Speeding Up HMM Decoding and Training by Exploiting Sequence Repetitions

Shay Mozes⋆[1], Oren Weimann[1], and Michal Ziv-Ukelson[2]

[1] MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA.
shaymozes@gmail.com,oweimann@mit.edu
[2] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel.
michaluz@post.tau.ac.il

Abstract. We present a method to speed up the dynamic program algorithms used for solving the HMM decoding and training problems for discrete time-independent HMMs. We discuss the application of our method to Viterbi's decoding and training algorithms [21], as well as to the forward-backward and Baum-Welch [4] algorithms. Our approach is based on identifying repeated substrings in the observed input sequence. We describe three algorithms based alternatively on byte pair encoding (BPE) [19], run length encoding (RLE) and Lempel-Ziv (LZ78) parsing [22]. Compared to Viterbi's algorithm, we achieve a speedup of $\Omega(r)$ using BPE, a speedup of $\Omega(\frac{r}{\log r})$ using RLE, and a speedup of $\Omega(\frac{\log n}{k})$ using LZ78, where k is the number of hidden states, n is the length of the observed sequence and r is its compression ratio (under each compression scheme). Our experimental results demonstrate that our new algorithms are indeed faster in practice. Furthermore, unlike Viterbi's algorithm, our algorithms are highly parallelizable.

Key words: HMM, Viterbi, dynamic programming, compression

1 Introduction

Over the last few decades, Hidden Markov Models (HMMs) proved to be an extremely useful framework for modeling processes in diverse areas such as error-correction in communication links [21], speech recognition [6], optical character recognition [2], computational linguistics [17], and bioinformatics [12].

The core HMM-based applications fall in the domain of classification methods and are technically divided into two stages: a training stage and a decoding stage. During the *training* stage, the emission and transition probabilities of an HMM are estimated, based on an input set of observed sequences. This stage is usually executed once as a preprocessing stage and the generated ("trained") models are stored in a database. Then, a *decoding* stage is run, again and again, in order to classify input sequences. The objective of this stage is to find the most probable sequence of states to have generated each input sequence given each model, as illustrated in Fig. 1.

⋆ Work conducted while visiting MIT

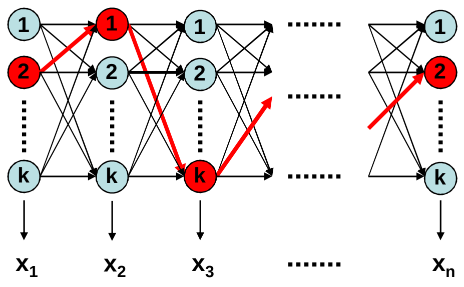

Fig. 1. The HMM on the observed sequence $X = x_1, x_2, \ldots, x_n$ and states $1, 2, \ldots, k$. The highlighted path is a possible path of states that generate the observed sequence. VA finds the path with highest probability.

Obviously, the training problem is more difficult to solve than the decoding problem. However, the techniques used for decoding serve as basic ingredients in solving the training problem. The Viterbi algorithm (VA) [21] is the best known tool for solving the decoding problem. Following its invention in 1967, several other algorithms have been devised for the decoding and training problems, such as the forward-backward and Baum-Welch [4] algorithms. These algorithms are all based on dynamic programs whose running times depend linearly on the length of the observed sequence. The challenge of speeding up VA by utilizing HMM topology was posed in 1997 by Buchsbaum and Giancarlo [6] as a major open problem. In this contribution, we address this open problem by using text compression and present the first provable speedup of these algorithms.

