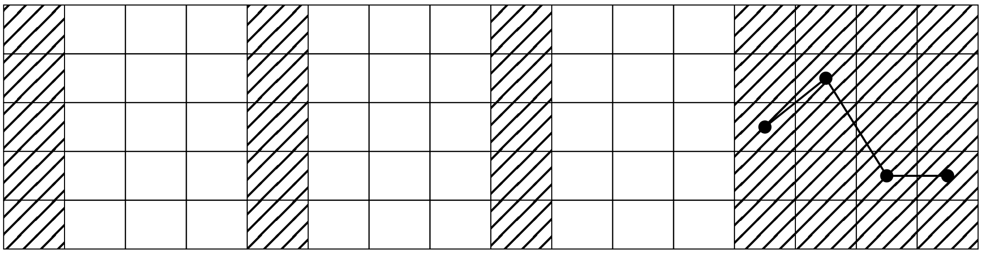


Figure 2.10: Checkpointing algorithm with $n = 16$ and $K = 4$ and the back-trace for the last block. Shaded columns are stored in the memory, all other columns have been discarded and will be recomputed at later stages.

we store at most $K + n/K - 1$ matrix columns, K checkpoints and the block of size n/K we are currently back-tracing (the first column of the block is also a checkpoint). In order to minimize the sum $K + n/K$, we will set $K = \sqrt{n}$. We also need to store the input sequence of n symbols. Therefore, we need $n + m(2\sqrt{n} - 1) = \Theta(n + m\sqrt{n})$ space.

□

The actual running time increases approximately by a factor of two compared to the original Viterbi algorithm because each column in the matrix γ needs to be recomputed twice.

Multilevel checkpointing

The idea of checkpointing can be generalized to L-level checkpointing [10]. In the checkpointing Viterbi algorithm described above, we place $\sqrt{n}$ checkpoints and then process blocks of $\sqrt{n}$ columns in a way similar to the Viterbi algorithm. In multilevel checkpointing algorithms, we further recursively checkpoint these blocks until we reach recursion level L, where we recompute the blocks of size nK^{-L} as in the original Viterbi algorithm. The optimal space is achieved for $K = \sqrt[L]{n}$. We have to recompute each column L times, but we only need to remember $L\sqrt[L]{n}$ columns, $L - 1$ layers of $\sqrt[L]{n}$ checkpoints each, and the final block of size $\sqrt[L]{n}$. The resulting space and time complexities will therefore be $\Theta(n + Lm\sqrt[L]{n})$ and $\Theta(Ln|E|)$ respectively.

2.4.2 On-line algorithms

All of the algorithms covered in the previous section were off-line. They require multiple passes of the input data. In contrast, on-line algorithms can often compute the prefix $s_1, \ldots, s_k$ of the most probable state path even before finishing reading the input sequence. In such case, the algorithm can free the memory storing the corresponding input sequence $X_1, \ldots, X_k$ and all the associated results computed by the algorithm. On-line algorithms can thus use less than $\Theta(n)$ of system memory in practice.

None of the algorithms described in previous section is suitable for the decoding of convolutional codes (see Section 2.3 for description of convolutional codes). Neither of

```
**Checkpointing-Viterbi**($M$, $X$, $K$)
for $i = 1$ to $m$ do {Initialization}
    $\gamma_0(i) = \pi_j$
    $\delta_0(i) = 0$
end for
for $j = 1$ to $n$ do {First phase}
    for $i = 1$ to $m$ do
        $\gamma_j(i) = \max_{k=1}^{m} \gamma_{j-1}(k) t_k(i) e_i(X_j)$ {Recurrence}
    end for
    if $j \mod n/K \neq 2$ and $j > 2$ then
        delete $\gamma_{j-1}$
    end if
end for
$s_n = \arg\max_{k=1}^{m} \gamma_n(k)$
for $l = K - 1$ to 1 do
    for $i = l(n/K) + 2$ to $(l+1)(n/K)$ do {Recompute $l$-th block}
        for $i = 1$ to $m$ do
            $\gamma_j(i) = \max_{k=1}^{m} \gamma_{j-1}(k) t_k(i) e_i(X_j)$ {Recurrence}
            $\delta_j(i) = \arg\max_{k=1}^{m} \gamma_{j-1}(k) t_k(i) e_i(X_j)$
        end for
    end for
    for $i = (l+1)(n/K)$ to $l(n/K) + 1$ do {Back-tracing}
        $s_i = \delta_{i+1}(s_{i+1})$
    end for
end for
return $s$
```

Figure 2.11: Checkpointing Viterbi algorithm

them has the on-line property. Therefore we would need to finish reading the data stream in order to decode it. This is impossible under most circumstances.

It is impossible to always find the correct answer to the most probable state path problem with any bounded amount of memory. More precisely, in Section 3.2.2 we prove that for any algorithm, and fixed amount of memory, we can find a HMM and an observation sequence such that algorithm will either exceed this memory allowance, or return an incorrect answer.

In this section, we present several algorithms that use only bounded amount of memory. Though they are practical for hardware implementation, they do not always find the most probable state path and therefore can only be considered heuristics.

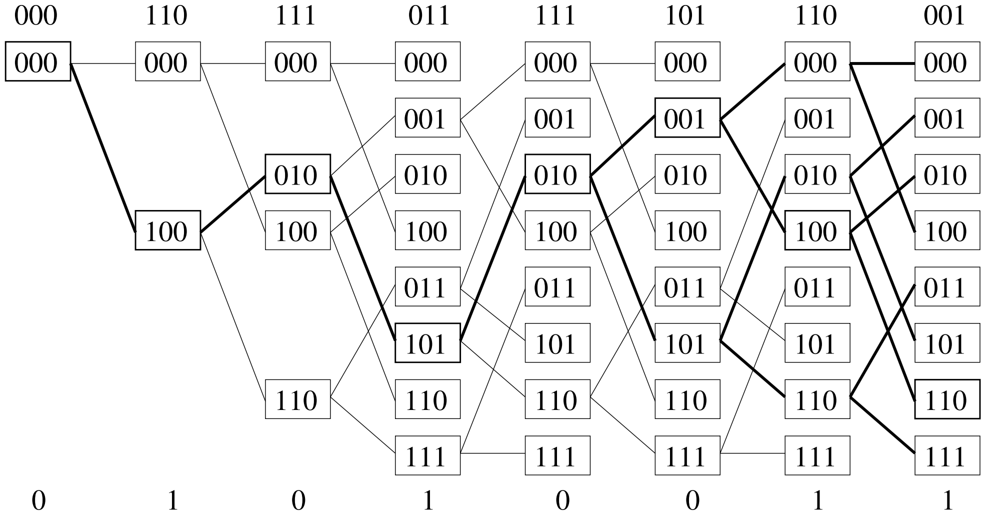


Figure 2.12: All paths from the last column of the trellis

Merging depth

In order to be able to successfully decode convolutional codes, we need to make one further observation. In Figure 2.12 is a trellis from Section 2.3. The figure only shows the most probable paths from all states. The most probable paths leaving the states in the last column (the actual state, when we are processing the sequence), are depicted in bold. Note that all paths coalesce in the fourth last column. Therefore, regardless of the state we begin the back-tracing in, the prefix of the most probable state path will be the same (01010 in this example).

If we are able to determine the depth D, in which all the paths have merged with high probability, we can start the decoding in any state, and after D steps output the last state which should be identical for all paths. Then we can truncate the last column of values $\gamma_i(j)$ and $\delta_i(j)$. It is necessary to find such D that the probability of the paths not coalescing in less than D steps is negligible compared to the probability of incorrect decoding caused by erroneous transmission. We will call D the *merging depth.* The actual depth in which we will truncate the memory is called *truncation length* T.

The minimum number of changes that need to be done to transform a correct encoded sequence into a different correct encoded sequence is called a code *free distance* d_{free}. For the code used as an example in Section 2.3, $d_{\text{free}} = 6$. For an example of two correct sequences differing in six characters we can take the all-zero path $000 - 000 - 000 - 000 - 000 - 000$ corresponding to the input sequence 000000 and the path $000 - 100 - 110 - 011 - 001 - 000$ corresponding to the input sequence 011000 (see Figure 2.9). The corresponding code words are $000 - 000 - 000 - 000 - 000 - 000$ and $000 - 110 - 001 - 010 - 101 - 000$ (see Figure 2.8), differing in six bits. Note that since the transition probabilities are symmetric, we could have chosen any path instead of the all-zeroes path. When no truncation is used a convolutional code can always repair at least $\lfloor d_{\text{free}}/2 \rfloor$ errors. For longer sequences, multiple non-adjacent groups of at most $\lfloor d_{\text{free}}/2 \rfloor$ errors can usually be corrected.

If we have an algorithm with truncation length T, we will call the minimum hamming distance between two paths that do not merge in T steps the *truncation distance* d_T. In the previously mentioned example, the truncation distance d_7 for $T = 7$ is 7, as the minimum weight path that does not merge with the all-zeros path is $000 - 100 - 110 - 111 - 111 - 111 - 111$. The output (emission) sequence $000 - 110 - 001 - 100 - 100 - 100 - 100$ has hamming distance of seven to the all-zero-path all-zero-output sequence.

