

---

Project report  
Summer School  
Jesper Nielsen

---

## 1 Introduction

A central problem in bioinformatics is that of alignment: Given a set of sequences and a phylogenetic tree, we believe relate these sequences, compute which part of a sequence correspond to which parts of the other sequences. The TKF91 model is a statistical model describing how one sequence may evolve into another. The TKF91 model can be rephrased as a HMM in multiple dimensions. Given a HMM model various statistics such as the most likely alignment and the probability the input sequences actually are related in the model can be computed. This can be done using dynamic programming and is efficient in the length of the sequences but scales badly with the number of sequences. A way to speed up the computations is to not compute the entire dynamic programming table, but to ignore less important parts. In this project we explore heuristics for inferring which parts of a higher-dimensional table may be ignored. Our idea is to align only a few of the sequences at first and use the information gained from these lower-dimensional alignments.

In this report we will start by briefly introducing our basic notation and the TKF91 model. Then we will discuss the construction of HMM's representing evolution in general before tying it together with the TKF91 model in particular. Next will be discussion of practical aspects of HMM construction followed by algorithms for actual use of a HMM. Finally corner cutting will be discussed and our results with this will be presented.

## 2 Basic notation

We first introduce the basic notation we are going to use. Let  $\Omega$  be our alphabet. For example  $\{A, C, G, T\}$ . We assume the same alphabet through the entire report. Let  $\Omega^l$  be the set of sequence of length  $l$  with characters from only from our alphabet  $\Omega$  and  $\Omega^* = \bigcup_{n=0}^{\infty} \Omega^n$  the set of all sequences over our alphabet, including the empty one. Given a sequence  $S \in \Omega^*$  define  $|S|$  to be the length of the sequence, that is:  $S \in \Omega^{|S|}$ .  $a \circ b$  the concatenation of  $a$  and  $b$  so that  $a \in \Omega^n$  and  $b \in \Omega^m \Rightarrow a \circ b \in \Omega^{n+m}$ . Finally let  $S_1, S_2, \dots, S_N \in \Omega^*$  be our input sequences, that is: The sequences we want to align.

To be able to address characters in a given sequence  $S$  we set  $S[o]$  to be the  $o$ 'th character of the sequence  $S$  and  $S[o : l]$  to be the subsequence of length  $l$  such that the last character in the subsequence is the  $o$ 'th character in  $S$ . For example let  $S = \text{ACAGGT}$  then  $S[1] = \text{A}$ ,  $S[|S|] = \text{T}$ ,  $S[4 : 2] = \text{AG}$  and  $S[|S| : |S|] = S$ . This notation we extend to multiple sequences by setting, for  $O, L \in \mathbb{Z}^N$  and  $S = (S_1, S_2, \dots, S_N)$ ,  $S[O] = (S_1[O_1], S_2[O_2], \dots, S_N[O_N])$  and  $S[O : L] = (S_1[O_1 : L_1], S_2[O_2 : L_2], \dots, S_N[O_N : L_N])$ .

### 3 The TKF91 model

In the TKF91 model [3], [5], [4] we consider a sequence of residues, that is: A strand of DNA evolving into another strand. To model this we conceptually let each residue have a link to its right. This link connects the residue with the residue to its right, except of course for the last residue which do not have any right neighbour. At the left-most end of the sequence is a link, called the immortal link not associated with any residue, but connected to the left-most one. The reason for the name will become clear shortly.

The model is that at any point in time a link may spontaneously spawn a new residue/link pair to its right and any residue/link pair may die. Since the immortal link does not have any residue associated with it this cannot die, thus the name. The rate at which new links are born is  $\lambda$  and the rate with which residues die is  $\mu$ . This means that, given a sequence  $S$ , after one unit of time has passed we expect  $\lambda(|S| + 1)$  new links to have been born since there is  $|S|$  mortal links and one immortal and we expect  $\mu|S|$  links to have died. This in turn implies that if we wait for an infinitely long time we expect the length of  $S$  to be  $\frac{1}{\mu/\lambda-1}$ .

Consider what would happen to a given mortal link in this model after some time  $t$  have elapsed. There are two possibilities: The link may have died or it may have survived. Denote the probability of the link surviving and spawning  $n - 1$  new links as  $p_n(t)$  and let  $p'_n(t)$  denote the probability that the link dies but leaves  $n$  descendants. Finally let  $p''_n(t)$  denote the probability that the immortal link spawns  $n$  new links during the time  $t$ . These probabilities can be defined through a set of differential equations. Solving them gives:

$$\begin{aligned} p_n &= \alpha\beta^{n-1}(1 - \beta) \\ p'_n &= \begin{cases} (1 - \alpha)(1 - \gamma) & \text{for } n = 0 \\ (1 - \alpha)\gamma\beta^{n-1}(1 - \beta) & \text{for } n > 0 \end{cases} \\ p''_n &= \beta^n(1 - \beta) \end{aligned}$$

where

$$\begin{aligned} \alpha &= e^{-\mu t} \\ \beta &= \lambda\beta(t) \\ \gamma &= 1 - \frac{\mu\beta(t)}{1 - e^{-\mu t}} \end{aligned}$$

and

$$\beta(\tau) = \frac{1 - e^{(\lambda-\mu)\tau}}{\mu - \lambda e^{(\lambda-\mu)\tau}}$$

Of course we also need to consider the substitution model. The above only describes indel events. Let  $\pi_x$  be the stationary probability of the residue  $x$ . When inserting a new residue the probability of that being any given residue  $x$  we simply set to  $\pi_x$ . If a link with residue  $x_1$  survives we can compute the probability that it will be  $x_2$  after time  $t$  as:

$$\begin{aligned} &\pi_{x_2}(1 - e^{-st}) && \text{for } x_1 \neq x_2 \\ e^{-st} + \pi_{x_2}(1 - e^{-st}) && \text{for } x_1 = x_2 \end{aligned}$$

where  $s$  is the substitution rate.

The TKF91 model was chosen in this project due to its simplicity. As we shall see shortly it will be used in a very flexible and powerful framework which can easily handle more refined models and it should thus be easy to extend this work to such models.

## 4 The Multiple HMM

We will use a multiple hmm (MHMM) to model the evolution through the tree. [1] A MHMM is a hidden markov model which emit symbols to multiple sequences. It may thus be viewed as having multiple time axes. In the following we will describe how to build a MHMM modelling  $S_1, \dots, S_N$ . Given our MHMM let  $\Phi$  be the set of states in the model. We let each state be named by a unique identifier, which may be anything. Of course  $|\Phi|$  is the number of states in our model. Define  $t(i, j)$  to be the probability of a transition from hidden state  $i$  to hidden state  $j$ . We need two special states: One to start in and one to end in a we will call them **START** and **END** and require from  $t$  that  $\forall i \in \Phi : t(i, \text{START}) = t(\text{END}, i) = 0$ . Each state may make emissions in any of the output sequences  $S_i$ , but only one character in each. Let  $E(i) \in \{0, 1\}^N$  be a vector describing which sequences state  $i$  emits a character to, such that  $E(i)_j = 1 \Leftrightarrow i$  emits a character to sequence  $j$ . Some states  $i$  may have  $E(i) = 0^N$  and thus not emit any characters, and in fact we require this is the case for **START** and **END**. Such a state we call a *silent state*. Now we know how much each state emits we also need to know what each state emits. This is decided randomly. Given a state  $i$  and a tuple of sequences  $X = (X_1, X_2, \dots, X_N)$  such that  $\forall j : |X_j| = E(i)_j$ , that is, the tuple could actually have been emitted by  $i$ , we denote the probability that  $i$  actually emits  $X$  as  $e(X, i)$ . Now we can, given our set of input sequences  $S$ , a MHMM and a sequence of hidden states  $\phi = (\phi_1, \phi_2, \dots, \phi_M)$ , where  $\phi_1 = \text{START}$  and  $\phi_M = \text{END}$ , compute the probability that the MHMM actually followed this sequence and emitted the sequence as:

$$p(S, \phi) = \left( \prod_{i=1}^{M-1} t(\phi_i, \phi_{i+1}) \right) \left( \prod_{i=1}^M e(S[\sum_{j=1}^i E(\phi_j) : E(\phi_i)], \phi_i) \right)$$

That is: Simply the product of the probabilities that the MHMM followed the path  $\phi$  and that it emitted the sequences  $S$  along the way. Note this may equivalently be expressed as:

$$p(S, \phi) = t(\text{START}, \phi_2) \left( \prod_{i=2}^{M-2} t(\phi_i, \phi_{i+1}) e(S[\sum_{j=1}^i E(\phi_j) : E(\phi_i)], \phi_i) \right) t(\phi_{M-1}, \text{END})$$

Where we require that  $\sum_{i=1}^M E(\phi_i) = (|S_1|, |S_2|, \dots, |S_N|)$ , that is: The sequence actually emits precisely the given sequences.

## 5 Transducer and branch HMMs

We will design our MHMM by considering the phylogenetic tree relating our input sequences  $S$  and model the evolutionary process along each branch in that tree. To describe the above evolutionary process we use a transducer HMM (THMM). A THMM is basically a MHMM which do not emit any characters to its first input sequence, but consumes them instead. It can thus be thought of as modifying a signal. A signal will arrive on its input sequence, it will be modified and output to its output sequences. On the branches in the phylogenetic tree we will place THMM's with one input sequence and one output sequence. These THMM's we call branch HMM's (BHMM's). To keep the probabilities right under this model we will require a certain structure of a BHMM. First of all all states will have a type: All BHMM's will have precisely one state of type *start* and one of type *end*. These are used for the obvious purpose and be silent states. It will have at least one state of type *wait*. Set the set of wait states:

$$\Phi_w = \{i \in \Phi \mid E(i) = 0^N \wedge i \neq \text{START} \wedge i \neq \text{END}\}$$

We will disallow any other silent states. The BHMM will be in a state of type *wait* while waiting for input from higher up in the tree and can only receive such input when in a wait state. To process input from higher up in the tree a BHMM will contain *match* and *delete* states. A *match* state consumes the input and emits a, possibly different, one. A *delete* state simply consumes the input and deletes it. States of those two types and the *end* state are collectively known as *receive* states. Let the set of receive states be:

$$\Phi_r = \{i \in \Phi \mid E(i)_1 = 1 \vee i = \text{END}\}$$

The final type of state is *insert*. *insert* nodes lets the branches insert characters into the sequence without receiving anything from the parent, hence the name. The set of insert states are:

$$\Phi_i = \{i \in \Phi \mid E(i) = (0, 1)\}$$

The idea is: A transducer waits in a wait state until it receives a symbol as input. If this symbol is **END** it goes to the end state. Otherwise it goes to an appropriate receive state and consumes the symbol. Next it may go to some insert states and emits its own symbols before it finally end in another wait state and waits for the next symbol. For a BHMM we require that the only way to go to a *receive* state is from a *wait* state and that that is the only way to leave a *wait* state:

$$t(i, j) > 0 \Rightarrow (i \in \Phi_w \Leftrightarrow j \in \Phi_r)$$

Secondly we require that when a character is received from the parent the probability this character is consumed is 1:

$$\forall i \in \Phi_w, \omega \in \Omega : \sum_{j: E(j)_1=1} \sum_{\omega' \in \Omega^{E(j)_2}} t(i, j) e(j, (\omega, \omega')) = 1$$

Equivalently the probability of stopping if the parent stops is also one:

$$\forall i \in \Phi_w : t(i, \text{END}) = 1$$

And finally the probability of leaving all other states should be one:

$$\forall i \in \Phi \setminus (\Phi_w \cup \{\text{END}\}) : \sum_j t(i, j) = 1$$

## 6 The Evolutionary HMM

Assume we are given a binary tree with  $N$  nodes corresponding to the  $N$  input sequences  $S_n$ . We will number the nodes in a depth-first manner such that the root has number 1, if  $i$  is a node and  $j$  is any descendant of  $i$  then  $i < j$ , and if  $k$  is a sibling of  $i$  so that  $i < k$  then  $j < k$ . With any node we will associate the BHMM on the branch leading to it. The root also have a BHMM associated with it on a branch from some imaginary ancestor. This BHMM for the root should be constructed so as not to have any receive state except END.