The traditional aim of text compression is the efficient use of resources such as storage and bandwidth. Here, however, we compress the observed sequences in order to speed up HMM algorithms. This approach, denoted "acceleration by text-compression", was previously applied to some classical problems on strings. Various compression schemes, such as LZ77, LZW-LZ78, Huffman coding, Byte Pair Encoding (BPE) and Run Length Encoding (RLE), were employed to accelerate exact and approximate pattern matching [14,16,19,1,13,18] and sequence alignment [3,7,11,15]. In light of the practical importance of HMM-based classification methods in state-of-the-art research, and in view of the fact that such techniques are also based on dynamic programming, we set out to answer the following question: can "acceleration by text compression" be applied to HMM decoding and training algorithms?

Our results. Let X denote the input sequence and let n denote its length. Let k denote the number of states in the HMM. For any given compression scheme, let n' denote the number of parsed blocks in X and let $r = n/n'$ denote the compression ratio. Our results are as follows.

1. BPE is used to accelerate decoding by a factor of $\Omega(r)$.
2. RLE is used to accelerate decoding by a factor of $\Omega(\frac{r}{logr})$.

3. Using LZ78, we accelerate decoding by a factor of $\Omega(\frac{\log n}{k})$. Our algorithm guarantees no degradation in efficiency even when $k > \log n$ and is experimentally more than five times faster than VA.
4. The same speedup factors apply to the Viterbi training algorithm.
5. For the Baum-Welch training algorithm, we show how to preprocess a repeated substring of size ℓ once in $O(\ell k^4)$ time so that we may replace the usual $O(\ell k^2)$ processing work for each occurrence of this substring with an alternative $O(k^4)$ computation. This is beneficial for any repeat with λ non-overlapping occurrences, such that $\lambda > \frac{\ell k^2}{\ell - k^2}$.
6. As opposed to VA, our algorithms are highly parallelizable. This is discussed in the full version of this paper.

Roadmap. The rest of the paper is organized as follows. In section 2 we give a unified presentation of the HMM dynamic programs. We then show in section 3 how these algorithms can be improved by identifying repeated substrings. Two compressed decoding algorithms are given in sections 4 and 5. In section 6 we show how to adapt the algorithms to the training problem. Finally, experimental results are presented in Section 7.

2 Preliminaries

Let Σ denote a finite alphabet and let $X \in \Sigma^n$, $X = x_1, x_2, \ldots, x_n$ be a sequence of observed letters. A Markov *model* is a set of k states, along with emission probabilities $e_k(\sigma)$ - the probability to observe $\sigma \in \Sigma$ given that the state is k, and transition probabilities $P_{i,j}$ - the probability to make a transition to state i from state j.

The Viterbi Algorithm. The Viterbi algorithm (VA) finds the most probable sequence of hidden states given the model and the observed sequence. i.e., the sequence of states $s_1, s_2, \ldots, s_n$ which maximize

$$\prod_{i=1}^{n} e_{s_i}(x_i) P_{s_i, s_{i-1}} \tag{1}$$

The dynamic program of VA calculates a vector $v_t[i]$ which is the probability of the most probable sequence of states emitting $x_1, \ldots, x_t$ and ending with the state i at time t. v_0 is usually taken to be the vector of uniform probabilities (i.e., $v_0[i] = \frac{1}{k}$). v_{t+1} is calculated from v_t according to

$$v_{t+1}[i] = e_i(x_{t+1}) \cdot \max_j \{P_{i,j} \cdot v_t[j]\} \tag{2}$$

Definition 1 (Viterbi Step). *We call the computation of v_{t+1} from v_t a Viterbi step.*

Clearly, each Viterbi step requires $O(k^2)$ time. Therefore, the total runtime required to compute the vector v_n is $O(nk^2)$. The probability of the most likely sequence of states is the maximal element in v_n. The actual sequence of states can be then reconstructed in linear time.