It was shown in [8] that a truncation length of $5k$ is enough for the loss from truncation to be comparable to the loss caused by errors in transmitted data. Later more precise results on a variety of channels have shown that the truncation length needs to be chosen so that $d_{\text{free}} < d_T$ [12], [16].

We say that models, for which the paths merge with high probability after T steps, satisfy the *merging property.* The path suffix that is common for all paths starting from current column is called *survivor path.* The memory, where all paths are stored is usually called *survivor memory.*

There are two basic methods for implementing Viterbi decoders: register exchange (RE) algorithms and trace-back algorithms. Most of Viterbi decoder implementations use a modification of one of these algorithms. All of these algorithms exploit the property of convolutional code models that each state (except the first and last one) has two possible predecessor and two possible successor states. This reduces the complexity of implementations of convolutional code decoders dramatically.

Register exchange algorithms

In register exchange algorithms, the survivor memory has $m \times (T + 1)$ registers, where T is the size of survivor memory and should be larger than the merging depth D. The register connections are reversed compared to the connections in the trellis. The i-th row of registers contains the survivor path that begins in state i. When we process a symbol of the observation sequence, we need to update the survivor path for each row of registers. As mentioned earlier, due to a special structure of the trellis for convolutional codes, the current value of $\gamma_i(j)$ is derived from one of only two possible previous values, see Figure 2.13. If the value $\gamma_i(j)$ is derived from $\gamma_{i-1}(k)$, we need to copy the path in the row k to the row j, shifted one register to the left, so that we don't overwrite the current state. This is always possible, as registers (i, j) and $(i + 1, k)$ are always connected due to the reversed connections (Figure 2.14). We can therefore update the columns of registers from right to left. The first column will be set to the input that corresponds to the actual state.

If the merging property holds, and $T > D$, all paths will coalesce and the values in the registers in the left-most column should all hold the same value. Thus, we can output any element from the left-most column as the next decoded bit.

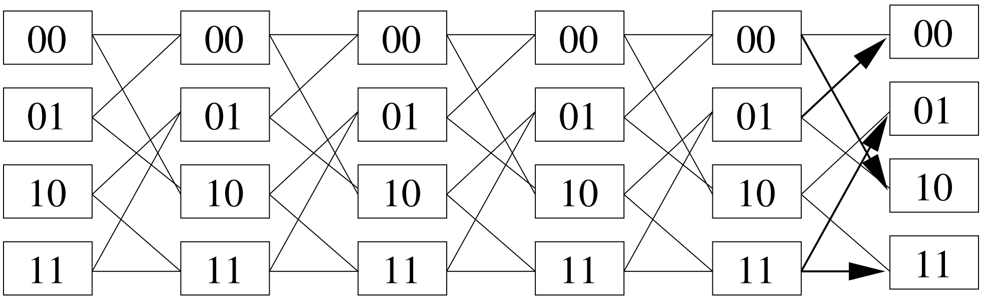


Figure 2.13: A trellis for code with $k = 2$. transitions used to derive last column are depicted in bold.

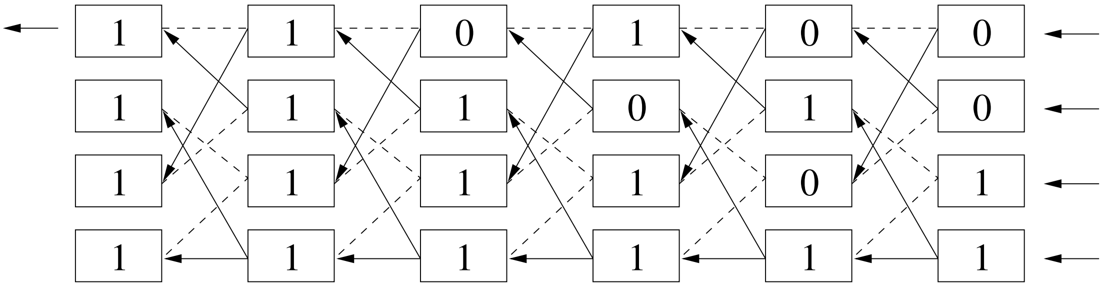


Figure 2.14: Survivor memory for trellis in Figure 2.13. Arrows show transitions that will be used to update the state paths in each row.

Trace-back algorithms

Register exchange methods are fast, but have large hardware complexity. Another method is a trace-back method, very similar to the standard Viterbi algorithm, described in the beginning of this chapter. A simple implementation would have $m \times T + 1$ registers, containing back pointers (also called decision values). The $T + 1$ registers in each row are linked. In each iteration, we shift all values to the left and we put new values to the right-most column of registers (dashed arrows in Figure 2.15). Then we start a back-trace from an arbitrary register, and the last result of the back-trace is returned. Again, given the merging property, we expect that all paths coalesce in less than T steps and the result will not depend on the register where we started the trace-back.

This method is obviously not very practical as each trace-back operation traces through T states in order to output a single bit. A number of improved practical algorithms involve increasing the memory size and running the trace-back less often, outputting bigger chunks of data at once [17].

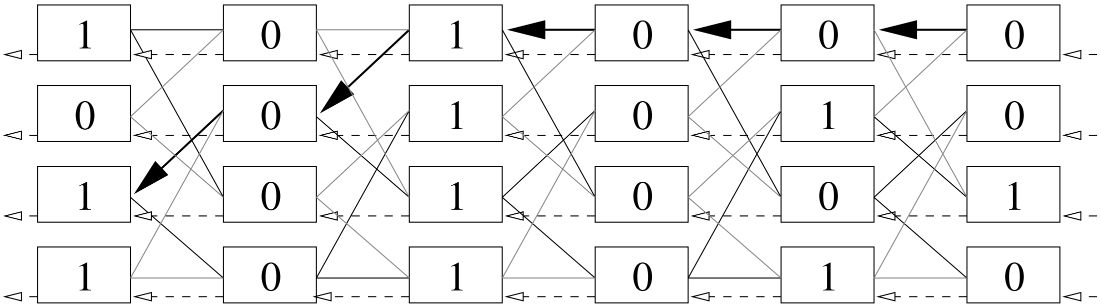


Figure 2.15: Tracing back to obtain one bit in the most probable state path

Chapter 3

The On-line Viterbi Algorithm

All algorithms for decoding of hidden Markov models shown in the previous chapter are either on-line (Definition 8) or correct (Definition 9), but never both. In almost all applications, having both properties would be desirable or beneficial. Here we propose an algorithm, that has the on-line property of algorithms associated with convolutional code decoding, while preserving correctness by using a memory of dynamic size.

As we have shown before, if there exists a coalescence point where all paths merge, we can output the initial segment of the most probable path up to that coalescence point. In this section, we show how to detect the existence and position of such coalescence point without adding significant overhead to the original Viterbi algorithm.

We can represent δ, the matrix of back pointers from the original Viterbi algorithm, as a tree of back pointers (Figure 3.1). We will store a compressed version of this back pointer tree in the memory. The nodes of the tree, each corresponding to a matrix element $\gamma_i(j)$, will contain the number of its children, its parent node, and the numbers i,j. In order to compress the tree, we will omit all leaves, except those in the last column. This will leave out paths that cannot be part of the most probable state path. We will further compress all internal nodes that have exactly one child. Paths containing only such nodes will be contracted to a single edge. The node of the tree corresponding to the coalescence point will have at least two children, as at least two paths will merge in it. It will therefore be left in the compressed tree (Figure 3.2). The compressed tree will have m leaves, one for each of the m last states and at most $m - 1$ internal nodes. We will store the tree in a linked list, ordered by the position of the node in the sequence (Figure 3.3). We also keep a list of pointers to the leaves of the tree.

Whenever we read an input symbol we update the tree as follows. First, we add the m new leaves, each having a parent in one of the former leaves. We append these new leaves to the end of the linked list. Then we proceed with the compression of the new tree. We process the linked list of nodes from the end. If a node has zero children, we remove it from the list and decrease the child count of its parent node. If it is the new leaf or an internal node, we check if its parent has more than one child. If not, we erase it and update the parent pointer to the parent's parent. We repeat this step, until the original node's parent has more than one child.

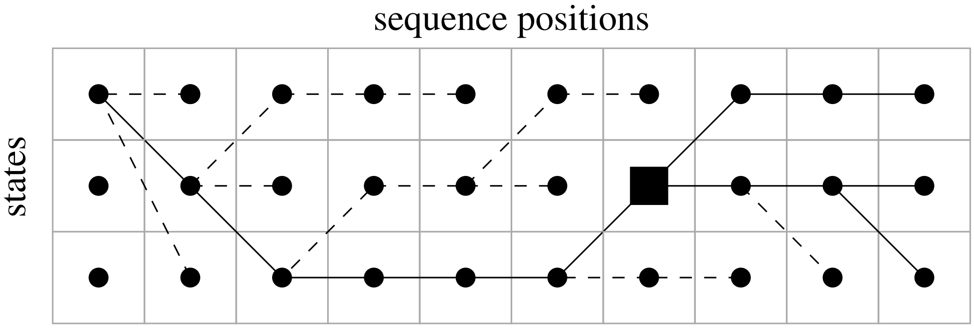


Figure 3.1: Tree of back pointers. Dashed edges cannot be part of the most probable state path. The square node marks the coalescence point.