To avoid confusion we will rename the definitions for the HMM if it is a BHMM. For the evolutionary HMM (EHMM) we will retain the old names. Let the state space be  $\Psi$ , the transition probabilities for a branch of length  $T$  be  $\tau(i, j|T)$  and the emission indicator be  $F(i)$ , with the emission probability  $f(X, i)$ . We will rename the start state **SBR** and the end state **EBR**. To ease discussion we will formalise the type of a state in a BHMM with  $\chi(i)$  where:

$$\chi(i) = \begin{cases} W & \text{for } i \neq \text{SBR} \wedge i \neq \text{EBR} \wedge F(i) = (0, 0) \\ M & \text{for } F(i) = (1, 1) \\ D & \text{for } F(i) = (1, 0) \\ I & \text{for } F(i) = (0, 1) \\ S & \text{for } i = \text{SBR} \\ E & \text{for } i = \text{EBR} \end{cases}$$

In the EHMM states are tuples of the states of the BHMM's:  $\phi = (\psi_1, \psi_2, \dots, \psi_N)$ . This includes **START** = (**SBR**, **SBR**, ..., **SBR**) and **END** = (**EBR**, **EBR**, ..., **EBR**). Define the active node of a given state in the EHMM to be:

$$\alpha(\phi) = \begin{cases} \max\{n \mid \chi(\phi_n) \neq W\} & \text{if } \exists n : \chi(\phi_n) \neq W \\ 1 & \text{otherwise} \end{cases}$$

That is: The leftmost node of the tuple that is not in a wait state and the root node otherwise. A transition by the EHMM is made by the active node making any legal transition in the associated BHMM and all its children making transitions to absorb any emitted characters, recursively. Only the root node is allowed to go to the **EBR** state in which case the entire HMM stops. The input to a given branch HMM of course matches exactly the output of its parent and the output of the entire EHMM is the output of all the BHMM's, where node one outputs to  $s_1$

and so on. The transition probabilities are the product of the transitions done by all the BHMM's. The probability of a transition from  $\phi_1 = (\psi_{11}, \psi_{12}, \dots, \psi_{1N})$  to  $\phi_2 = (\psi_{21}, \psi_{22}, \dots, \psi_{2N})$ , where  $\Psi'$  is the set of nodes actually making a transition, is simply the product of the transition probabilities:

$$t(\phi_1, \phi_2) = \prod_{i \in \Psi'} \tau(\psi_{1i}, \psi_{2i} | T_i)$$

where  $T_i$  is the length of the branch to node  $i$ . The emission indicator is:

$$E(\phi_2)_i = \begin{cases} F(\psi_{2i}) & \text{for } i \in \Psi' \\ 0 & \text{otherwise} \end{cases}$$

The emission probability is also simply the product of the emission probabilities of the nodes actually emitting anything. If  $A(i)$  is the immediate ancestor of  $i$  then:

$$e(X, \phi_2) = \prod_{i \in \Psi'} f((X_{A(i)}[F(\psi_{2i})_1 : F(\psi_{2i})_1], X_i[F(\psi_{2i})_2 : F(\psi_{2i})_2]), \psi_{2i})$$

## 7 EHMM's and TKF91

Now we will see how to use the above EHMM framework with the TKF91 process. Again I would like to stress that the TKF91 process only is chosen because it is simple to work with and that everything should generalise to a more sophisticated model. We, of course, still assume we are given a tree describing the relationships of our output sequences. This tree will have a root and we will first consider the special transducer needed at this root to generate the oldest ancestral sequence. The root transducer will have four states. Of course it has the **SBR** and **EBR** states for starting and stopping. It is also required to have a **WAIT** state. Finally it has a simple **INSERT** state. It does not have any other receive states than **EBR** since it has no ancestral sequence to receive symbols from. The transition probabilities are [2]:

	INSERT	WAIT	EBR
SBR	$\kappa$	$1 - \kappa$	
INSERT	$\kappa$	$1 - \kappa$	
WAIT			1

Where empty cells means 0 or "no transition" and  $\kappa = \frac{\lambda}{\mu}$  the probability a symbol exists after waiting for an infinitely long time.

The transducer placed on the branches further down the tree will have to be more sophisticated to handle the symbols coming from further up the tree. It will have the same four states as the root transducer, but also have a **DELETE** state that ignores incoming symbols and a **MATCH** state that emits the symbols possibly after

a substitution. The transition probabilities becomes:

	INSERT	MATCH	DELETE	WAIT	EBR
SBR	$\beta$			$1 - \beta$	
INSERT	$\beta$			$1 - \beta$	
MATCH	$\beta$			$1 - \beta$	
DELETE	$\gamma$			$1 - \gamma$	
WAIT		$\alpha$	$1 - \alpha$		1

To define above probabilities we will first set  $\tau$  to the time along the branch the transducer is placed on and the function

$$\beta(\tau) = \frac{1 - e^{(\lambda - \mu)\tau}}{\mu - \lambda e^{(\lambda - \mu)\tau}}$$

We now get:

$$\begin{aligned}\alpha &= e^{-\mu\tau} \\ \beta &= \lambda\beta(\tau) \\ \gamma &= 1 - \frac{\mu\beta(\tau)}{1 - e^{-\mu\tau}}\end{aligned}$$

We now need to consider the emission probabilities. Note that only three states have any emissions: INSERT, MATCH and DELETE. For both insert states we simply set  $f((, x), \text{INSERT}) = \pi_x$  and for the DELETE state we have to set  $f((x, ), \text{DELETE}) = 1$  to fulfill the normalization requirements of a BHMM. Finally the probabilities for MATCH state is:

$$f((x_1, x_2), \text{MATCH}) = \begin{cases} \pi_{x_2}(1 - e^{-st}) & \text{for } x_1 \neq x_2 \\ e^{-st} + \pi_{x_2}(1 - e^{-st}) & \text{for } x_1 = x_2 \end{cases}$$

Note how these definitions match those in the section on the TKF91 model.

## 8 Removing unwanted sequences

Sometimes we may not be interested in the emissions of a given sequence. For example we will usually only know the sequences at the leaves, while the inner nodes represent some unknown prehistoric sequence, and we are thus only able to align the leaf nodes. Another reason, that is particularly relevant for this project is that we will want to align only a few of the known sequences and use the information from that alignment to speed up the alignment of all the sequences. The unknown sequences may be sampled or summed out. In this project we will not consider sampling and will simply sum out unwanted sequences.

Assume we have a model in the form of an EHMM, that represents alignment of the sequences  $S = \{s_1, s_2, \dots, s_N\}$  but we only wish to align the subset  $S' = \{s'_1, s'_2, \dots, s'_{N'}\} \subseteq S$ . The new EHMM will have the same states with the same transition probabilities as the old one, except the emission indications and probabilities will be different. Given a state  $i$  in the old EHMM call the corresponding

state in the new EHMM  $i'$ . Similarly we will call the emission indication  $E'$  and the emission probability  $e'$  in the new EHMM.

Let  $R = S \setminus S'$  and set  $j_k$  so that  $s'_k = s_{j_k}$ . Then the emission indication for the new EHMM will simply be the old emission indication with the unwanted sequences removed:

$$E'(i') = (E(i)_{j_1}, E(i)_{j_2}, \dots, E(i)_{j_{N'}})$$

and the emission probability of  $X' \in \Omega^{N'}$  will be the sum over all the emission probabilities of the old EHMM where  $X_{j_k} = X'_k$ :

$$E'(X', i') = \sum_{\substack{X \in \Omega^N \\ :X_{j_k} = X'_k}} E(X, i)$$

Of course all states  $i'$  that represents states  $i$  in the old EHMM, where  $i$  only emitted anything to an unwanted sequence will now become a silent state. We will shortly see how to remove those.

## 9 Silent state removal

If we build an EHMM using the methods described above. We will get silent states from two sources. First of all every state created by a transducer moving to a wait state will be silent. Secondly we will get a silent state if we eliminate all the sequences where a given states has emissions. We wish to remove these states because they would cause infinite loops in the recursive algorithm defined later and even if they did not they would consume computer memory and processor resources needlessly.

We will now construct a reduced EHMM where silent states have been removed. Please note that we will always need to retain two silent states: the **START** and **END** states. Remember we call out state space  $\Phi$ . Define the set of good states to be  $\Phi_G = \{\phi \in \Phi \mid \exists i : E(\phi) = 1\} \cup \{\text{START}, \text{END}\}$  and the set of bad (silent, unwanted) states  $\Phi_B = \Phi \setminus \Phi_G$ . Let  $t$  be a matrix representing the transition probabilities such that  $t_{ij} = t(i, j)$  and  $t'$  be the matrix of new transition probabilities we want to find: That is the effective transition probabilities between states in  $\Phi_G$ .