It is useful for our purposes to rewrite VA in a slightly different way. Let M^σ be a $k \times k$ matrix with elements $M^\sigma_{i,j} = e_i(\sigma) \cdot P_{i,j}$. We can now express v_n as:

$$v_n = M^{x_n} \odot M^{x_{n-1}} \odot \cdots \odot M^{x_2} \odot M^{x_1} \odot v_0 \tag{3}$$

where $(A \odot B)_{i,j} = \max_k\{A_{i,k} \cdot B_{k,j}\}$ is the so called max-times matrix multiplication. VA computes v_n using (3) from right to left in $O(nk^2)$ time. Notice that if (3) is evaluated from left to right the computation would take $O(nk^3)$ time (matrix-vector multiplication vs. matrix-matrix multiplication). Throughout, we assume that the max-times matrix-matrix multiplications are done naïvely in $O(k^3)$. Faster methods for max-times matrix multiplication [8] and standard matrix multiplication [20,10] can be used to reduce the k^3 term. However, for small values of k this is not profitable.

The Forward-Backward Algorithms. The *forward-backward* algorithms are closely related to VA and are based on very similar dynamic programs. In contrast to VA, these algorithms apply standard matrix multiplication instead of max-times multiplication. The forward algorithm calculates $f_t[i]$, the probability to observe the sequence $x_1, x_2, \ldots, x_t$ requiring that $s_t = i$ as follows:

$$f_t = M^{x_t} \cdot M^{x_{t-1}} \cdot \cdots \cdot M^{x_2} \cdot M^{x_1} \cdot f_0 \tag{4}$$

The backward algorithm calculates $b_t[i]$, the probability to observe the sequence $x_{t+1}, x_{t+2}, \ldots, x_n$ given that $s_t = i$ as follows:

$$b_t = b_n \cdot M^{x_n} \cdot M^{x_{n-1}} \cdot \cdots \cdot M^{x_{t+2}} \cdot M^{x_{t+1}} \tag{5}$$

Another algorithm which is used in the training stage and employs the forward-backward algorithm as a subroutine, is the Baum-Welch algorithm, to be further discussed in Section 6.

A motivating example. We briefly describe one concrete example from computational biology to which our algorithms naturally apply. CpG islands [5] are regions of DNA with a large concentration of the nucleotide pair CG. These regions are typically a few hundred to a few thousand nucleotides long, located around the promoters of many genes. As such, they are useful landmarks for the identification of genes. The observed sequence (X) is a long DNA sequence composed of four possible nucleotides $(\Sigma = \{A, C, G, T\})$. The length of this sequence is typically a few millions nucleotides $(n \simeq 2^{25})$. A well-studied classification problem is that of parsing a given DNA sequence into CpG islands and non CpG regions. Previous work on CpG island classification used Markov models with either 8 or 2 states ($k = 8$ or $k = 2$) [9,12].

3 Exploiting Repeated Substrings in the Decoding Stage

Consider a substring $W = w_1, w_2, \ldots, w_\ell$ of X, and define

$$M(W) = M^{w_\ell} \odot M^{w_{\ell-1}} \odot \cdots \odot M^{w_2} \odot M^{w_1} \tag{6}$$

Intuitively, $M_{i,j}(W)$ is the probability of the most likely path starting with state j, making a transition into some other state, emitting w_1, then making a transition into yet another state and emitting w_2 and so on until making a final transition into state i and emitting w_ℓ.

In the core of our method stands the following observation, which is immediate from the associative nature of matrix multiplication.

Observation 1. *We may replace any occurrence of $M^{w_\ell} \odot M^{w_{\ell-1}} \odot \cdots \odot M^{w_1}$ in eq.* (3) *with $M(W)$.*

The application of observation 1 to the computation of equation (3) saves $\ell - 1$ Viterbi steps *each* time W appears in X, but incurs the additional cost of computing $M(W)$ once.

An intuitive exercise. Let λ denote the number of times a given word W appears, in non-overlapping occurrences, in the input string X. Suppose we naïvely compute $M(W)$ using $(|W| - 1)$ max-times matrix multiplications, and then apply observation 1 to all occurrences of W before running VA. We gain some speedup in doing so if

$$\begin{aligned}(|W| - 1)k^3 + \lambda k^2 &< \lambda |W| k^2 \\ \lambda &> k\end{aligned} \tag{7}$$

Hence, if there are at least k non-overlapping occurrences of W in the input sequence, then it is worthwhile to naïvely precompute $M(W)$, regardless of it's size $|W|$.