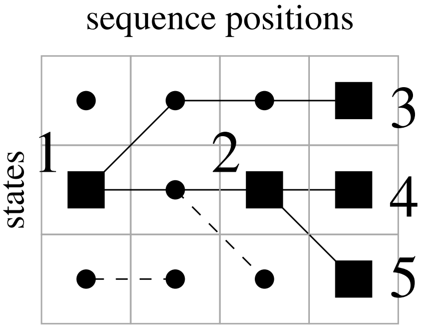


Figure 3.2: Compressed tree. Nodes present in the compressed representation are labeled and marked by squares.

If the root node of the tree is changed by the compression, we begin trace-back from $\gamma_i j$, the entry in the matrix γ corresponding to the new root node, erasing the columns of the matrices γ and δ on the way. When we finish reading the sequence, we need to start a regular trace-back from the maximum in the right-most column to output the suffix of the sequence for which no coalescence point has been found. The pseudo-code for the algorithm is shown in Figure 3.6, with auxiliary functions for tree compression and trace-back shown in Figures 3.4 and 3.5 respectively.

3.1 Running time analysis

Theorem 7 *Given a hidden Markov model M and observed sequence X, the on-line Viterbi algorithm finds the most probable state path in $\Theta(n|E|)$ time, where $|E|$ is the number of non-zero transitions.*

Proof. The running time of the original Viterbi algorithm is $O(n|E|)$ (Theorem 3). Compared to the Viterbi algorithm, we add the tree compression routine at each iteration. On every call of the compression routine each list and parent pointer will be followed at most

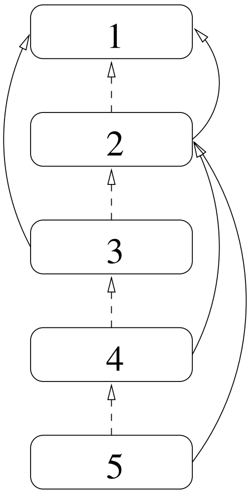


Figure 3.3: Compressed tree in linked list representation. Full lines are parent pointers.

once. There are at most $3m - 1$ nodes in the tree before compression, at most $m - 1$ internal nodes, m previous leaves, and m current leaves. Therefore the tree has at most $3m - 2$ list and parent pointers. The tree compression will therefore run in $O(m)$ time. Trace-back of both algorithms traces through n symbols, only in different order.

The total running time will be $\Theta(n(|E| + O(m))) = \Theta(n(|E|))$ because $m = O(|E|)$.

□

3.2 Space analysis

The space analysis will not be as straight forward as in the case of the previous algorithms. Our new algorithm does not use a predetermined amount of memory because we do not know the location of the coalescence points. We will show, that in the best case, we will only use a constant amount of memory, and in the worst case, our space complexity will be $\Theta(nm)$, same as the Viterbi algorithm.

3.2.1 Best case space analysis

Theorem 8 *There exists a hidden Markov model M such that given an arbitrary observed sequence X, the on-line Viterbi algorithm will find the most probable state path while using only $O(m)$ memory.*

Proof. Consider a hidden Markov model $M = \{S, \Sigma, T, E, \Pi\}$, where $S = \{S_1, S_2\}$, $\Sigma = \{0, 1\}$, $t_i(j) = 0.5$ for i and j in $\{0, 1\}$, $e_i(j) = e$ for $i = j$ and $e_i(j) = 1 - e$ otherwise, for some $e \in (0.5, 1)$ (Figure 3.7), $\Pi = \{0, 1\}$. The two states generate symbols with different probabilities, but the transition probabilities are equal.

```
Compress($L$)
while $L.next \neq NULL$ do
    if $L.children < 1$ then
        dec($(L.parent).children$)
        $T = L.next$
        delete $L$
        $L = T$
    else
        while $(L.parent).children = 1$ do
            $T = (L.parent).parent$
            delete $L.parent$
            $L.parent = T$
        end while
    end if
    $L = L.next$
end while
return $L$
```

Figure 3.4: Tree compression function

```
Trace-back($\delta$, $i$, $s_k$)
while $\delta_i$ do
    dec($k$)
    $s_k = \delta_i(s_{k+1})$
    dec($i$)
end while
return $s$
```

Figure 3.5: Trace-back function

Let's examine one step of the algorithm at time t for this model. With the transition probabilities $t_i(j)$ equal, both states $\gamma_t(0)$ and $\gamma_t(1)$ will be derived from the state with higher γ_{t-1} value, which will thus be a coalescence point. For example, if $\gamma_{t-1}(0)$ is the maximum probability state in time $t-1$ and $X_t = 0$ then $\gamma_t(0) = \gamma_{t-1}(0)t_0(0)e_0(0) = 0.5\gamma_{t-1}(0)e$ and $\gamma_t(1) = \gamma_{t-1}(0)t_0(1)e_1(0) = 0.5\gamma_{t-1}(0)(1-e)$. Since $e > 1-e$, one of the new states will be more probable leading to a new coalescence point when we process X_{t+1}. Given the initial distribution $\Pi = \{0, 1\}$ there will be a single maximum γ matrix element in every column of the matrix γ which will also be the coalescence point when we process the next column. We will therefore never need to store more than two columns of the matrix. Additionally, the resulting state path will be equivalent to the input sequence, as the coalescence points always occur in the states that have maximum probability in their column.

□

```
On-line_Viterbi($M$, $X$)
for $i = 1$ to $m$ do {Initialization}
    $\gamma_0(i) = \pi_j$
    $\delta_0(i) = 0$
    $P_i$ = new node
    $P_i.i = 0$
    $P_i.j = i$
    $L.Push(P_i)$
end for
for $j = 1$ to $n$ do
    for $i = 1$ to $m$ do
        $\gamma_j(i) = \max_{k=1}^{m} \gamma_{j-1}(k)t_k(i)e_i(X_j)$
        $\delta_j(i) = \arg\max_{k=1}^{m} \gamma_{j-1}(k)t_k(i)e_i(X_j)$
        $N_i$ = new node {Add new leaf nodes}
        $N_i.i = j$
        $N_i.j = i$
        $N_i.parent = P_{\delta_j(i)}$
        $L.Push(N_i)$
    end for
    Compress(L)
    if $L.root\_changed$ then
        Partial_output(Trace-back($\delta$, $(L.root).i$, $(L.root).j$))
        $P = N$
    end if
end for
$s_n = \arg\max_{k=1}^{m} \gamma_n(k)$
Partial_output(Trace-back($\delta$, $n$, $s_n$)) {Trace back the last part}
return
```

Figure 3.6: On-line Viterbi algorithm

3.2.2 Worst case space analysis

Theorem 9 *There exists a hidden Markov model M such that given an arbitrary observed sequence X, the on-line Viterbi algorithm will find the most probable state path while using $O(nm)$ memory.*

Proof. We can modify the two state HMM used in the previous proof. The new model is $M' = \{S, \Sigma, T', E', \Pi'\}$, where $\Pi' = \{0.5, 0.5\}$, $t_i(j) > 0.5$ for $i = j$ and $e_i(j) = 0.5$ for all i,j. For arbitrary input sequence X there will be no coalescence point in the matrix γ, because the states are indistinguishable with respect to the input sequence X. Transitions between states 0 and 1 and vice versa will never happen, because the values of $\gamma_i(0)$ and $\gamma_i(1)$ will remain equal and the transition probabilities between different states are lower than between the same states. We will need to finish reading the sequence in order to begin

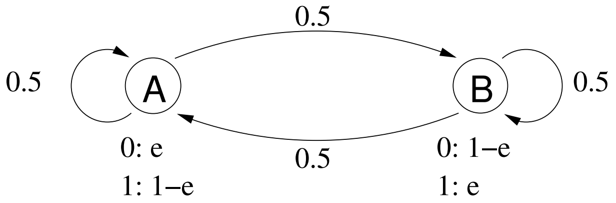


Figure 3.7: Two state model with constant memory requirements.

