In Proceedings of
Third South American Workshop
on String Processing
Recife, Brazil, August 1996
Carlton University Press

Approximate String Searching under Weighted Edit Distance

Stefan Kurtz

Universität Bielefeld, Technische Fakultät,
Postfach 100 131, 33501 Bielefeld, Germany,
E-mail: kurtz@techfak.uni-bielefeld.de

Abstract. Let $p \in \Sigma^*$ be a string of length m and $t \in \Sigma^*$ be a string of length n. The approximate string searching problem is to find all approximate matches of p in t having weighted edit distance at most k from p. We present a new method that preprocesses the pattern into a DFA which scans t online in linear time, thereby recognizing all positions in t where an approximate match ends. We show how to reduce the exponential preprocessing effort and propose two practical algorithms. The first algorithm constructs the states of the DFA up to a certain depth $r \geq 1$. It runs in $\mathcal{O}(|\Sigma|^{r+1} \cdot m + q \cdot m + n)$ time and $\mathcal{O}(|\Sigma|^{r+1} + |\Sigma|^r \cdot m)$ space where $q \leq n$ decreases as r increases. The second algorithm constructs the transitions of the DFA when they are demanded. It runs in $\mathcal{O}(q_s \cdot |\Sigma| + q_t \cdot m + n)$ time and $\mathcal{O}(q_s \cdot (|\Sigma| + m))$ space where $q_s \leq q_t \leq n$ depend on the problem instance. Practical measurements show that our algorithms work well in practice and beat previous methods for problems of interest in molecular biology.

1 Introduction

We consider the approximate string searching problem. Given a pattern $p \in \Sigma^*$ of length m and an input string $t \in \Sigma^*$ of length n, it consists of finding the positions in t where an approximate match ends. These positions are referred to as solutions of the approximate string searching problem. An approximate match is a subword of t whose distance to p is at most k, for a given threshold value $k \in \mathbb{R}^+$. The distance is measured in terms of edit operations, that is, deletions, insertions, and replacements of single characters. Edit operations are weighted according to a given weight function δ.

The approximate string searching problem is of special interest in biological sequence analysis. For instance, when searching a DNA database (the input string) for a query (the pattern), a small but significant error must be allowed, to take into account experimental inaccuracies as well as small differences in DNA among individuals of the same or related species. Note that in biological context, the weight function plays an important role. It provides a simple way to consider knowledge about biological phenomena on the nucleotide level. That is, by choosing an appropriate weight function, one can select those matches which make biological sense, and reject others which do not.

So far, computer scientists have mainly focused on the k-differences problem [17, 9, 3, 11, 4]. This is the approximate string searching problem, restricted to the unit weight function (each edit operation has weight 1). From the view of applications in biological sequence analysis, however, arbitrary weight functions are definitely required, so that the k-differences problem must be considered a "toy" problem [12].

For the general case, Sellers [16] has developed an algorithm that evaluates for each character in t a "distance column" of $m+1$ entries. If the last entry in the jth distance column is at most k, then an approximate match ending at position j in t is found. This solves the approximate string searching problem in $\mathcal{O}(m \cdot n)$ time and $\mathcal{O}(m)$ space. Sellers' method has been adopted to solve generalizations of the approximate string searching problem, where the pattern is a regular expression [14] or a network expression with spacers [13].

When p is a simple string, Ukkonen [17] has observed that in each distance column it suffices to compute the essential entries, that is, those entries which are at most k. This observation leads to a considerable improvement of Sellers' algorithm. Moreover, it is the point of reference from where further improvements can proceed in two ways.