Definition 2 (Good Substring). *We call a substring W good if we decide to compute $M(W)$.*

We can now give a general four-step framework of our method:

(I) *Dictionary Selection:* choose the set $D = \{W_i\}$ of good substrings.
(II) *Encoding:* precompute the matrices $M(W_i)$ for every $W_i \in D$.
(III) *Parsing:* partition the input sequence X into consecutive good substrings $X = W_{i_1} W_{i_2} \cdots W_{i_{n''}}$ and let X' denote the compressed representation of this parsing of X, such that $X' = i_1 i_2 \cdots i_{n''}$.
(IV) *Propagation:* run VA on X', using the matrices $M(W_i)$.

The above framework introduces the challenge of how to select the set of good substrings (step I) and how to efficiently compute their matrices (step II). In the next two sections we show how the RLE and LZ78 compression schemes can be applied to address this challenge. The utilization of the BPE compression scheme is discussed in the full version of this paper. Another challenge is how to parse the sequence X (step III) in order to maximize acceleration. We show that, surprisingly, this optimal parsing may differ from the initial parsing induced by the selected compression scheme. To our knowledge, this feature was not applied by previous "acceleration by compression" algorithms.

Throughout this paper we focus on computing path probabilities rather than the paths themselves. The actual paths can be reconstructed in linear time as described in the full version of this paper.

4 Acceleration via Run-length Encoding

In this section we obtain an $\Omega(\frac{r}{logr})$ speedup for decoding an observed sequence with run-length compression ratio r. A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i". Each pair corresponds to a *run* in S, consisting of i consecutive occurrences of the character σ. For example, the string *aaabbcccccc* is encoded as $a^3b^2c^6$. Run-length encoding serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels. The four-step framework described in section 3 is applied as follows.

(I) *Dictionary Selection:* for every $\sigma \in \Sigma$ and every $i = 1, 2, \ldots, \log n$ we choose σ^{2^i} as a *good substring.*

(II) *Encoding:* since $M(\sigma^{2^i}) = M(\sigma^{2^{i-1}}) \odot M(\sigma^{2^{i-1}})$, we can compute the matrices using repeated squaring.

(III) *Parsing:* Let $W_1W_2\cdots W_{n'}$ be the RLE of X, where each W_i is a run of some $\sigma \in \Sigma$. X' is obtained by further parsing each W_i into at most $\log|W_i|$ good substrings of the form σ^{2^j}.

(IV) *Propagation:* run VA on X', as described in Section 3.

Time and Space Complexity Analysis. The offline preprocessing stage consists of steps I and II. The time complexity of step II is $O(|\Sigma|k^3 \log n)$ by applying max-times repeated squaring in $O(k^3)$ time per multiplication. The space complexity is $O(|\Sigma|k^2 \log n)$. This work is done offline once, during the training stage, in advance for all sequences to come. Furthermore, for typical applications, the $O(|\Sigma|k^3 \log n)$ term is much smaller than the $O(nk^2)$ term of VA.

Steps III and IV both apply one operation per occurrence of a good substring in X': step III computes, in constant time, the index of the next parsing-comma, and step IV applies a single Viterbi step in k^2 time. Since $|X'| = \sum_{i=1}^{n'} log|W_i|$, the complexity is

$$\sum_{i=1}^{n'} k^2 log|W_i| = k^2 log(|W_1| \cdot |W_2| \cdots |W_{n'}|) \leq k^2 log((n/n')^{n'}) = O(n'k^2 log\frac{n}{n'}).$$

Thus, the speedup compared to the $O(nk^2)$ time of VA is $\Omega(\frac{\frac{n}{n'}}{log\frac{n}{n'}}) = \Omega(\frac{r}{logr})$.