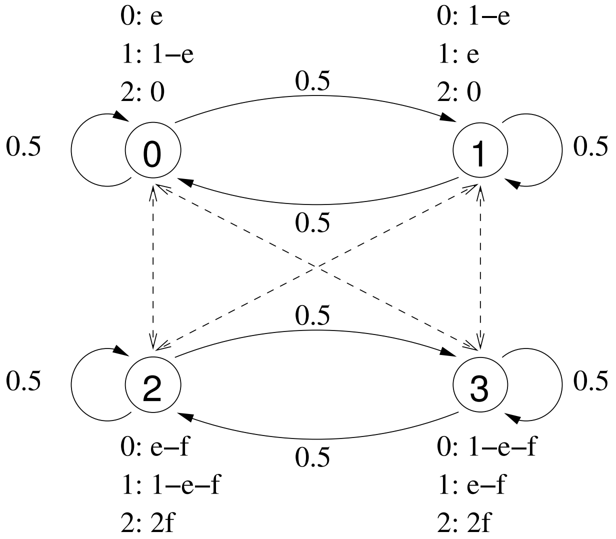


Figure 3.8: Model on which any correct decoding algorithm will have at least $O(n)$ time complexity. Zero probability transitions are depicted by dashed lines.

the trace-back; the algorithm behaves as the Viterbi algorithm, using $\Theta(mn)$ memory.

□

Although this seems as an unsatisfactory property of the on-line Viterbi algorithm, we will show that it is not possible to find the most probable state path with $o(n)$ memory with any correct algorithm.

One category of algorithms stores X in memory and accesses it multiple times, having even best-case memory complexity of $\omega(n)$. For example, the checkpointing-Viterbi as well as some versions of the original Viterbi algorithm fall into this category. We will use a simple example to show, that any algorithm must use at least $\omega(n)$ memory in some cases.

Theorem 10 *For any algorithm for hidden Markov model decoding exists a HMM M and observed sequence X, such that the algorithm will require $\omega(n)$ memory to find the most probable state path.*

Proof. We will consider a modification of the simple model used in the proof of Theorem 8. The model (see Figure 3.8) will have four states and will be over three letter alphabet,

$\Sigma = \{0, 1, 2\}$. The model will have two pairs of states, $(0, 1)$ and $(2, 3)$. Initial probability distribution is uniform, $\Pi = \{0.25, 0.25, 0.25, 0.25\}$. All transition probabilities between the states of each pair are 0.5, transition probabilities between states from different pairs are 0. Emission probabilities for states 0 and 1 and symbols 0 and 1 are $e_i(j) = e$ for $i = j$ and $e_i(j) = 1 - e$ otherwise, $e > 0.5$. For states 0 and 1 and symbol 2, $e_i(2) = 0$. Emission probabilities for states 2 and 3 are $e_i(j) = e - f$ for $i - 2 = j$ and $e_i(j) = 1 - e - f$ otherwise for symbols 0 and 1, $f < (1 - e)/2$, and emission probabilities for symbol 2 are $e_i(2) = 2f$.

The most probable paths beginning in both states of either pair converge in one step, by the same argument as in the proof of Theorem 8. While only symbols 0 and 1 appear in the sequence, the most probable path in states $\{0, 1\}$ will be more probable than the one in $\{2, 3\}$, since the transition probabilities are the same and the corresponding emission probabilities are higher in states $\{0, 1\}$. If the sequence ends without the symbol 2, this most probable state path will be the result. However if symbol 2 occurs, the emission probabilities of $e_i(2)$ in states $\{0, 1\}$ are 0, this state path will be therefore impossible, and the state path from the states $\{2, 3\}$ will be the correct solution. In order to decide, which state path to output, we need to wait until either the symbol 2 occurs, or we need to read the whole sequence. We need at least $\omega(n)$ memory to remember either the sequence X or some description of it. If this was not the case, we could code sequences from $\{0, 1\}^n$ using the description of the algorithm, the HMM, and the memory the algorithm is using at the point when it finishes reading the sequence, but before it outputs the correct result. By the argument from the proof of Theorem 8, the result is exactly the same as the input sequence when there is no character 2 in the input sequence. The total space required for such representation of arbitrary sequence of length n is $o(n) + O(m) + O(1) = o(n)$ given that $n \gg m$. But there are less than 2^n sequences of length shorter than n over the alphabet $\{0, 1\}$. At least two different sequences must be represented by a single shorter sequence, which is a contradiction.

□

We have shown special cases, where the algorithm requires very large or very small memory. The worst case was somehow artificial and we would be hard-pressed to find an application in which we would use such an HMM. Interesting question is, how will the algorithm perform on average on a random observation sequence. Of special interest is expected maximum amount of memory used while decoding a sequence of length n. Knowing this value would enable us to determine the amount of memory necessary to decode a sequence of known length. In the next chapter, we will prove the expected maximum memory to be $\Theta(m \log(n))$ for two-state HMMs. Then we will show experimental results for larger and more complex HMMs, which show that our algorithm almost always satisfies the on-line property implied by its name.

Chapter 4

Two-state hidden Markov models

We will divide the problem of expected worst case memory complexity for two-state HMMs into two main categories. Symmetrical HMMs and non-symmetrical, or general, HMMs. Further, the input sequence can be independently and identically distributed or generated by a HMM. We will first discuss symmetrical models and i.i.d. sequences. We will present a proof for the symmetrical, i.i.d. case and then extend it for general models on i.i.d. or HMM generated sequences.

4.1 Symmetric models

We call a two-state HMM M over two letter alphabet symmetric if $M = \{S, \Sigma, T, E, \Pi\}$ and $t_i(j) = t$ for $i \neq j$ and $t_i(j) = 1 - t$ for $i = j$, $e_i(j) = e$ for $i = j$ and $e_i(j) = 1 - e$ for $i \neq j$ (Figure 4.1a). We will consider the situation with $t < 0.5$ first. The back pointers in matrix λ between sequence positions i and $i + 1$ can form one of configurations i–iii in Figure 4.1b. Note that configuration iv cannot happen, since $t < 0.5$. Coalescence point occurs when either of the configurations ii or iii happens. Maximal memory used by HMM is proportional to the maximum length of continuous sequence of configurations i. We will call such a sequence of configurations a *run*.

Note that the algorithm is actually adding logarithmic probabilities, as mentioned in Viterbi algorithm description on page 10. The probabilities will be denoted $\lambda_i(j) = \log \gamma_i(j)$ as before. The recurrence used in the Viterbi algorithm (line 7 in algorithm in Figure 2.4) implies that the configuration i occurs when $\lambda_i(0) + \log(1 - t) \geq \lambda_i(1) + \log t$ and $\lambda_i(1) + \log(1 - t) \geq \lambda_i(0) + \log t$, thus when the difference of $\lambda_i(0) - \lambda_i(1)$ lies between $\log t - \log(1 - t)$ and $\log(1 - t) - \log t$. Configuration ii occurs, when $\lambda_i(0) - \lambda_i(1) > \log(1 - t) - \log t$, configuration iii when $\lambda_i(0) - \lambda_i(1) < \log t - \log(1 - t)$.

We will be interested in the value of $\lambda_i(0) - \lambda_i(1)$ during one run. The value will be initially $\log(1 - t) - \log t \pm (\log(1 - e) - \log e)$ for configuration ii and $\log t - \log(1 - t) \pm (\log(1 - e) - \log e$ for configuration iii. Then, each step it will be changed by $\log e - \log(1 - e)$ in either direction, depending on the input symbol X_i. We can define an integer random variable $Y = \frac{\lambda_i(0) - \lambda_i(1)}{\log(1-e) - \log e} - (\log t - \log(1 - t))$, that will correspond to the value of

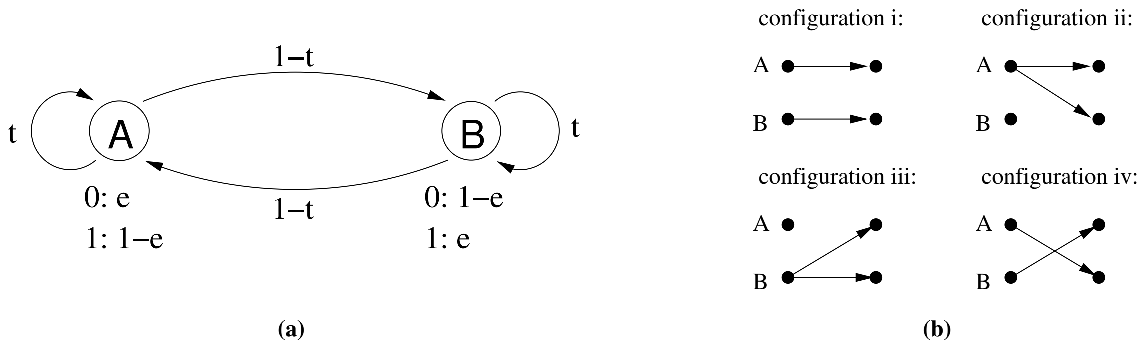


Figure 4.1: (a) Symmetric two-state HMM. (b) Possible back pointer configurations for two state HMM.

$\lambda_i(0) - \lambda_i(1)$. X will be changed by $+1$ or -1 each step, until leaving the interval $(0, K)$, where $K = \left\lceil 2\frac{\log(1-t) - \log t}{\log(1-e) - \log e} \right\rceil$.