Split  $t$  into four matrices  $a$ ,  $b$ ,  $c$  and  $d$  in a way so that  $a$  contains all good-good transitions,  $b$  all good-bad transitions,  $c$  all bad-good and  $d$  all bad-bad and  $t = a + b + c + d$ . Formally:

$$\begin{aligned} a_{ij} &= \begin{cases} t_{ij} & \text{if } i \in \Phi_G \wedge j \in \Phi_G \\ 0 & \text{otherwise} \end{cases} \\ b_{ij} &= \begin{cases} t_{ij} & \text{if } i \in \Phi_G \wedge j \in \Phi_B \\ 0 & \text{otherwise} \end{cases} \\ c_{ij} &= \begin{cases} t_{ij} & \text{if } i \in \Phi_B \wedge j \in \Phi_G \\ 0 & \text{otherwise} \end{cases} \\ d_{ij} &= \begin{cases} t_{ij} & \text{if } i \in \Phi_B \wedge j \in \Phi_B \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Now, a transition between two good states  $i, j \in \Phi_G$  can happen directly, it can go through one bad state, it can go through two bad states and so on. This means the effective transition probability becomes:

$$t'_{ij} = a_{ij} + \left( \sum_k b_{ik} c_{kj} \right) + \left( \sum_k \sum_l b_{ik} d_{kl} c_{lj} \right) + \left( \sum_k \sum_l \sum_m b_{ik} d_{kl} d_{lm} c_{mj} \right) + \dots$$

and thus we get

$$\begin{aligned} t' &= a + bc + bdc + bd^2c + \dots \\ &= a + \sum_{i=0}^{\infty} bd^i c \\ &= a + b(I - d)^{-1}c \end{aligned}$$

where  $I$  is the identity matrix.

## 10 Probability of given sequences

Given an EHMM and a set of sequences  $S = (S_1, \dots, S_N)$  there are several interesting questions one might want to ask. They will usually be of the form: What is the probability  $S$  were generated by the EHMM and what is the most likely sequence of states in the EHMM to generate  $S$ . These two questions can be varied by considering only a subset of the sequences in  $S$ ; only the leaves for example. To calculate the probability of the emission of  $S$  we define the forward algorithm as  $\alpha(L, \phi)$  is the probability of having emitted  $S[L : L]$  and being in state  $\phi$  having started in state **START**. We can calculate  $\alpha$  by simply summing over all possible paths through the EHMM, ending in  $\phi$ , but this would be intractable since there are an exponential number of these paths. Instead we can define it recursively by:

$$\alpha(L, \phi) = \begin{cases} 1 & \text{if } L = 0 \wedge \phi = \text{START} \\ 0 & \text{if } L = 0 \wedge \phi \neq \text{START} \\ 0 & \text{if } L \neq 0 \wedge \phi = \text{START} \\ e(S[L : E(\phi)], \phi) & \\ \sum_{\phi': (\forall i: L_i - E(\phi') \geq 0)} t(\phi', \phi) \alpha(L - E(\phi), \phi') & \text{otherwise} \end{cases}$$

We now see that the probability of the emission of  $S$  is simply  $\alpha(L, \text{END})$ . This can be implemented on a computer use a technique called dynamic programming or memorization. The key observation is that if we start by wanting to evaluate  $\alpha(L, \text{END})$  and start doing the recursion, for a given  $L'$  and  $\phi'$  we will end up evaluating  $\alpha(L', \phi')$  several times because there are several different ways through the EHMM that emits the same symbols.  $\alpha(L', \phi')$  is always the same, no matter which way through the EHMM we reached it and thus we may save the result once computed and use it repeatedly instead of computed it again. The algorithm becomes: Create an  $N + 1$ -dimensional table: One dimension for each input sequence and one for the states. Now fill out the table with  $\alpha(L, \phi)$  starting at  $(0, 0, \dots, 0)$  and working your way towards  $(|S_1|, |S_2|, \dots, |S_N|, |\Phi|)$  in lexicographical order. That way the cells you will need will always be filled before you need them. This algorithm is called this

*forward* algorithm. This table will take up  $\mathcal{O}(|S_1||S_2|\dots|S_N||\Phi|)$  memory and take  $\mathcal{O}(|S_1||S_2|\dots|S_N||\Phi|^2)$  time to compute. This is much better than the exponential execution time, but will still be prohibitive for large numbers of sequences. A note on practical implementations: Most probabilities in the MHMM are significantly smaller than one. We end up multiplying a lot of these and in the execution of the above algorithm you may fairly fast reach a point where the result is below the dynamic range of your computers floating point format. This will result in your result being rounded to zero. The common solution to this is to do the computations in logarithmic space.

## 11 Finding the most likely alignment

This can be done in nearly the same way as the above. Define  $v(L, \phi)$  to be the probability of emitting  $S[L : L]$  while following the most probable path through the EHMM and ending in  $\phi$ .  $v$  can be computed recursively by:

$$v(L, \phi) = \begin{cases} 1 & \text{if } L = 0 \wedge \phi = \text{START} \\ 0 & \text{if } L = 0 \wedge \phi \neq \text{START} \\ 0 & \text{if } L \neq 0 \wedge \phi = \text{START} \\ e(S[L : E(\phi)], \phi) & \\ \max_{\phi': (\forall i: L_i - E(\phi')_i \geq 0)} \{t(\phi', \phi)v(L - E(\phi), \phi')\} & \text{otherwise} \end{cases}$$

This algorithm is called the *viterbi* algorithm. We note the similarity with the above forward algorithm and it does indeed have same memory and processing complexity. Filling out the above table only tells you the probability of the most likely path through the EHMM, though. We would like to find an actual alignment. First we observe that a path through an EHMM actually is an alignment. Simply let characters emitted at the same time in the EHMM be aligned together while leaving gaps in those sequences not emitted to. We still, however, need to find the actual path. This can be done by simple backtracking. We know the most likely path passes through  $(|S_1|, |S_2|, \dots, |S_N|, \text{END})$  and that this is the last cell visited. We also know it reached that state from the most likely state with a transition to this state. Thus we can find the second-to-last cell visited by examining all cells representing a state able to do the transition and choosing the one containing the highest probability. This process we simply repeat until we find the cell  $(0, 0, \dots, 0)$  and this is exactly the path we are searching for.

## 12 Probability of an alignment using a given cell

Some paths through the EHMM and above table are much more likely than others. The difference in probability will often be so great that a lot of the cells in the table adds so little to our final result that we cannot tell whether they were actually computed or simply left as zero. To help us find out which cells we do not need to

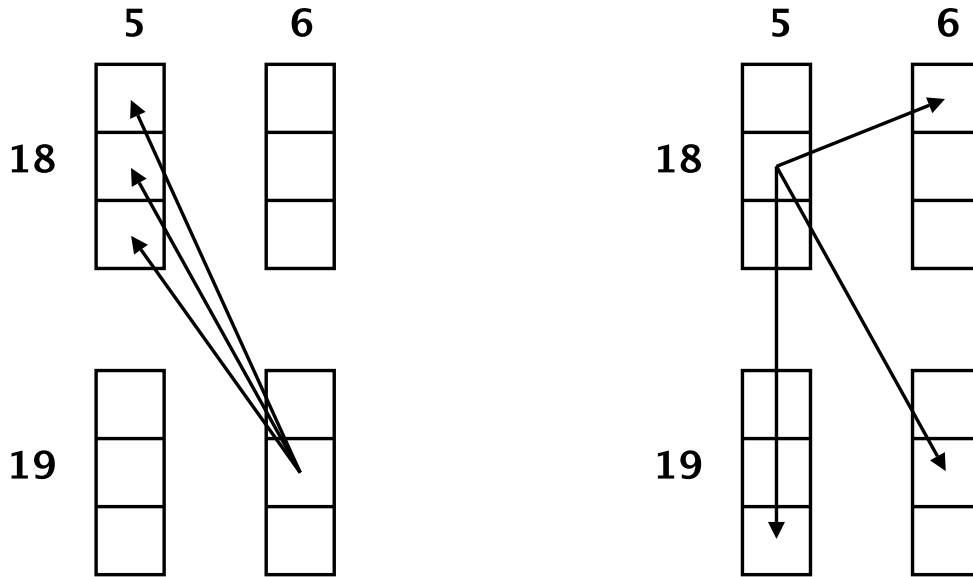
visit we would like to compute the probability of passing through the given cell. For this we will use the *backward* algorithm. Let  $\beta(L, \phi)$  be the probability of emitting  $S[(|S_1|, \dots, |S_N|) : L]$  given that we start in the state  $\phi$ .  $\beta$  can be computed in a way similar to  $\alpha$  and  $v$ , but backwards:

$$\beta(L, \phi) = \begin{cases} 1 & \text{if } L = (|S_1|, \dots, |S_N|) \wedge \phi = \text{END} \\ 0 & \text{if } L = (|S_1|, \dots, |S_N|) \wedge \phi \neq \text{END} \\ 0 & \text{if } L \neq (|S_1|, \dots, |S_N|) \wedge \phi = \text{END} \\ \sum_{\phi': (\forall i: L_i + E(\phi')_i \leq |S_i|)} [e(S[L + E(\phi') : E(\phi')], \phi') & \text{otherwise} \\ t(\phi, \phi')\beta(L + E(\phi'), \phi')] & \end{cases}$$

Using above we can compute the probability of the EHMM generating the given sequences a passing through a given cell  $(L, \phi)$  as:  $\alpha(L, \phi)\beta(L, \phi)$ . Define a *unit* as the set of cells with same offset into the sequences  $L$ . Then the probability of generating the given sequences and passing through a given unit is simply the sum of the probabilities for the cells in the unit, assuming there are no silent states in the unit:  $\sum_{\phi} \alpha(L, \phi)\beta(L, \phi)$ . If we want the probability of the unit being visited given the input sequences simply normalise with the probability of the sequences alone:

$$\frac{\sum_{\phi} \alpha(L, \phi)\beta(L, \phi)}{\alpha((|S_1|, \dots, |S_N|), \text{END})}$$

The above definition  $\beta$  have a few practical problems though. First of all we note that the emission probability which we would only evaluate once in the previous two algorithms now have to be evaluated  $|\Phi|$  times for each cell. Secondly: For a given cell the forward and the viterbi algorithms will only access cells in the same unit. This algorithm will access multiple different units for a given cell.



A section of a table of a EHMM with three states showing parts of columns 5 and 6 and rows 18 and 19. The left illustration shows how a cell gets its values from one

other unit in the forward algorithm. The right how a cell in the backward algorithm gets its values from several other units.