5 Acceleration via LZ78 Parsing

In this section we obtain an $\Omega(\frac{\log n}{k})$ speedup for decoding, and a constant speedup in the case where $k > \log n$. We show how to use the LZ78 [22] (henceforth LZ) parsing to find good substrings and how to use the incremental nature of the LZ parse to compute $M(W)$ for a good substring W in $O(k^3)$ time.

LZ parses the string X into substrings (LZ-words) in a single pass over X. Each LZ-word is composed of the longest LZ-word previously seen plus a single letter. More formally, LZ begins with an empty dictionary and parses according to the following rule: when parsing location i, look for the longest LZ-word W starting at position i which already appears in the dictionary. Read one more letter σ and insert $W\sigma$ into the dictionary. Continue parsing from position $i + |W| + 1$. For example, the string "AACGACG" is parsed into four words: A, AC, G, ACG. Asymptotically, LZ parses a string of length n into $O(hn/\log n)$ words [22], where $0 \leq h \leq 1$ is the entropy of the string. The LZ parse is performed in linear time by maintaining the dictionary in a trie. Each node in the trie corresponds to an LZ-word. The four-step framework described in section 3 is applied as follows.

(I) *Dictionary Selection:* the good substrings are all the LZ-words in the LZ-parse of X.
(II) *Encoding:* construct the matrices incrementally, according to their order in the LZ-trie, $M(W\sigma) = M(W) \odot M^{\sigma}$.
(III) *Parsing:* X' is the LZ-parsing of X.
(IV) *Propagation:* run VA on X', as described in section 3.

Time and Space Complexity Analysis. Steps I and III were already conducted offline during the pre-processing compression of the input sequences (in any case LZ parsing is linear). In step II, computing $M(W\sigma) = M(W) \odot M^{\sigma}$, takes $O(k^3)$ time since $M(W)$ was already computed for the good substring W. Since there are $O(n/\log n)$ LZ-words, calculating the matrices $M(W)$ for all Ws takes $O(k^3 n/\log n)$. Running VA on X' (step IV) takes just $O(k^2 n/\log n)$ time. Therefore, the overall runtime is dominated by $O(k^3 n/\log n)$. The space complexity is $O(k^2 n/\log n)$.

The above algorithm is useful in many applications, such as CpG island classification, where $k < \log n$. However, in those applications where $k > \log n$ such an algorithm may actually slow down VA.

We next show an adaptive variant that is guaranteed to speed up VA, regardless of the values of n and k. This graceful degradation retains the asymptotic $\Omega(\frac{\log n}{k})$ acceleration when $k < \log n$.

5.1 An improved algorithm

Recall that given $M(W)$ for a good substring W, it takes k^3 time to calculate $M(W\sigma)$. This calculation saves k^2 operations each time $W\sigma$ occurs in X in comparison to the situation where only $M(W)$ is computed. Therefore, in step

I we should include in D, as good substrings, only words that appear as a prefix of at least k LZ-words. Finding these words can be done in a single traversal of the trie. The following observation is immediate from the prefix monotonicity of occurrence tries.

Observation 2. *Words that appear as a prefix of at least k LZ-words are represented by trie nodes whose subtrees contain at least k nodes.*

In the previous case it was straightforward to transform X into X', since each phrase p in the parsed sequence corresponded to a good substring. Now, however, X does not divide into just good substrings and it is unclear what is the optimal way to construct X' (in step III). Our approach for constructing X' is to first parse X into all LZ-words and then apply the following greedy parsing to each LZ-word W: using the trie, find the longest good substring $w' \in D$ that is a prefix of W, place a parsing comma immediately after w' and repeat the process for the remainder of W.