First, we consider the input sequence X to be a sequence of uniform, independent and identically distributed binary random variables. In this case, the sequence of values of Y will describe a random walk on integers, a well understood problem. This random walk starts at value $+1$ and ends by reaching either zero or K. In case that zero is reached, we have a coalescence point in configuration iii and Y is initialized to value $+1$, which corresponds to initial value $\log t - \log(1-t) + (\log(1-e) - \log e)$ for $\lambda_i(0) - \lambda_i(1)$, or to -1, which causes the following configuration to be a new type iii coalescence point and corresponds to the initial value $\log t - \log(1-t) - (\log(1-e) - \log e)$. The choice of initialization value depends on the symbol X_i of the input sequence. In the case, where the random walk is ended by reaching K, the situation is symmetric, with the random variable $Y' = -Y$.

We can now apply a classical result from the theory of random walks.

Theorem 11 *Given $0 \leq z \leq a$ and $a, z \in \mathbb{N}$, the expected time before a random walk on integers that begins in z leaves interval $(0, a)$ is $z(a - z)$.*

Proof. See [6, ch.14.3]

We solve our problem by substituting $a = K$ and $z = 1$.

Corollary 1 *Assuming the input sequence is uniformly i.i.d., the expected length of a run of a symmetrical two-state HMM is $K - 1$.*

Therefore the larger is K, the more memory is required to decode the HMM. The worst case is achieved as e approaches 0.5. In such case, the two states are indistinguishable, as in the example used to prove theorem 9. Using the theory of random walks, we can also characterize the distribution of length of runs.

Theorem 12 *Let W be a random variable that realizes a random walk on integers, let p be the probability of changing W by $+1$ and $1 - p$ the probability of changing W by -1.*

The probability of $W_{z,n}$, the event that the random walk beginning at z and never reaching K reaches 0 in n steps, is

$$\Pr(W_{z,n}) = a^{-1}2^{n+1}p^{(n-z)/2}(1-p)^{(n+z)/2} \sum_{0<v<a/2} \cos^{n-1}\frac{\pi v}{a} \sin\frac{\pi v}{a} \sin\frac{\pi z v}{a} \tag{4.1}$$

for $n-z$ even and $n>1$, $W_{z,n}=0$ otherwise.

Proof. See [6, ch.14.5])

Lemma 1 *Let R_ℓ be the event that the length of a run of symmetrical two-state HMM is either $2\ell+1$ or $2\ell+2$. Then, assuming that the input sequence is uniformly i.i.d., following holds for some constants $b, c > 0$:*

$$b \cdot \cos^{2\ell}\frac{\pi}{K} \leq \Pr(R_\ell) \leq c \cdot \cos^{2\ell}\frac{\pi}{K} \tag{4.2}$$

Proof. In the case K is odd, the walk can either reach 0 in odd number of steps or K in even number of steps. Substituting $a = K$, $z = 1$,$p = 1/2$ to (4.1) yields the probability $\Pr(W_{1,n})$ of the random walk on interval $(0, K)$ beginning at 1 and first ending at 0 in $2\ell+1$ steps.

$$\Pr(W_{1,2\ell+1}) = \frac{2}{K} \sum_{0<v<K/2} \cos^{2\ell}\frac{\pi v}{K} \sin^2\frac{\pi v}{K} \tag{4.3}$$

As $p = 1/2$, the walk ending in K is symmetric of even length and the probability of it is equal to the probability $\Pr(W_{K-1,2\ell+2})$, of a walk beginning in $K-1$ and first ending in 0.

$$\Pr(W_{K-1,2\ell+2}) = \frac{2}{K} \sum_{0<v<K/2} \cos^{2\ell+1}\frac{\pi v}{K} \sin\frac{\pi v}{K} \sin\frac{\pi v(K-1)}{K} \tag{4.4}$$

The probability R_ℓ will therefore be

$$\begin{aligned} \Pr(R_\ell) &= \Pr(W_{1,2\ell+1}) + \Pr(W_{K-1,2\ell+2}) \\ &= \frac{2}{K} \sum_{0<v<K/2} \cos^{2\ell}\frac{\pi v}{K} \sin^2\frac{\pi v}{K} \left(1 + \frac{\sin\frac{\pi v(K-1)}{K}}{\sin\frac{\pi v}{K}} \cos\frac{\pi v}{K}\right) \\ &= \frac{2}{K} \sum_{0<v<K/2} \cos^{2\ell}\frac{\pi v}{K} \sin^2\frac{\pi v}{K} \left(1 + (-1)^{v+1}\cos\frac{\pi v}{K}\right) \end{aligned} \tag{4.5}$$

There are at most $K/2$ terms in the sum, and all can be bounded by $2\cos^{2\ell}\frac{\pi v}{K}$ from above. To get a lower bound, we can use the first term of the sum.

$$\frac{2}{K}\sin^2\frac{\pi}{K}\left(1+\cos\frac{\pi}{K}\right)\cos^{2\ell}\frac{\pi}{K} \leq \Pr(R_\ell) \leq 2\cos^{2\ell}\frac{\pi}{K} \tag{4.6}$$

The case, where K is even is similar. The walk will reach either 0 or K in odd number of steps, therefore

$$\begin{aligned}
\Pr(R_\ell) &= \Pr(W_{1,2\ell+1}) + \Pr(W_{K-1,2\ell+1}) \\
&= \frac{2}{K} \sum_{0<v<K/2} \cos^{2\ell}\frac{\pi v}{K}\sin^2\frac{\pi v}{K}\left(1+(-1)^{v+1}\right) \qquad (4.7)\\
&= \frac{4}{K} \sum_{0<v<K/4,\ v \text{ odd}} \cos^{2\ell}\frac{\pi v}{K}\sin^2\frac{\pi v}{K}
\end{aligned}$$

At most $K/4$ summands can again be bounded from above by $\cos^{2\ell}\frac{\pi v}{K}$ and from below by using the first term of the sum. Therefore, for K odd

$$\frac{4}{K}\sin^2\frac{\pi}{K}\cos^{2\ell}\frac{\pi}{K} \le \Pr(R_\ell) \le \cos^{2\ell}\frac{\pi}{K} \qquad (4.8)$$

□

The previous lemma characterizes the length distribution of a single run. However, to analyze memory requirements for a sequence of length n, we need to consider maximum over several runs whose total length is n. Similar problem was studied for the runs of heads in a sequence of n coin tosses [11, 9]. For coin tosses, the length distribution of runs is geometric, while in our case the runs are only bounded by geometrically decaying functions. Still, we can prove that the expected length of the longest run grows logarithmically with the length of the sequence, as is the case for coin tosses.

Lemma 2 *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables drawn from a geometrically decaying distribution over positive integers, i.e. there exist constants a, b, c, $a \in (0,1)$, $0 < b \le c$, such that for all integers $k \ge 1$, $ba^k \le \Pr(X_i > k) \le ca^k$.*

Let N be the largest index such that $\sum_{i=1}^N X_i \le n$, and let Y_n be $\max\{X_1, X_2, \ldots, X_N, n - \sum_{i=1}^N X_i\}$. Then

$$E[Y_n] = \log_{1/a} n + o(\log n) \qquad (4.9)$$

Proof. Let $Z_n = \max_{i=1}^n X_i$ be the maximum of the first n runs. Clearly, $\Pr(Z_n \le k) = \Pr(X_i \le k)^n$. From the premise $ba^k \le \Pr(X_i > k) \le ca^k$ we get $(1 - ca^k) \le Pr(X_n \le k) \le (1 - ba^k)$ and thus $(1 - ca^k)^n \le \Pr(Z_n \le k) \le (1 - ba^k)^n$ for all integers $k \le \log_{1/a}(c)$

Lower bound: We will show that the expected value of Y_n is greater than $t_n = \log_{1/a} n - \sqrt{\ln n}$ for $n \to \infty$. If $Y_n \le t_n$, we need at last n/t_n runs to reach the sum n, i.e. $N \ge n/t_n - 1$ (discontinuing the last incomplete run). Therefore

$$\Pr(Y_n \le t_n) \le \Pr(Z_{\frac{n}{t_n}-1} \le t_n) \le (1 - ba^{t_n})^{\frac{n}{t_n}-1} = (1 - ba^{t_n})^{a^{-t_n}a^{t_n}(\frac{n}{t_n}-1)} \qquad (4.10)$$

Since $\lim_{n\to\infty} a^{t_n}(n/t_n - 1) = \infty$ and $\lim_{x\to 0}(1 - bx)^{1/x} = e^{-b}$, we get $\lim_{n\to\infty} \Pr(Y_n \le t_n) = 0$.

For the expected value of a random variable holds:

$$\begin{aligned} E[Y_n] &= \sum_{i=1}^{\infty} i \Pr(Y_n = i) \geq \sum_{i=k}^{\infty} i \Pr(Y_n = i) \\ &\geq \sum_{i=k}^{\infty} k \Pr(Y_n = i) = k \Pr(Y_n > k) = k(1 - \Pr(Y_n \leq k)) \end{aligned} \tag{4.11}$$

Substituting t_n for k we get the desired bound:

$$E[Y_n] \geq t_n(1 - \Pr(Y_n \leq t_n)) \tag{4.12}$$

Upper bound: Clearly, $Y_n \leq Z_n$ and so $E[Y_n] \leq E[Z_n]$. Let Z'_n be the maximum of n i.i.d geometric random variables $X'_1, \ldots, X'_n$ such that $\Pr(X'_i \leq k) = 1 - a^k$.