The way moderne computers are designed they expect you to access memory in contiguous scans while moving forward and are thus designed to be really fast when you this is the case. The backward algorithm however accesses the cells backward and with the above explained scattered memory access pattern. These two facts means the backward algorithm may completely defeat the design of the memory hierarchy of a modern computer. To design an efficient backward algorithm we first notice that to provide forward iteration we can simply reverse the input sequences. This, however does not help us with the scattered memory access pattern and the multiple evaluations of the emission probability. Define instead:

$$\beta'(L, \phi) = \beta(L, \phi)e(S[L : E(\phi)])$$

From  $\beta'$  it is, of course, easy to find  $\beta$ , but the recursions for  $\beta'$  is:

$$\beta(L, \phi) = \begin{cases} 1 & \text{if } L = (|S_1|, \dots, |S_N|) \wedge \phi = \text{END} \\ 0 & \text{if } L = (|S_1|, \dots, |S_N|) \wedge \phi \neq \text{END} \\ 0 & \text{if } L \neq (|S_1|, \dots, |S_N|) \wedge \phi = \text{END} \\ e(S[L : E(\phi)]) \sum_{\phi': (\forall i: L_i + E(\phi')_i \leq |S_i|)} t(\phi, \phi')\beta(L + E(\phi), \phi') & \text{otherwise} \end{cases}$$

If we run this on the input sequences reversed the algorithm simple becomes the forward algorithm, except it is run on the EHMM in the other direction: All transition probabilities are reversed. This also means it is simple to implement a single function that can be used for calculating both the forward, backward and viterbi algorithm, reducing the number of lines of code and thus making maintenance of the code simpler and less error-prone.

## 13 Cornercutting

We will be using two running examples shown in the next two figures. The first example is the alignment of the two sequences:

GGCTGTTTACCCAATAGCATTAGTTGTTCTGGAGCAGCGGTTTCACGCC

and

GGCTGTTTACCCAATAGCATTAGTTGTTTCGATACACGCC

The first sequence is completely randomly generated. The second is a copy of the first sequence, with a large section deleted and som random changes made in the area around the deletion.

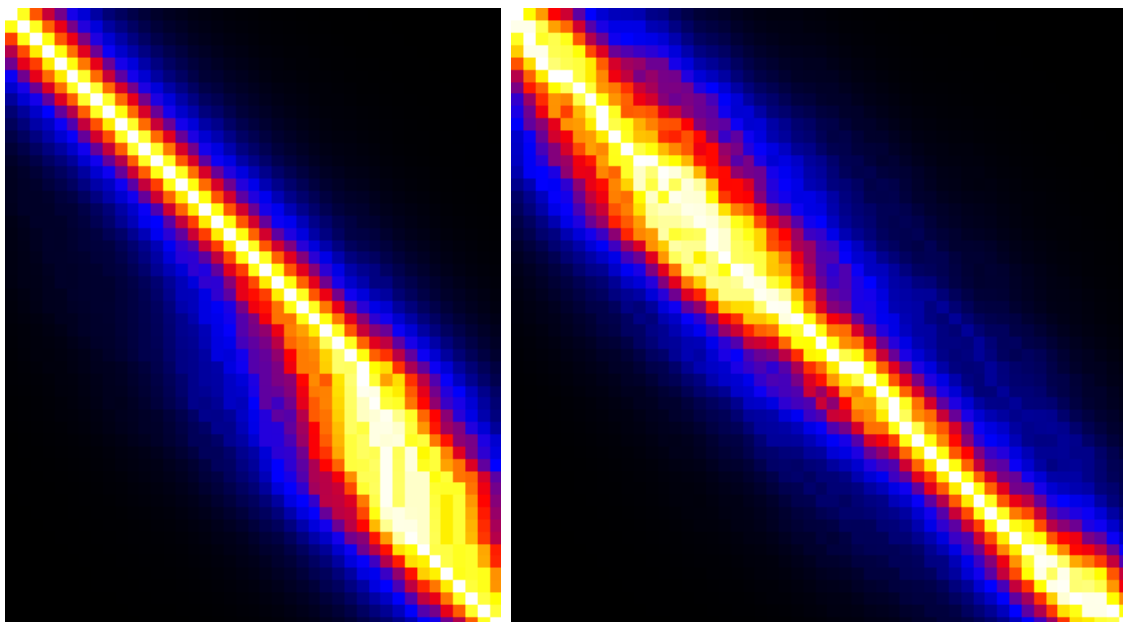
The second example are the two sequences:

CCGGCGGATGTATGCTTTCTGTTTACACGTGACTTCTACAGGACATATCT

and

CGCTGGAGTACTCGGTTGATTAACACGATGATTCTACAGGACAATTCGCT

These two are generated by first generating a completely random ancestral sequence and next simulating an evolution-like process like the TKF91 model on them.



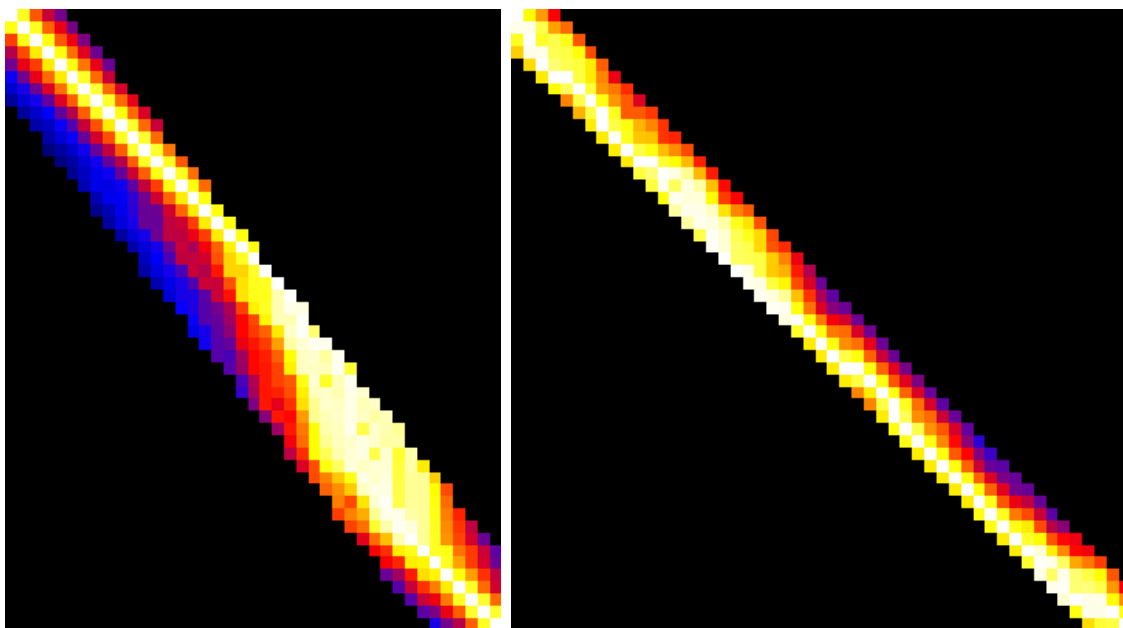
*Heatmap showing the probability of an alignment visiting each unit in the alignments of the two examples. On the left the first example with the deletion in the lower right corner and on the right the second example.*

In the colors of figure can be interpreted as:

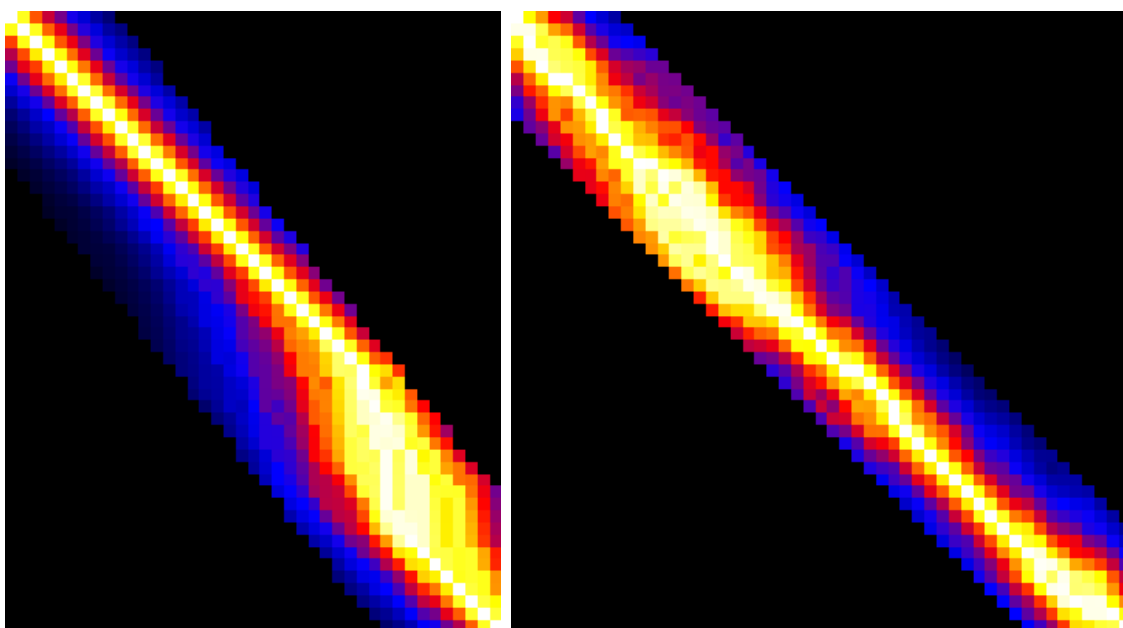
Color	Probability
Black	0
Blue	$1.7 \cdot 10^{-20}$
Red	$1.3 \cdot 10^{-10}$
Yellow	$7.9 \cdot 10^{-5}$
White	1

The inspiration for corner-cutting is the fact that the upper right and lower left corners of the above figures are nearly completely black. This means that by not visiting these corners and simply leaving their values as zero it has little influence on our final result, whether it be the forward or the viterbi algorithm we are running.

The obvious way to do corner-cutting is to choose a distance  $d$  and only visit units with a manhattan distance (or similar measure) less than  $d$  to the diagonal, thus creating a band of evaluated units. This, however, is not very efficient. The problem can easily be seen in example one: At the indel event a wide band is needed, yet for about half the alignment only a narrow band is needed.