Time and Space Complexity Analysis. The improved algorithm utilizes substrings that guarantee acceleration (with respect to VA) so it is therefore faster than VA even when $k = \Omega(\log n)$. In addition, in spite of the fact that this algorithm re-parses the original LZ partition, the algorithm still guarantees an $\Omega(\frac{\log n}{k})$ speedup over VA as shown by the following lemma.

Lemma 1. *The running time of the above algorithm is bounded by $O(k^3 n/\log n)$.*

Proof. The running time of step II is at most $O(k^3 n/\log n)$. This is because the size of the entire LZ-trie is $O(n/\log n)$ and we construct the matrices, in $O(k^3)$ time each, for just a subset of the trie nodes. The running time of step IV depends on the number of new phrases (commas) that result from the re-parsing of each LZ-word W. We next prove that this number is at most k for each word.

Consider the first iteration of the greedy procedure on some LZ-word W. Let w' be the longest prefix of W that is represented by a trie node with at least k descendants. Assume, contrary to fact, that $|W| - |w'| > k$. This means that w'', the child of w', satisfies $|W| - |w''| \geq k$, in contradiction to the definition of w'. We have established that $|W| - |w'| \leq k$ and therefore the number of re-parsed words is bounded by $k+1$. The propagation step IV thus takes $O(k^3)$ time for each one of the $O(n/\log n)$ LZ-words. So the total time complexity remains $O(k^3 n/\log n)$. □

Based on Lemma 1, and assuming that steps I and III are pre-computed offline, the running time of the above algorithm is $O(nk^2/e)$ where $e = \Omega(\max(1, \frac{\log n}{k}))$. The space complexity is $O(k^2 n/logn)$.

6 The Training Problem

In the training problem we are given as input the number of states in the HMM and an observed training sequence X. The aim is to find a set of model parameters θ (i.e., the emission and transition probabilities) that maximize the likelihood to observe the given sequence $P(X|\,\theta)$. The most commonly used training algorithms for HMMs are based on the concept of Expectation Maximization. This is an iterative process in which each iteration is composed of two steps. The first step solves the decoding problem given the current model parameters. The second step uses the results of the decoding process to update the model parameters. These iterative processes are guaranteed to converge to a local maximum. It is important to note that since the dictionary selection step (I) and the parsing step (III) of our algorithm are independent of the model parameters, we only need run them once, and repeat just the encoding step (II) and the propagation step (IV) when the decoding process is performed in each iteration.

6.1 Viterbi training

The first step of Viterbi training [12] uses VA to find the most likely sequence of states given the current set of parameters (i.e., decoding). Let A_{ij} denote the number of times the state i follows the state j in the most likely sequence of states. Similarly, let $E_i(\sigma)$ denote the number of times the letter σ is emitted by the state i in the most likely sequence. The updated parameters are given by:

$$P_{ij} = \frac{A_{ij}}{\sum_{i'} A_{i'j}} \text{ and } e_i(\sigma) = \frac{E_i(\sigma)}{\sum_{\sigma'} E_i(\sigma')} \tag{8}$$

Note that the Viterbi training algorithm does not converge to the set of parameters that maximizes the likelihood to observe the given sequence $P(X|\,\theta)$, but rather the set of parameters that locally maximizes the contribution to the likelihood from the most probable sequence of states [12]. It is easy to see that the time complexity of each Viterbi training iteration is $O(k^2n+n) = O(k^2n)$ so it is dominated by the running time of VA. Therefore, we can immediately apply our compressed decoding algorithms from sections 4 and 5 to obtain a better running time per iteration.

6.2 Baum-Welch training

The Baum-Welch training algorithm [4,12] converges to a set of parameters that locally maximize the likelihood to observe the given sequence $P(X|\,\theta)$, and is the most commonly used method for model training. We give here a brief explanation of the algorithm and of our acceleration approach. The complete details appear in the full version of this paper.