We will compare $E[Z_n]$ to the expected value of variable Z'_n. Without loss of generality, $c \geq 1$. For any real $x \geq \log_{1/a}(c) + 1$ we have:

$$\begin{aligned} \Pr(Z_n \leq x) &\geq (1 - ca^{\lfloor x \rfloor})^n \\ &= (1 - a^{\lfloor x \rfloor - \log_{1/a}(c)})^n \\ &\leq (1 - a^{x - \log_{1/a}(c) - 1})^n \\ &= \Pr(Z'_n \leq x - \log_{1/a}(c) - 1) \\ &= \Pr(Z'_n + \log_{1/a}(c) + 1 \leq x) \end{aligned} \tag{4.13}$$

This inequality holds even for $x < \log_{1/a}(c)+1$, since the right hand side is equal to $\Pr(Z'_n < 0)$ and thus 0 in such case. Therefore, $E[Z_n] \leq E[Z'_n + \log_{1/a}(c) + 1] = E[Z'_n] + O(1)$.

In [19], authors proved that the maximum of n independent random variables with geometric distribution $\Pr(X'_i \leq k) = 1 - a^k$ (specifically runs of coin tosses, until head falls) is $\log_{1/a}(n) + o(\log n)$, which proves the upper bound for $E[Z_n]$ and thus $E[Y_n]$.

□

We are now prepared to prove the theorem about the expected memory required to process a uniform i.i.d. sequence using symmetric two-state HMMs.

Theorem 13 *Given a uniform i.i.d. sequence of length n over binary alphabet and a symmetric two-state hidden Markov model M, the expected maximum memory used to process the sequence using on-line Viterbi algorithm is less than $(2K^2/\pi^2)\ln n$, where $K = \lceil 2\frac{\log(1-t)-\log t}{\log(1-e)-\log e} \rceil$*

Proof. In the case $t < 0.5$, we get from lemma 1:

$$\begin{aligned}
&b \cdot \cos^{2\ell}\frac{\pi}{K} \leq \Pr(R_\ell) \leq c \cdot \cos^{2\ell}\frac{\pi}{K} \\
&\sum_{i=\ell}^{\infty} b \cdot \cos^{2\ell}\frac{\pi}{K} \leq \sum_{i=\ell}^{\infty}\Pr(R_\ell) \leq \sum_{i=\ell}^{\infty} c \cdot \cos^{2\ell}\frac{\pi}{K} \\
&b\sum_{i=\ell}^{\infty}\cos^{2\ell}\frac{\pi}{K} \leq \Pr(X_i > 2\ell) \leq c\sum_{i=\ell}^{\infty}\cos^{2\ell}\frac{\pi}{K} \\
&b\frac{\cos^{2\ell}\frac{\pi}{K}}{1-\cos^2\frac{\pi}{K}} \leq \Pr(X_i > 2\ell) \leq c\frac{\cos^{2\ell}\frac{\pi}{K}}{1-\cos^2\frac{\pi}{K}} \\
&b'\cos^{2\ell}\frac{\pi}{K} \leq \Pr(X_i > 2\ell) \leq c'\cos^{2\ell}\frac{\pi}{K}
\end{aligned} \tag{4.14}$$

From lemma 2 we get

$$E[Y_n] = \log_{1/cos\frac{\pi}{K}} n + o(\log n) = \frac{1}{\ln\left(1/\cos\frac{\pi}{K}\right)}\ln n + o(\log n) \tag{4.15}$$

Now, using the MacLaurin expansion of cos and ln as K grows to infinity, we get:

$$\begin{aligned}
\frac{1}{\ln\left(1/\cos\frac{\pi}{K}\right)}\ln n + o(\log n) &= \frac{1}{-\ln\left(\cos\frac{\pi}{K}\right)}\ln n + o(\log n) \\
&= \frac{1}{-\ln(1-\frac{\pi^2}{2K}+O(\frac{\pi^4}{K}))}\ln n + o(\log n) \\
&= \frac{1}{\frac{\pi^2}{2K^2}+O(\frac{\pi^4}{K})}\ln n + o(\log n) \\
&= (\frac{2K^2}{\pi^2}+O(1))\ln n + o(\log n) \\
&= \frac{2K^2}{\pi^2}\ln n + o(\log n)
\end{aligned} \tag{4.16}$$

The maximum memory grows approximately as $(2K^2/\pi^2)\ln n$.

Above, we supposed that $t < 0.5$. The case where $t = 0.5$ is the best case example in theorem 8, the maximum run length will be 0. Let's suppose $t > 0.5$. Configuration iv will extend the run instead of configuration i, which can not happen. The random walk will start in position 1 and the sequence of values Y will be updated as $Y_{t+1} = -Y_t \pm 1$. This corresponds to the update of the difference $\lambda_{i+1}(0) - \lambda_{i+1}(1) = \lambda_i(1) - \lambda_i(0) \pm (\log(1-e) - \log e)$. In the first step, the value Y will be either 0 or -2, thus terminating the random walk. The maximum run length will be 1.

□

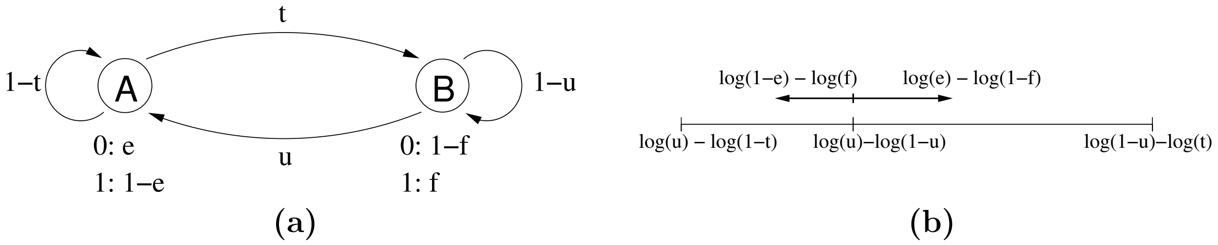


Figure 4.2: a) A general two-state model b) Random walk for configuration iii. and $t < u < 0.5$

4.2 General models

The situation is similar in the case of general two-state models. We also have four possible configurations as in the previous case (see Figure 4.1b). We will label the new transition and emission probabilities as in Figure 4.2a. Configuration i. happens while $\log u - \log(1-t) < \lambda_i(0) - \lambda_i(1) < \log(1-u) - \log t$, configurations ii. and iii. when $\lambda_i(0) - \lambda_i(1)$ is outside of this interval. The value $\lambda_i(0) - \lambda_i(1)$ updates by $\log e - \log(1-f)$ or $\log(1-e) - \log f$. These numbers are no longer equal in absolute value. The resulting random walk will be on real numbers, not integers. It will start in the points $\log(1-t) - \log t + \log e - \log(1-f)$ or $\log(1-t) - \log t + \log(1-e) - \log f$ in the case the previous run ended with configuration ii. and $\log u - \log(1-u) + \log e - \log(1-f)$ or $\log u - \log(1-u) + \log(1-e) - \log f$ in the case the previous run ended with configuration iii. . We will consider only those model parameters that do not result in trivial runs of length 0 or 1.

We are not aware of any results that would allow us to analyze such random walks directly; we will do upper bounds on their run lengths by transforming them to symmetric two-state models with longer run lengths.

In Theorem 13 we bounded $\Pr(X_i > 2\ell)$, the probability of a run in a symmetrical two-state model being longer than 2ℓ. This was the key step necessary for proving the expected maximum to be less than $2K^2/\pi^2 \ln n + o(\log n)$. Let us now consider $\Pr(X_i > 2\ell)$, the probability that a non-symmetrical random walk run is longer than 2ℓ. We need to show that this probability decays geometrically, as required by Lemma 2. Unless $\log e = \log 1 - f$ and $\log(1-e) = \log f$, which is the pathological worst-case situation from Theorem 9, we will have at least one step of non-zero size. The interval in which the random walk moves is of constant size, therefore there exists a number d, such that the random walk will leave the interval if d consecutive steps in the same direction are taken, regardless of the starting position before the d consecutive steps. We can bound the probability that d consecutive steps happen in a sequence of n steps using the following theorem:

Theorem 14 *Let $R_p(h,n)$ be the probability of a run of h or more consecutive heads appearing in a sequence of n independent coin tosses, p being the probability of obtaining head. Then*

$$\lim_{n\to\infty} 1 - R_p(h,n) = \beta_k \alpha_k^{-n-1} \tag{4.17}$$

where α_k is the smallest positive root of the function

$$1 - x + (1-p)p^k x^{k+1} = 0$$

and

$$\beta_k = \frac{1 - p\alpha_k}{(k+1-k\alpha_k)p}$$

Proof. See [6, pp. 322-325]

Setting $p = 1/2$, we can therefore reformulate Theorem 13 for general two-state hidden Markov models:

Theorem 15 *Given a uniform i.i.d. sequence of length n over binary alphabet and a two-state hidden Markov model M, the expected maximum memory used to process the sequence using on-line Viterbi algorithm is less than $O(\ln n)$.*

Proof.