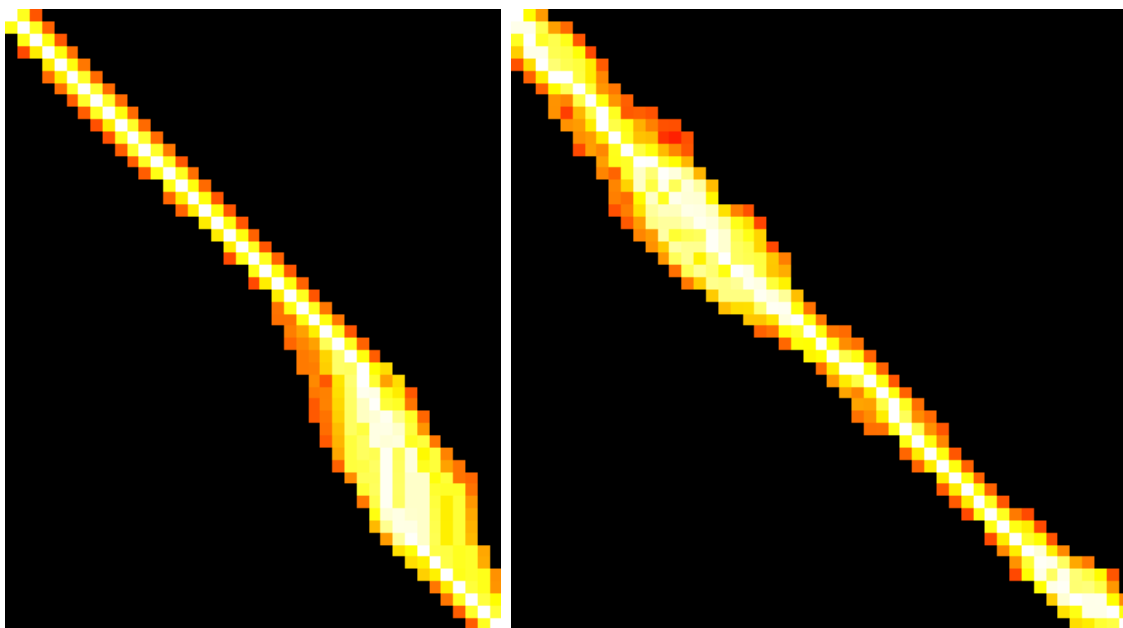


*Trying to do aggressive cornercutting with a simple banding heuristic makes us miss important units giving us results that are significantly different from the true values. Notice how the probabilities are actually different from the ones shown above. Cutting away important units forces the algorithm to find another way to do the alignment.*



*If we do the banding more defensively we may get the correct answers, but will spend a lot of time visiting unneeded units.*

It is easy to compute which units should be evaluated once the forward and the backward algorithm has been run, the problem is of course that this information is wanted before those algorithms are run.



*Here only units with a probability of being used greater than  $10^{-8}$  have been visited resulting in a good compromise. Notice that this example is, of course, only for illustration. To do above computation the complete forward and the complete backward algorithms were first run enabling us to do a third calculation really fast. If it really was the result we were interested in it would be faster and simpler to simply use the result of the first forward algorithm.*

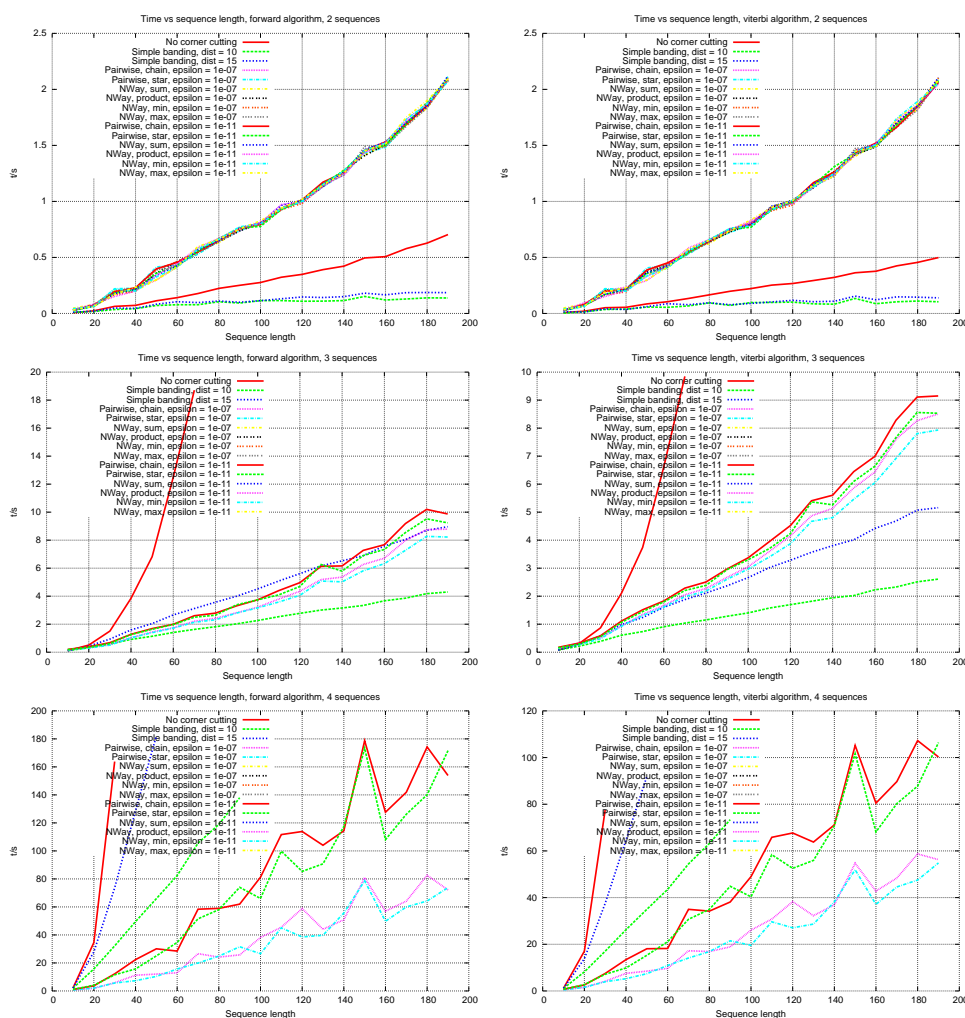
The concept of corner cutting however, works equally well on multiple alignments and when doing a multiple alignment the alignment of a small subset of the sequences will be very cheap compared to the alignment of all the sequences. Thus it may pay to align small subsets of the sequences and use the information from those alignment to guide corner cutting of larger alignments.

We consider two ways of doing this, both based on pairwise alignments. The problem we are trying to solve is: Given a tuple  $(x_1, \dots, x_{n-1})$ , being a partial coordinate into our  $N$ -dimensional dynamic programming table, find a range of  $n$  we should visit. This gives us a set of other partial coordinates  $(x_1, \dots, x_n)$  which we can each visit and solve recursively until we reach a complete coordinate  $(x_1, \dots, x_N)$ . The first idea is to base our choice on one of the first coordinates  $x_i$ , where  $i < n$ . Simply do a pairwise alignment of  $S_i$  and  $S_n$  and let the set of  $x_n$  be the set such that the entry  $(x_i, x_n)$  in the table of the pairwise alignment is greater than some threshold  $\varepsilon$ . There is no reason to expect any given  $i$  to be better than any other in general. I have implemented two variants of this: One where we let  $i = 1$  and one where we let  $i = n - 1$ . The assumption in the above, is that if a certain alignment of two sequences  $S_i$  and  $S_j$  is very unlikely it will still be very unlikely if you also align with an extra sequence  $S_k$ . However you choose  $i$  with a deterministic strategy it will be a bit arbitrary, and there may exist certain pairs of sequence which inherently does not give a very good results. The second idea it therefore to compute a pairwise alignment of all pairs of sequences and use these. Let  $c(x_i, x_n)$  be the value in unit  $(x_i, x_n)$  in the table of pairwise alignment between  $S_i$  and  $S_n$ . Given  $(x_1, \dots, x_{n-1})$  we

chose all  $x_n$  such that  $f\{c(x_1, x_n), c(x_2, x_n), \dots, c(x_{n-1}, x_n)\} > f\{\varepsilon, \varepsilon, \dots, \varepsilon\}$  for some appropriate function  $f$ . In this project this has been implemented with  $f$  as: Sum, product, maximum and minimum.

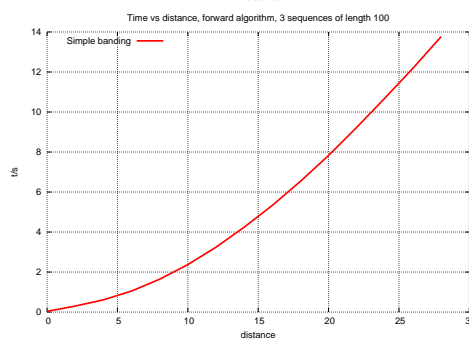
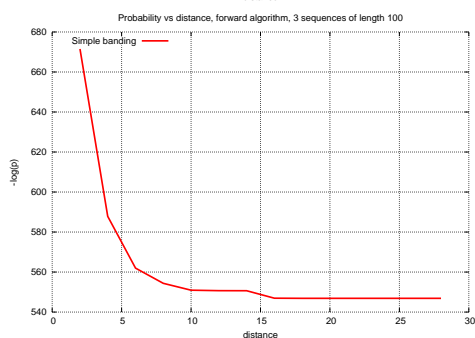
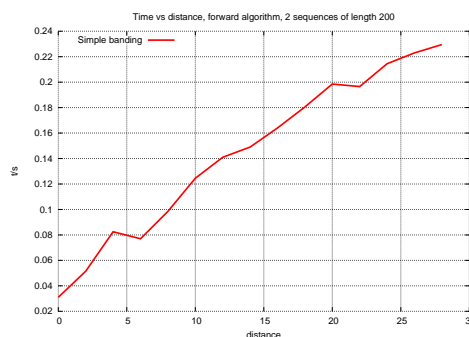
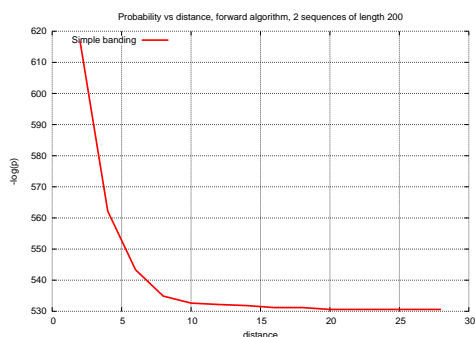
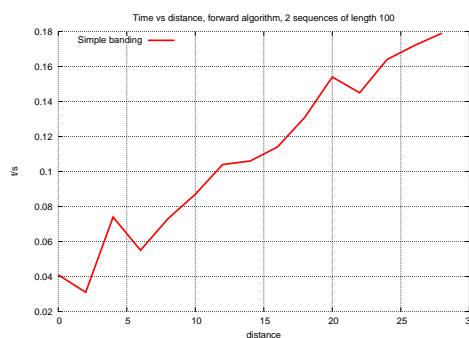
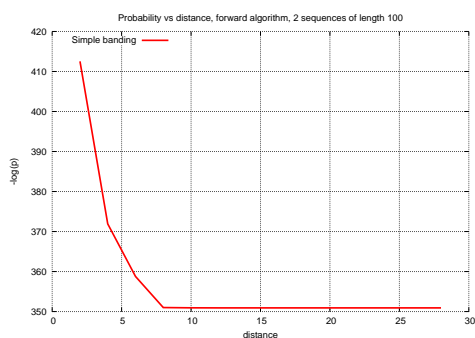
## 14 Results