Recall the forward-backward matrices: $f_t[i]$ is the probability to observe the sequence $x_1, x_2, \ldots, x_t$ requiring that the t'th state is i and that $b_t[i]$ is the probability to observe the sequence $x_{t+1}, x_{t+2}, \ldots, x_n$ given that the t'th state is i. The first step of Baum-Welch calculates $f_t[i]$ and $b_t[i]$ for every $1 \leq t \leq n$ and every $1 \leq i \leq k$. This is achieved by applying the forward and backward algorithms to the input data in $O(nk^2)$ time (see eqs. (4) and (5)). The second step recalculates A and E according to

$$\begin{aligned} A_{i,j} &= \sum_t P(s_t = j, s_{t+1} = i | X, \theta) \\ E_i(\sigma) &= \sum_{t | x_t = \sigma} P(s_t = i | X, \theta) \end{aligned} \tag{9}$$

where $P(s_t = j, s_{t+1} = i|X, \theta)$ is the probability that a transition from state j to state i occurred in position t in the sequence X, and $P(s_t = i|X, \theta)$ is the probability for the t'th state to be i in the sequence X. These quantities are given by:

$$P(s_t = j, s_{t+1} = i|X, \theta) = \frac{f_t[j] \cdot P_{i,j} \cdot e_i(x_{t+1}) \cdot b_{t+1}[i]}{\sum_i f_n[i]} \tag{10}$$

and

$$P(s_t = i|X, \theta) = \frac{f_t[i] \cdot b_t[i]}{\sum_i f_n[i]}. \tag{11}$$

Finally, after the matrices A and E are recalculated, Baum-Welch updates the model parameters according to equation (8).

We next describe how to accelerate the Baum-Welch algorithm. Note that in the first step of Baum-Welch, our algorithms to accelerate VA (Sections 4 and 5) can be used to accelerate the forward-backward algorithms by replacing the max-times matrix multiplication with regular matrix multiplication. However, the accelerated algorithms only compute f_t and b_t on the boundaries of good substrings. In order to solve this problem and speed up the second step of Baum-Welch as well, we observe that when accumulating the contribution of some appearance of a good substring W of length $|W| = \ell$ to A, Baum-Welch performs $O(\ell k^2)$ operations, but updates at most k^2 entries (the size of A). Hence, it is possible to obtain a speedup by precalculating the contribution of each good substring to A and E. For brevity the details are omitted here and will appear in the full version of this paper. To summarize the results, preprocessing a good substring W requires $O(\ell k^4)$ time and $O(k^4)$ space. Using the preprocessed information and the values of f_t and b_t on the boundaries of good substrings, we can update A and E in $O(k^4)$ time per good substring (instead of ℓk^2). To get a speedup we need λ, the number of times the good substring W appears in X to satisfy:

$$\begin{aligned} \ell k^4 + \lambda k^4 &< \lambda \ell k^2 \\ \lambda &> \frac{\ell k^2}{\ell - k^2} \end{aligned} \tag{12}$$

This is reasonable if k is small. If $\ell = 2k^2$, for example, then we need λ to be greater than $2k^2$. In the CpG islands problem, if $k = 2$ then any substrings of length eight is good if it appears more than eight times in the text.

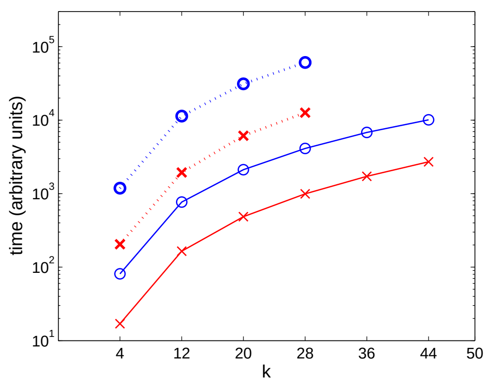

Fig. 2. Comparison of the cumulative running time of steps II and IV of our algorithm (marked x) with the running time of VA (marked o), for different values of k. Time is shown in arbitrary units on a logarithmic scale. Runs on the 1.5Mbp chromosome 4 of S. cerevisiae are in solid lines. Runs on the 22Mbp human Y-chromosome are in dotted lines. The roughly uniform difference between corresponding pairs of curves reflects a speedup factor of more than five.