1.1 Previous Improvements by Preprocessing the Input String

In [18, 5] algorithms are developed which assume that t is fixed and preprocessed into a suffix tree, denoted $stree(t)$ in the following. In particular, Ukkonen [18] describes three algorithms with running times $\mathcal{O}(m \cdot q + n)$, $\mathcal{O}(m \cdot q \cdot \log q + t_{occ})$, and $\mathcal{O}(m^2 \cdot q + t_{occ})$ where $q \leq n$ varies depending on the problem instance, and t_{occ} is the number of positions in t where an approximate match occurs. Cobbs [5] has given an improved algorithm which runs in $\mathcal{O}(m \cdot q + t_{occ})$ time, for the same value of q as above. All results are for a fixed and finite alphabet. Note that for the case that t is not fixed, an additional summand n must be added to each of the $\mathcal{O}$-terms. This summand results from constructing $stree(t)$ in $\mathcal{O}(n)$.

1.2 A New Method Based on Preprocessing the Pattern

Our new method does not require t to be preprocessed. Instead, it preprocesses Σ, δ, k, and p into a deterministic finite automaton (DFA for short) which recognizes shortest essential suffixes, i.e., strings determining the essential entries of a distance column. The DFA scans t online in $\mathcal{O}(n)$ time, thereby recognizing all positions where an approximate match ends. For each such position it additionally computes an approximate match ending at that position. This also holds for the algorithms in [18, 5], but not for Sellers' method.

As well known, a DFA runs in $\mathcal{O}(n)$ time with a very small constant factor. So w.r.t. the scanning phase, our method is certainly the fastest that has been achieved under general weight functions. However, the price to pay is the construction of the DFA. Precomputing it completely requires $\mathcal{O}(|\Sigma|^{m+k+1} \cdot m)$ time and $\mathcal{O}(|\Sigma|^{m+k+1} + m^2)$ space in the worst case. This becomes infeasible already for small m and k. The whole point of our approach are two algorithms that

reduce preprocessing effort, while still retaining efficient search for approximate matches: The first algorithm constructs the states of the DFA up to a certain depth $r \geq 1$. It runs in $\mathcal{O}(|\Sigma|^{r+1} \cdot m + q \cdot m + n)$ time and $\mathcal{O}(|\Sigma|^{r+1} + |\Sigma|^r \cdot m)$ space where $q \leq n$ decreases as r increases. Thus r is a parameter that allows to balance preprocessing versus search efficiency. The second algorithm applies the lazy evaluation technique already used in the implementation of *egrep* (see [2, Section 3.7]). It constructs the transitions of the DFA on-the-fly, i.e., when they are demanded. This leads to a running time of $\mathcal{O}(q_s \cdot |\Sigma| + q_t \cdot m + n)$ and a space consumption of $\mathcal{O}(q_s \cdot (|\Sigma| + m))$ where $q_s \leq q_t \leq n$ depend on the problem instance.

In comparison to the algorithms of Ukkonen [18] and Cobbs [5], our algorithms have two important advantages. First, they use simpler data structures and are therefore easier to implement. Second, they are applicable to very large input strings. This is not true for the suffix tree based methods, as already remarked in [18]. Practical measurements show that our algorithms work well in practice and beat previous online algorithms for problems of interest in molecular biology.

1.3 Overview

The paper is organized as follows. Section 2 recalls some basic definitions and notations. In Section 3 we take some time to carefully describe the basic techniques for solving the approximate string searching problem. Section 4 recalls the basic properties of essential suffixes. In Section 5, we present our basic method, including techniques to reduce preprocessing effort. Section 6 gives practical measurements, and Section 7 concludes. For reasons of space, we omit some of the proofs. They can be found in [8].

2 Basic Definitions and Notations

Let Σ be a finite set, the *alphabet*. The elements of Σ are *characters*. ε denotes the *empty string* and Σ^* denotes the set of strings over Σ. The *length* of a string x, denoted by $|x|$, is the number of characters in x. If $x = uvw$ for some (possibly empty) strings u, v, and w, then u is a *prefix* of x, v is a *subword* of x, and w is a *suffix* of x. A prefix or suffix of x is *proper* if it is different from x. A set X of strings is *prefix-closed* if $u \in X$ whenever $ua \in X$. x_i is the ith character of x, i.e., if $|x| = n$, then $x = x_1 \dots x_n$, where $x_i \in \Sigma$.