Using the previous theorem, we can bound the probability of a run exceeding length ℓ:

$$\Pr(X_i > 2\ell) \le \beta_d \left(\frac{1}{\alpha_d}\right)^{\ell+1} \tag{4.18}$$

then using Lemma 2 with only the upper bound, we get the upper bound for expected maximum run length[1]:

$$E[Y_n] = \log_{\alpha_d} n + o(\log n) = \frac{1}{\ln \alpha_d} \ln n + o(\log n) \tag{4.19}$$

□

This theorem is applicable also in the case where the input sequence is arbitrary Bernoulli sequence, or is generated by a HMM which is capable of non-zero steps in the same direction in both states. We simply take the lower of the two probabilities for p and the upper bound will still hold.

[1] Trying to determine the constants as in the symmetric case is not sensible due to the extremely generous upper bound

Chapter 5

Multi-state hidden Markov models

Analysis technique used in previous chapter to show the expected maximum memory used can not be easily extended to HMMs with more than two states. A coalescence point occurrence clears the memory in a two-state HMM but can leave a non-trivial back-pointer tree in HMMs with more states. The length of the next run depends on the tree left in memory, the consecutive runs are therefore no longer independent.

For multi-state HMMs, we did not succeed in proving results similar to the two-state HMMs. Experimental results however suggest that similar bounds could hold.

We have evaluated the performance of our algorithm on the problem of gene finding (see Section 2.2). The HMM used had 256 states of at most 4-th order. A simplified structure of the HMM is shown in Figure 5.1. The HMM was trained on RefSeq annotations of human chromosomes 1 and 22.

We have tested the algorithm on 20 MB long sequences: regions from the human genome, simulated sequences generated by the HMM, and i.i.d. sequences. Regions of the human genome were chosen from hg18[1] so that they do not contain sequencing gaps. Sequencing gaps, regions that have not been yet sequenced, almost always cause a coalescence point and the results would be biased if they were contained in the data. The distribution for the i.i.d. sequences mirrors the distribution of bases in human chromosome 1.

An example graph for current table length is in Figure 5.2a. Each spike in the graph corresponds to one run. The results for 20 MB sequences are shown in Figure 5.2b. The average maximum length of the table over several samples appears to grow faster than logarithmically with the length of the sequence at least for human genome and for data generated by HMM. We tried processing 50 longer 100 MB sequences generated by HMM, as there exist no continuous 100 MB sequences in hg18. The results are still unclear. The average table length divided by its logarithm increases with sequence length in Figure 5.4b, while the average table length divided by its square logarithm decreases after reaching maximum in Figure 5.4c. This suggests that the space complexity should be between $\Theta(log(n))$ and $\Theta(log^2(n))$ and therefore, poly-logarithmic.

We have seen in two-state symmetric HMMs in the previous chapter, that the expected

[1] Human genome, assembly 18: http://hgdownload.cse.ucsc.edu/goldenPath/hg18/

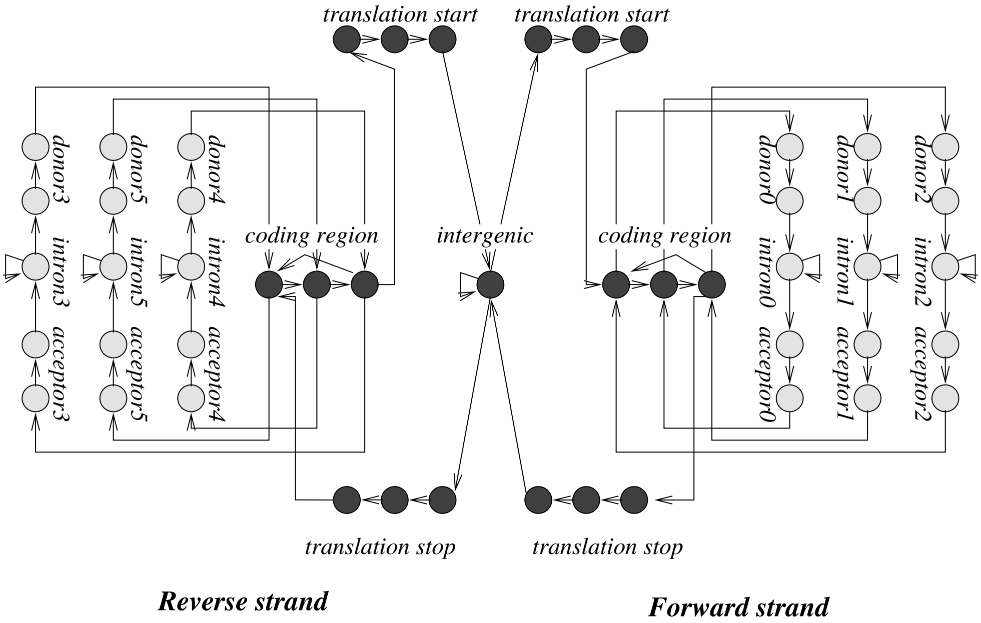


Figure 5.1: A simplified HMM for gene finding

run length increases with the difference between emission probabilities decreasing. The situation is somewhat similar in this case, there are three copies of the state for introns that have the same emission probabilities and the same self-transition probability. The more than logarithmic growth may therefore be an artifact that would disappear with longer sequences.

Nonetheless, we were able to process whole chromosomes at once without significant memory consumption. The maximum table length did not exceed $222,000$ on any chromosome. This is significant improvement over the Viterbi algorithm, which would keep the whole chromosome in memory. Indeed, the decoding of, for instance, the first human chromosome of 245 million bases using our model and the original Viterbi algorithm, would be impossible on common computational platforms at present time. Furthermore, as can be seen in Figure 5.2a, most of the time the program keeps only relatively short table; the average length on the 20 MB human genome segments is only $11,000$. The low average length can be of significant advantage if multiple processes share the same memory.

We have proved in Section 3.1 that the asymptotic running times of our algorithm and the Viterbi algorithm are equivalent. For gene finding, we experienced only 5% slowdown compared to the original Viterbi algorithm, before the original algorithm allocated all available memory (see Figure 5.3). This is to be expected for large models, because the cost of the tree maintenance in $O(m)$ time is significantly lower than the calculation of new values γ and δ in $O(|E|)$ time. The algorithm will therefore likely outperform any other known correct algorithms capable of decoding whole chromosomes or long sequences at once, given similar implementation.

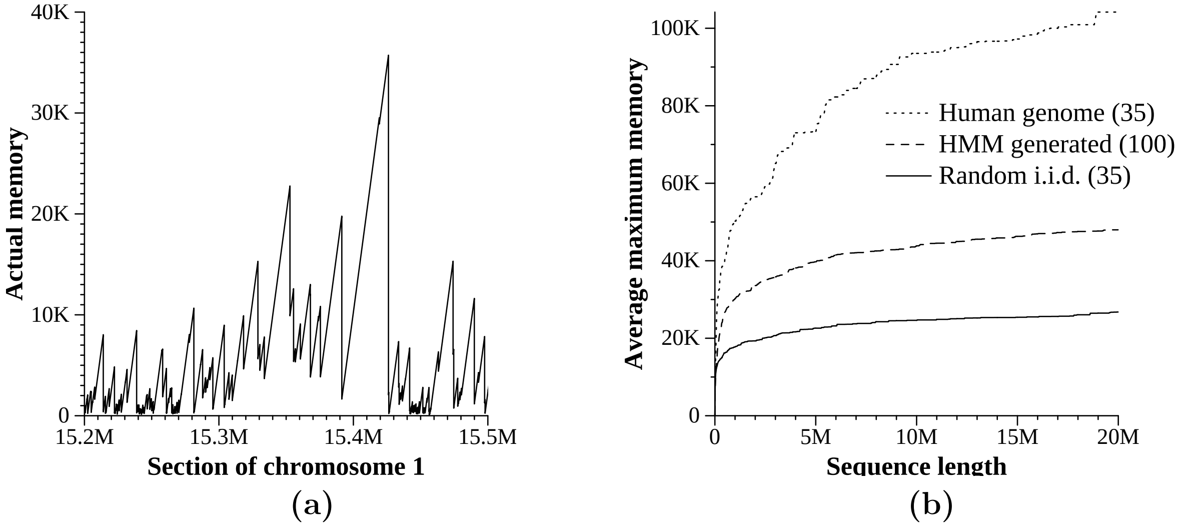


Figure 5.2: **Memory requirements of a gene finding HMM.** a) Actual length of table used on a segment of human chromosome 1. b) Average maximum table length needed for prefixes of 20 MB sequences.