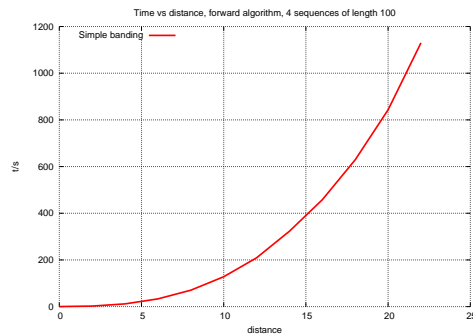
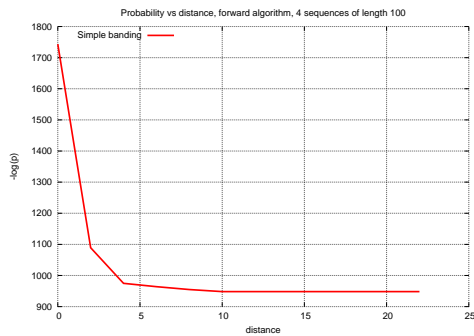
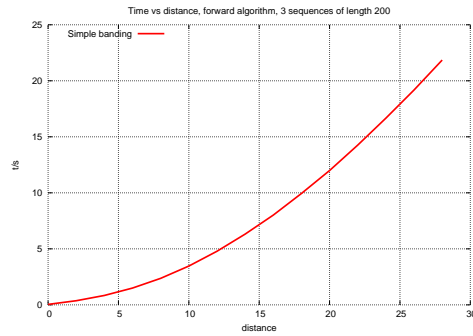
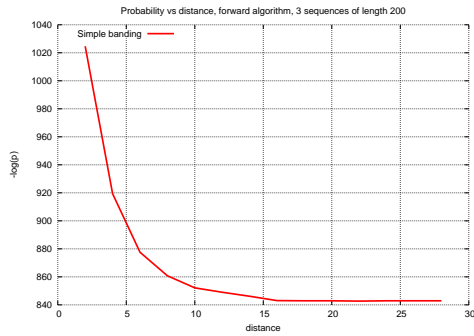
We tested the cornercutting on randomly generated strings. The strings were generated by an TKF91 like process on a star-shaped tree with the root unknown when doing the actual alignment. More precisely: First an ancestral string were generated as a sequences of length  $m$  with each character chosen uniformly amongst A, C, G and T. Next subsequences of this ancestral sequence were chosen as volatile. From the ancestral sequence we derived  $n$  sequences used for the actual testing. We did this by randomly deleting, inserting or substituting characters on the ancestral sequence, with the probability of changes happening significantly greater in the volatile regions. Each test was repeated ten times and the values shown are the average. Similar tests were all run on the same actual data.



These first benchmarks simply shows the time usage on sequences of different length. We should bear in mind that these numbers actually are completely worthless without knowing the quality of the results. Nonetheless we see that for two sequences the fastest strategy is the naive banding, no cornercutting fares well, while the strategies based on probabilities trails far behind. This is because the strategies based on probabilities use more time doing preprocessing than actually aligning, while the simple banding does not use any time on preprocessing. For more than two sequences this changes. The time spent preprocessing becomes negligible compared to the time spent aligning and thus only the advantages gained by corner-cutting can be seen. Generally no corner-cutting is very slow, the speed of simple banding depends entirely on sequence length and the distance parameter, while the speed of the probabilistic approaches depend on sequence length,  $\varepsilon$  and the data itself.

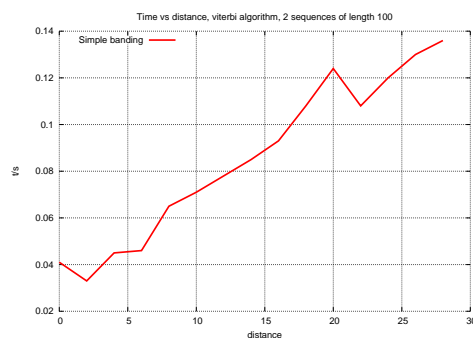
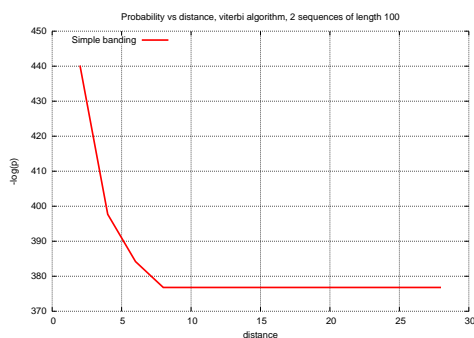
Let us first investigate the simple banding strategy:

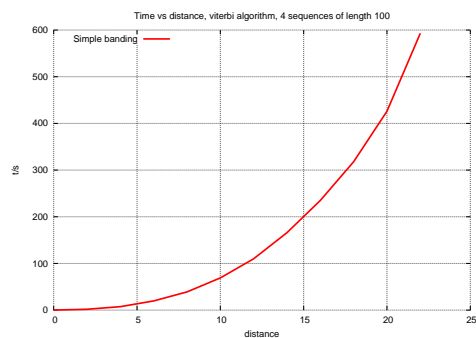
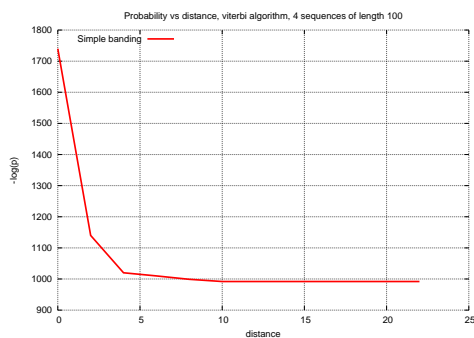
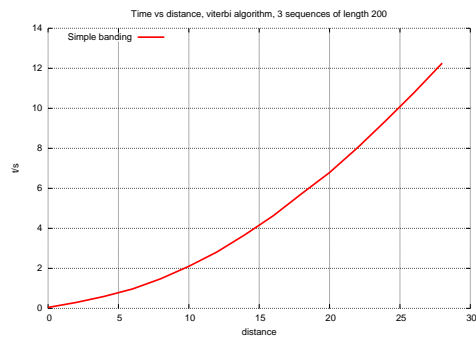
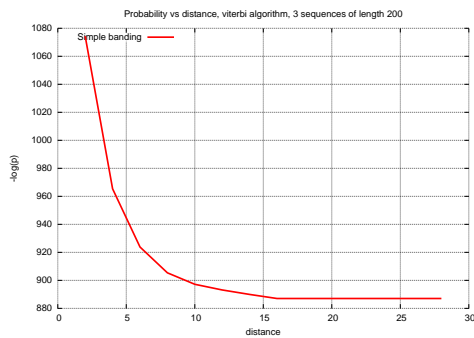
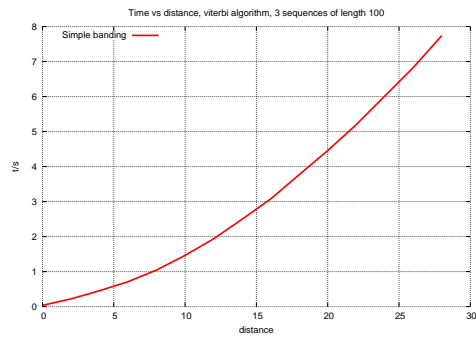
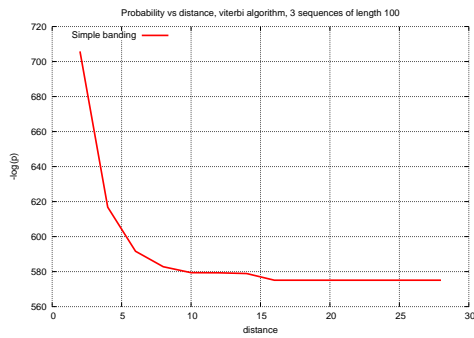
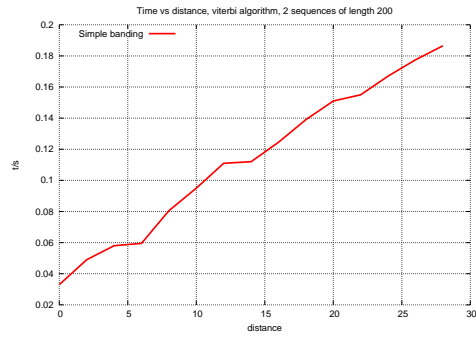
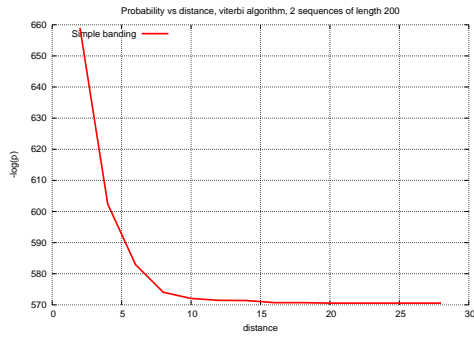