7 Experimental Results

We implemented both a variant of our improved LZ-compressed algorithm from subsection 5.1 and classical VA in C++ and compared their execution times on a sequence of approximately 22,000,000 nucleotides from the human Y chromosome and on a sequence of approximately 1,500,000 nucleotides from chromosome 4 of S. Cerevisiae obtained from the UCSC genome database. The benchmarks were performed on a single processor of a SunFire V880 server with 8 UltraSPARC-IV processors and 16GB main memory. The implementation is just for calculating the probability of the most likely sequence of states, and does not traceback the optimal sequence itself. As we have seen, this is the time consuming part of the algorithm. We measured the running times for different values of k. As we explained in the previous sections we are only interested in the running time of the encoding and the propagation steps (II and IV) since the combined parsing/dictionary-selections steps (I and III) may be performed in advance and are not repeated by the training and decoding algorithms. The results are shown in Fig. 2. Our algorithm performs faster than VA even for surprisingly large values of k. For example, for $k = 60$ our algorithm is roughly three times faster than VA.

References

1. G. Benson A. Amir and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Comp. and Sys. Sciences*, 52(2):299–307, 1996.

2. O. Agazzi and S. Kuo. HMM based optical character recognition in the presence of deterministic transformations. *Pattern recognition*, 26:1813–1826, 1993.
3. A. Apostolico, G.M. Landau, and S. Skiena. Matching for run length encoded strings. *Journal of Complexity*, 15:1:4–16, 1999.
4. L.E. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of a Markov process. *Inequalities*, 3:1–8, 1972.
5. A.P. Bird. Cpg-rich islands as gene markers in the vertebrate nucleus. *Trends in Genetics*, 3:342–347, 1987.
6. A. L. Buchsbaum and R. Giancarlo. Algorithmic aspects in speech recognition: An introduction. *ACM Journal of Experimental Algorithms*, 2:1, 1997.
7. H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings. *Information Processing Letters*, 54:93–96, 1995.
8. T.M. Chan. All-pairs shortest paths with real weights in $O(n^3/logn)$ time. In *Proc. 9th Workshop on Algorithms and Data Structures*, pages 318–324, 2005.
9. G.A. Churchill. Hidden Markov chains and the analysis of genome structure. *Computers Chem.*, 16:107–115, 1992.
10. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetical progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
11. M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proc. 13th Annual ACMSIAM Symposium on Discrete Algorithms*, pages 679–688, 2002.
12. R. Durbin, S. Eddy, A. Krigh, and G. Mitcheson. *Biological Sequence Analysis.* Cambridge University Press, 1998.
13. J. Karkkainen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. *Proc. 11th Annual Symposium On Combinatorial Pattern Matching (CPM)*, LNCS 1848:195–209, 2000.
14. J. Karkkainen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. *Proc. Third South American Workshop on String Processing (WSP)*, pages 141–155, 1996.
15. V. Makinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. *Proc. 12th Annual Symposium On Combinatorial Pattern Matching (CPM)*, LNCS 1645:1–13, 1999.
16. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *Proc. 5th Annual Symposium On Combinatorial Pattern Matching (CPM)*, LNCS 2089:31–49, 2001.
17. C. Manning and H. Schutze. *Statistical Natural Language Processing.* The MIT Press, 1999.
18. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. *Proc. Data Compression Conference (DCC)*, pages 459–468, 2001.
19. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. *Lecture Notes in Computer Science*, 1767:306–315, 2000.
20. V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
21. A. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, 1967.
22. J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.