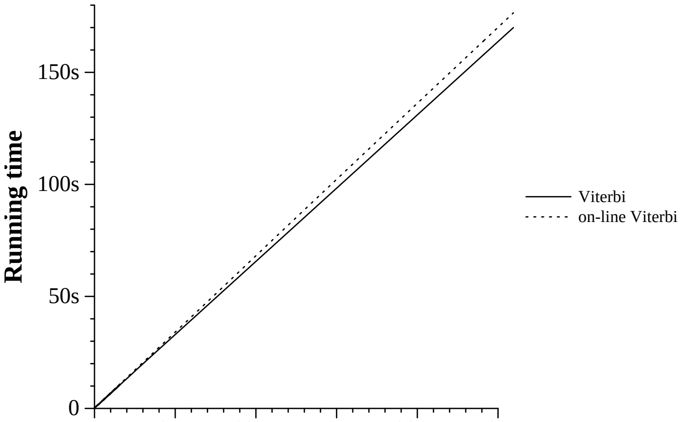
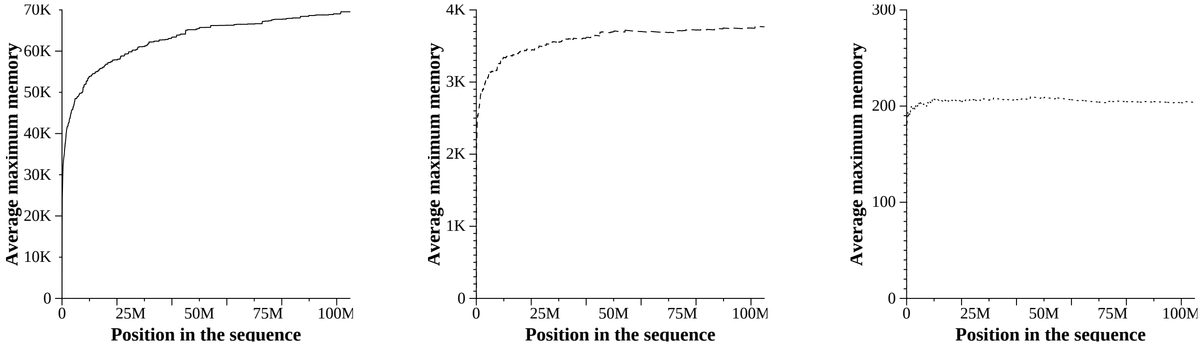


Figure 5.3: Comparison of running time of Viterbi and on-line Viterbi algorithms on a subsequence of the human genome.



(a) (b) (c)

Figure 5.4: **Memory requirements of a gene finding HMM.** a) Average maximum table length needed for prefixes of 100 MB sequences. b) Average maximum table length from a. divided by its logarithm c) Average maximum table length from a) divided by its squared logarithm

Chapter 6

Conclusion

In this thesis, we introduced new on-line Viterbi algorithm for hidden Markov model decoding. Our algorithm is based on efficient detection of coalescence points in the tree representing the state-paths under consideration of the dynamic programming algorithm. The algorithm requires variable space that depends on the HMM and on the local properties of the analyzed sequence. For two-state symmetric HMMs, we have shown that the expected maximum memory used for analysis of a sequence of length n is approximately $(2K^2/\pi^2)\ln n$, where K depends on the HMM. We implemented our algorithm and experiments on both simulated and real data and the results suggest that the asymptotic bound $\Theta(m \ln n)$ could also be extended to multi-state HMMs, and in fact, most of the time throughout the execution of the algorithm, much less memory is used. We have also shown, that no algorithm can have better worst case space complexity than our algorithm if it must always find the correct solution.

Our on-line Viterbi algorithm has multiple advantages over the previously known algorithms. It always returns the most probable state path, but does not need to read the whole sequence in order to start decoding. We demonstrated our algorithm on the problem of gene finding, and we found that our algorithm outperforms other decoding algorithms in this setting. The DNA sequences used as inputs to algorithms used for gene finding have tens of millions of symbols. Given a HMM with over 250 states, such as the one we used and similar to models used in real gene finders [2], we would need tens or even hundreds of gigabytes of memory to process such sequence using the Viterbi algorithm. Various modifications of the original algorithm can process the sequence with lower memory requirements, but add at least two-fold slowdown, which can be a considerable factor as the decoding of a sequence of such length takes hours on present computational platforms. Our algorithm tackles this problem flawlessly with only negligible slowdown compared to the Viterbi algorithm. Unlike the rest of the algorithms that always return correct solution, our algorithm can be used to process continuous streams of data, and could lead to improvement in such applications.

Future work and open problems

A more thoughtful analysis of multi-state HMMs could perhaps allow us to determine or at least estimate expected maximum memory used for concrete HMMs. Perhaps we could also be able to characterize the states that are likely to serve as coalescence points. This could provide yet another level of useful information about the problem being modeled.

Bibliography

[1] D. B. Rubin A. P. Dempster, N. M. Laird. Maximum likelihood from incomplete data via the *em* algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[2] Brona Brejova, Daniel G. Brown, Ming Li, and Tomas Vinar. ExonHunter: a comprehensive approach to gene finding. *Bioinformatics*, 21(S1):i57–65, 2005.

[3] Brona Brejova, Daniel G. Brown, and Tomas Vinar. The most probable labeling problem in HMMs and its application to bioinformatics. *Journal of Computer and System Sciences*, 2007. Early version in WABI 2004. Accepted November 2006.

[4] S. Karlin C. Burge. Prediction of complete gene structures in human genomic dna. *Journal of Molecular Biology*, (268):78–94, 1997.

[5] Piero Fariselli, Pier Luigi Martelli, and Rita Casadio. The posterior-viterbi: a new decoding algorithm for hidden markov models. 2005.

[6] W. Feller. *An Introduction to Probability Theory and Its Applications, Third Edition, Volume 1.* Wiley, 1968.

[7] G. David Forney Jr. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[8] G. David Forney Jr. Convolutional codes ii: Maximum likelihood decoding. *Information and control*, 25:222–266, 1974.

[9] L. Gordon, M. F. Schilling, and M. S. Waterman. An extreme value theory for long head runs. *Probability Theory and Related Fields*, 72:279–287, 1986.

[10] J. A. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *Computer Applications in the Biosciences*, 13(1):45–53, 1997.

[11] L. J. Guibas and A. M. Odlyzko. Long repetitive patterns in random sequences. *Probability Theory and Related Fields*, 53:241–262, 1980.

[12] F. Hemmati and Jr. Costello, D. Truncation error probability in Viterbi decoding. *IEEE Transactions on Communications*, 25(5):530–532, 1977.

[13] A. Krogh, B. Larsson, G. von Heijne, and E. L. Sonnhammer. Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes. *Journal of Molecular Biology*, 305(3):567–570, 2001.

[14] J. A. Eagon L. E. Baum. An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model of ecology. *Bulletin of the American Mathematical Society*, 73:360–363, 1967.

[15] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[16] I.M. Onyszchuk. Truncation length for Viterbi decoding. *IEEE Transactions on Communications*, 39(7):1023–1026, 1991.

[17] C. B. Shung R. Cypher. Generalized trace-back techniques for survivor memory management in viterbi algorithm. *Journal of VLSI Signal Processing*, 5:85–94, 1993.

[18] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[19] Eugene F. Schuster. On overwhelming numerical evidence in the settling of Kinney's waiting-time conjecture. *SIAM Journal on Scientific and Statistical Computing*, 6(4):977–982, 1985.

[20] Andrew J. Viterbi. Error bounds for convolutional codes and asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[21] Andrew J. Viterbi. Convolutional codes and their performance in communication systems. *IEEE Transactions on communication technology*, COM-19:751–772, 1971.

Abstrakt

Skryté Markovovské modely sú pravdepodobnostné modely ktoré sú úspešne používané pri riešení problémov týkajúcich sa napríklad bioinformatiky, samoopravujúcich kódov a rozpoznávania reči. V mnohých z týchto aplikácií sa používa Viterbiho algoritmus na hľadanie najpravdepodobnejšej anotácie sekvencií, ktoré môžu byť veľmi dlhé. Známe algoritmy buď používajú $\Omega(n)$ pamäte a vždy nájdu najpravdepodobnejšiu anotáciu, alebo potrebujú v najhoršom prípade iba $o(n)$ pamäte, ale negarantujú správny výsledok.

V práci zavádzame modifikáciu Viterbiho algoritmu, ktorá vždy nájde správny výsledok a použije na to v najhoršom prípade $O(1)$ a v najlepšom prípade $O(n)$ pamäte. Pamäťová náročnosť závisí na vstupnom modeli a spracúvanej sekvencii. Pre jednoduchú triedu modelov ukážeme, že očakávaná pamäť potrebná na spracovanie sekvencie dĺžky n je $O(\log(n))$. Ďalej predložíme experimentálne výsledky z oblasti hľadania génov v bioinformatike, ktoré naznačujú podobnú očakávanú pamäťovú zložitosť.

KĽÚČOVÉ SLOVÁ: skryté Markovovské modely, Viterbiho algoritmus, teória informácií, hľadanie génov