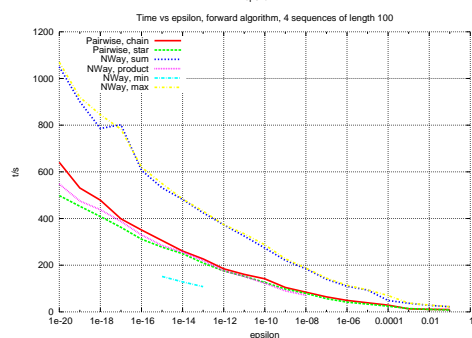
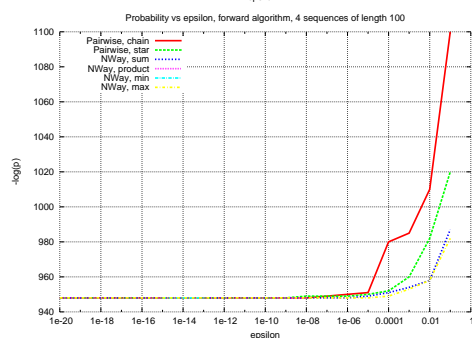
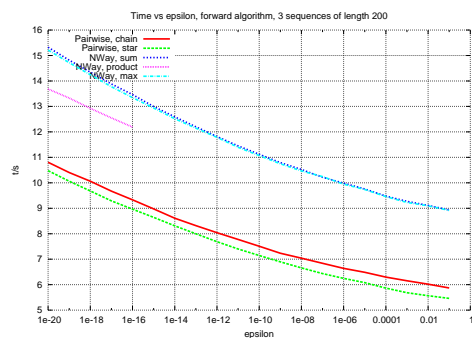
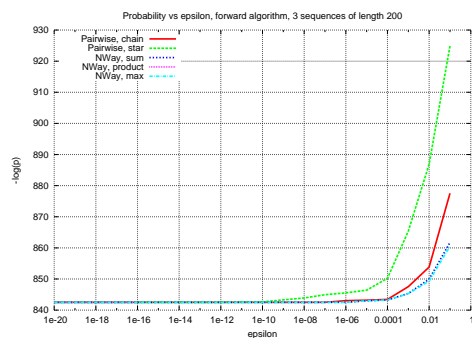
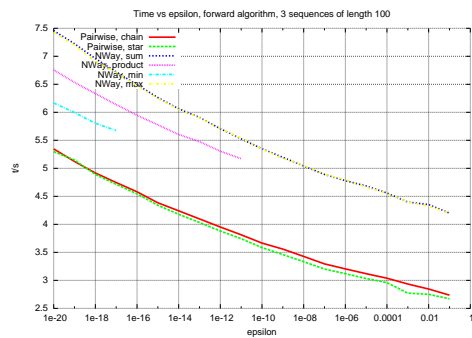
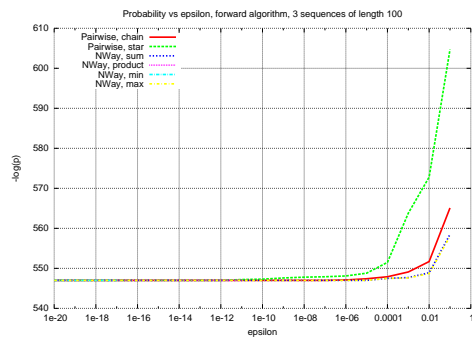
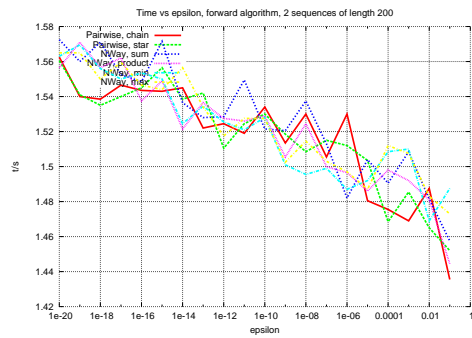
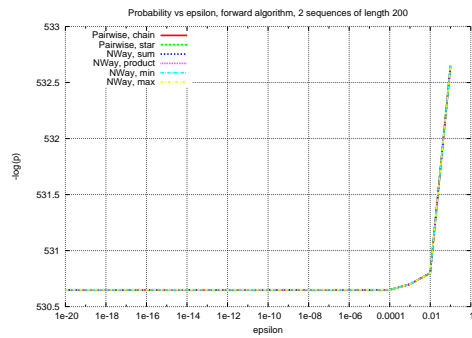
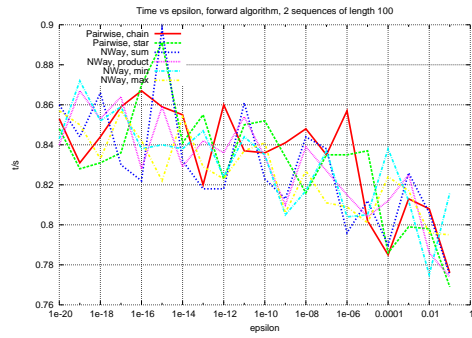
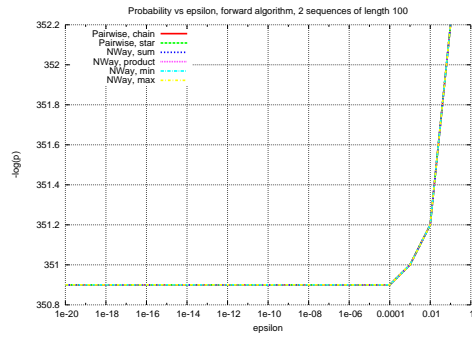
The fact that it is hard to say anything general from above graphs speaks against the strategy of simple banding. We would like to be able to give general recommendations for the distance parameter, but no such seem obvious. For the current data a distance of 20 seems enough, but I strongly suspect the data itself has a large influence on this number. In general the more large insertion or deletions the data contain the wider a band will be needed. This has not been investigated further as the simple banding strategy is not the main focus of this report.

The results for the viterbi algorithm are similar and only included for completeness:





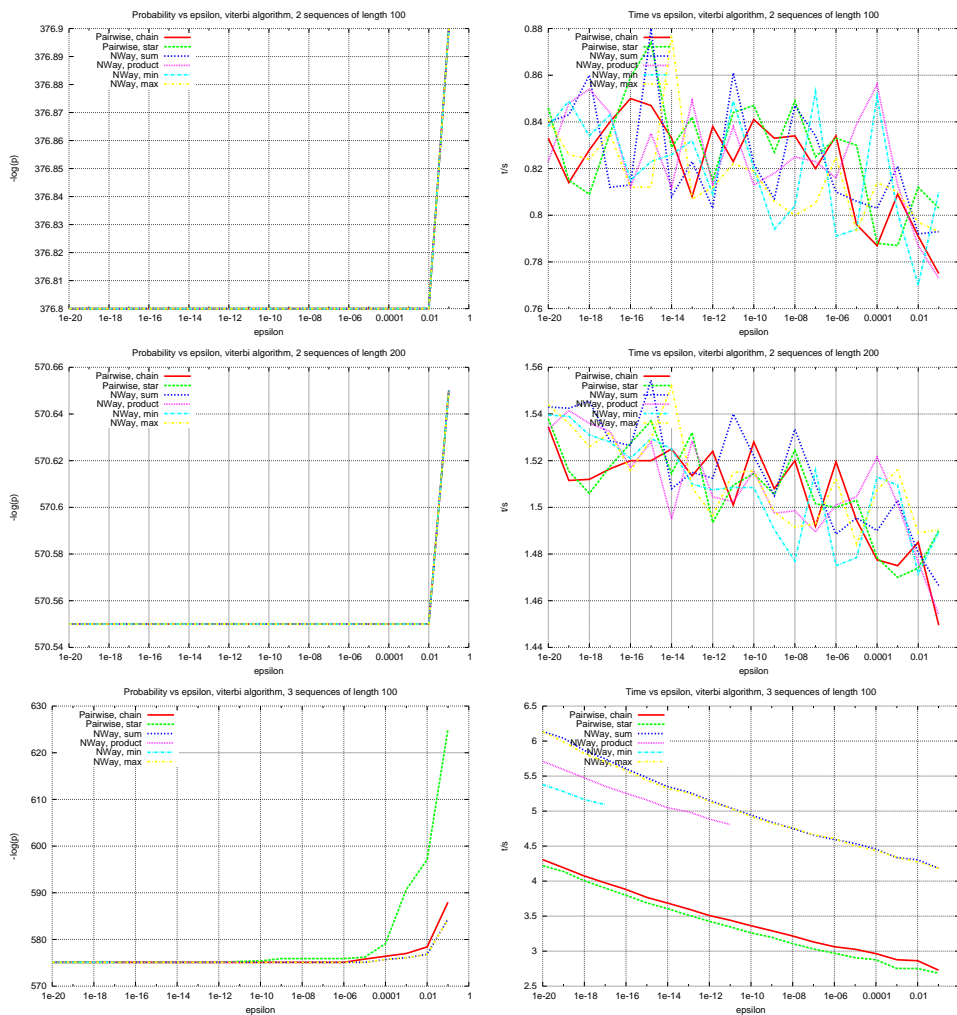
Next we will look at the actual subject of this report: The strategies based on probabilities.

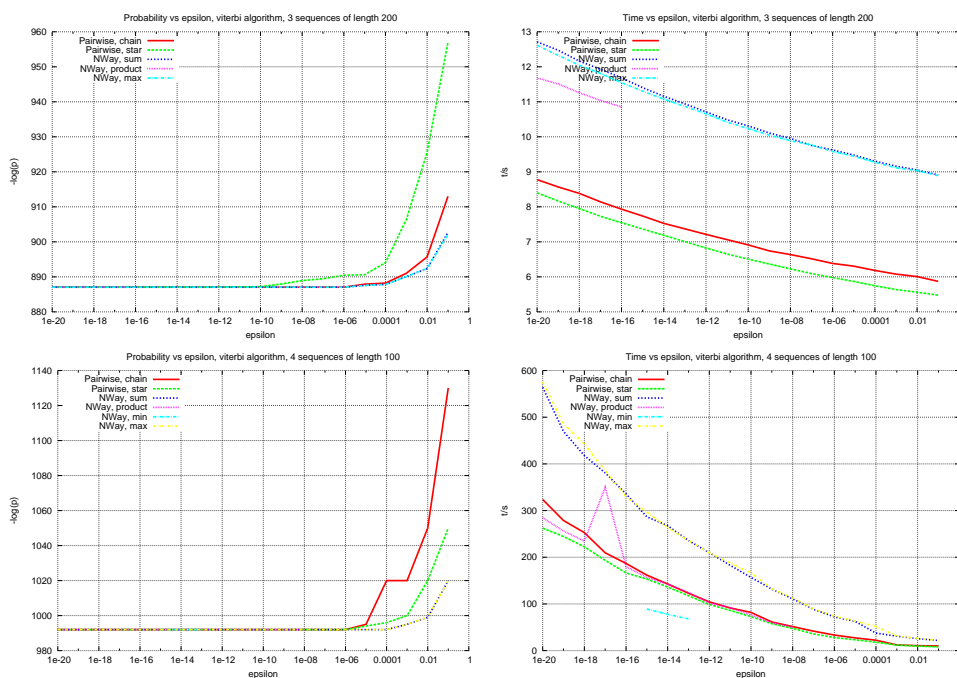


First notice that some strategies do not have a graph or have segments of it missing. This is because that sometimes no unit exist that are able to fulfill the requirements of the strategy so that it will be computed during the alignment. Thus no alignment will be possible and the strategy fails completely. This is especially a problem for the min and product strategies.

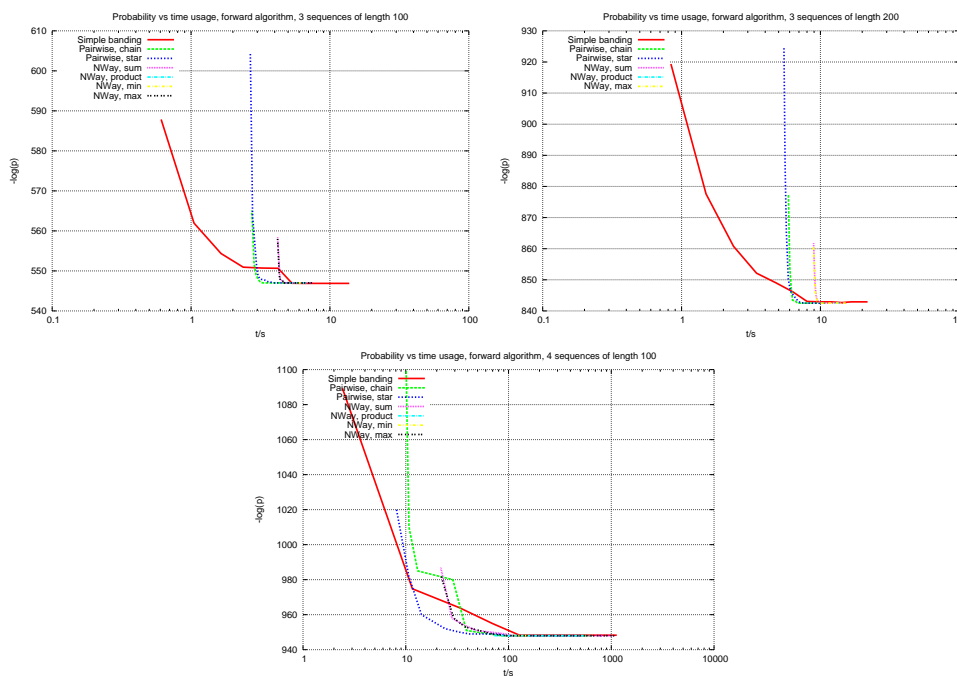
Next we notice that the probabilities are exactly the same for two dimensions. This is due to the fact that the only difference is how to handle more than two sequences and thus they all reduce to the same algorithm. For more sequences we note that the two strategies relying on pairwise alignments require a smaller  $\varepsilon$  to obtain reliable results. This means they have to look at more units in the table which in turn will make them relatively slower. To find the optimal strategy you should find the strategy that gives the best probability and the best time for the corresponding  $\varepsilon$ .

Again the results for the viterbi algorithm are similar and only included for completeness:





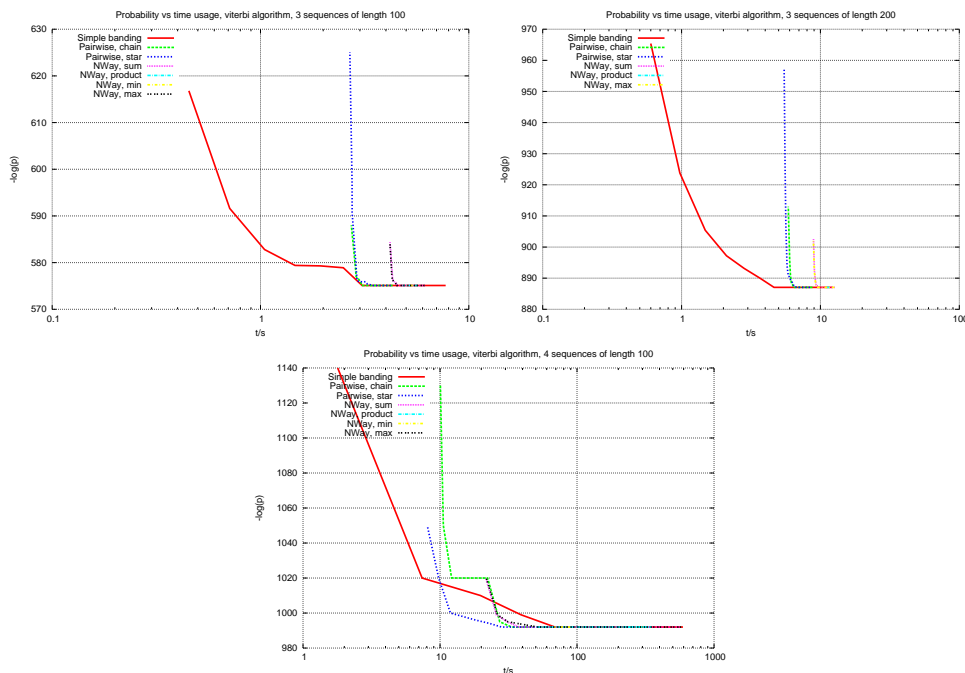
As discussed above what we really want to determine is how much time is necessary to obtain maximum probability. We can do this by combining the above graphs: For different parameters plot the time usage against the probabilities computed:



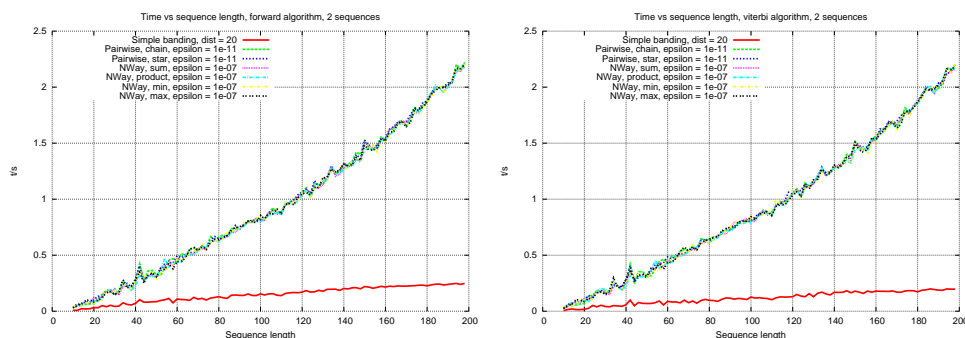
If you look closely at above graphs it turns out that the difference really is not that big. What can be said though is that if you use the max or sum strategy, setting  $\epsilon = 10^{-6}$  seem to be a good choice independtly of the other parameters. If you use any of the other strategies there does not seem to be any single choice for parameter

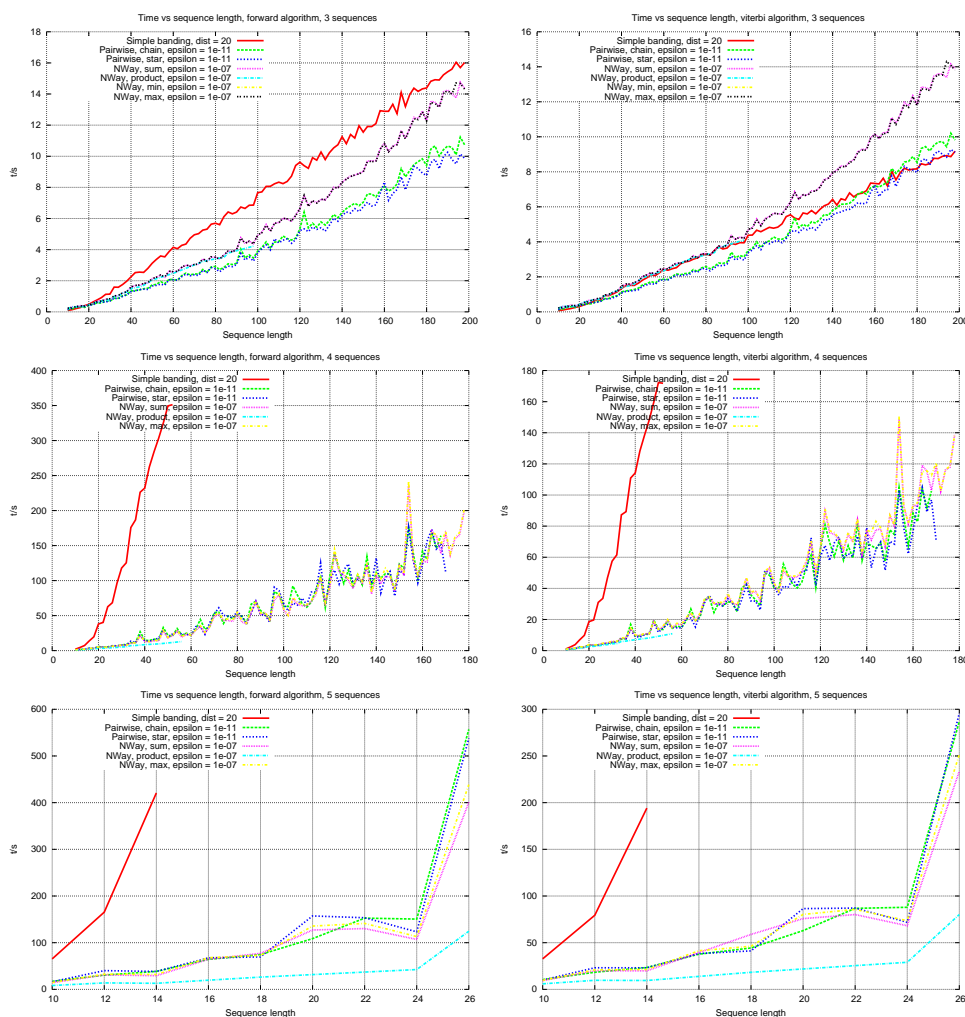
that always give good results. This, in turn, mean that the sum and max strategies seem to be a good choice for practical applications, since the user does not have to spend any time testing different settings and can have confidence in her results.

And, once again, the difference between the forward and viterbi algorithm is negligible:



Finally we did some experiments with higher dimensional alignments. The strategy of no corner cutting were dropped and the rest were only tested with parameters that seemed sensible from above benchmarks. This was done simply to test how the cornercutting scaled to more dimensions.





Experiments in even higher dimensions were attempted but very few actual results were obtained because the machine ran out of memory. What we see is that the naive corner cutting strategy handles poorly for higher dimensions. The min and product strategies also fares very poorly because it is difficult to choose an appropriate  $\epsilon$  for them. You will not see very much of them on above graphs because they fail completely when unable to find any satisfactory units. The rest of the probability based strategies fares similarly, though you have to be aware they will need different *varepsilon* to fare well.

## Litteratur

- [1] I. Holmes. Using guide trees to construct multiple-sequence evolutionary hmms. *Bioinformatics*, 19:i147–i157, 2003.
- [2] I. Holmes and William J. Bruno. Evolutionary hmms: a bayesian approach to multiple alignment. *Bioinformatics*, 17:803–820, 2001.
- [3] Hirohisa Kishino Jeffrey L. Thorne and Joseph Felsenstein. An evolutionary model for maximum likelihood alignment of dna sequences. *Journal of Molecular Evolution*, 33(2):114–124, august 1991.
- [4] J.L. Jensen and J. Hein. Gibbs sampler for statistical multiple alignment. *Statistica Sinica*, 15:889–907, 2005.
- [5] Jens Ledet Jensen Jotun Hein and Christian N. S. Pedersen. Recursions for statistical multiple alignment. *Proceedings of the National Academy of Sciences*, 100(25):14960–14965, december 2003.