An *edit operation* is a pair $(a, b) \in (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. It is usually written as $a \to b$. An *alignment* A of two strings u and v is a sequence $(a_1 \to b_1, \dots, a_h \to b_h)$ of edit operations such that $u = a_1 \dots a_h$ and $v = b_1 \dots b_h$. A *weight function* δ assigns to each edit operation $a \to b$, $a \neq b$ a positive real weight $\delta(a \to b)$. The weight $\delta(a \to a)$ of an edit operation $a \to a$ is 0. If $\delta(a \to b) = 1$ for all edit operations $a \to b$, $a \neq b$, then δ is the *unit weight function*. The weight $\delta(A)$ of an alignment A is defined by $\delta(A) = \sum_{a \to b \in A} \delta(a \to b)$. The *weighted edit distance* of u and v is the minimum possible weight of an alignment of u and v.

3 Approximate String Searching

By a slight modification of the dynamic programming algorithm for computing the weighted edit distance, Sellers [16] obtained a simple method (SEL for short) to solve the approximate string searching problem. Sellers' method is usually described by giving the recurrence for an $(m+1) \times (n+1)$-table. Our approach is slightly different. We specify SEL by an initial distance column and a function that transforms one distance column into the next distance column, according to some character b. This schema will prove to be very convenient when we describe our new algorithms.

Definition 1. C denotes the set of functions $f : \{0, \ldots, m\} \to \mathbb{R}^+$ where $\mathbb{R}^+$ is the set of non-negative real numbers. The elements of C are *columns*. We define a function $nextdcol : C \times \Sigma \to C$ as follows. For all $f \in C$ and all $b \in \Sigma$, $nextdcol(f, b) = f_b$ where $f_b(0) = 0$ and

$$f_b(i+1) = \min \left\{ \begin{array}{l} f_b(i) + \delta(p_{i+1} \to \varepsilon), \\ f(i) + \delta(p_{i+1} \to b), \\ f(i+1) + \delta(\varepsilon \to b) \end{array} \right\}$$

Moreover, we define a function $dcol : \Sigma^* \to C$ by $dcol(vb) = nextdcol(dcol(v), b)$ and $dcol(\varepsilon) = f_\varepsilon$ where $f_\varepsilon(0) = 0$ and $f_\varepsilon(i+1) = f_\varepsilon(i) + \delta(p_{i+1} \to \varepsilon)$. $dcol(v)$ is the *distance column* of v. $f \in C$ is a distance column if $f = dcol(v)$ for some $v \in \Sigma^*$. □

Algorithm SEL [16] Compute $dcol(\varepsilon)$. For each $j, 1 \leq j \leq n$, compute $dcol(t_1 \ldots t_j) = nextdcol(dcol(t_1 \ldots t_{j-1}), t_j)$. If $dcol(t_1 \ldots t_j)(m) \leq k$, then output j. □

It is straightforward to show that $dcol(t_1 \ldots t_j)(i)$ is the minimal weighted edit distance of $p_1 \ldots p_i$ and a suffix of $t_1 \ldots t_j$. This implies the correctness of Algorithm SEL. For each $j, 0 \leq j \leq n$, the distance column $dcol(t_1 \ldots t_j)$ can be computed in $\mathcal{O}(m)$ steps. This gives an overall time efficiency of $\mathcal{O}(m \cdot n)$. Since in each step at most two columns have to be stored, SEL needs $\mathcal{O}(m)$ space. In the following, we show how to improve SEL. The idea is to compute the distance column of $t_1 \ldots t_j$ modulo some equivalence.

Definition 2. [18] An entry $f(i)$ of a distance column f is *essential* if $f(i) \leq k$. $lei(f) = \max\{i \mid 0 \leq i \leq m, f(i) \leq k\}$ is the *last essential index* of f. The distance columns f and f' are equivalent, denoted by $f \equiv f'$, if for all $i, 0 \leq i \leq m$, $f(i) = f'(i)$ whenever $f(i) \leq k$ or $f'(i) \leq k$. □

The equivalence notion for distance columns was basically introduced in [18]. As already stated by Ukkonen (see [18, Lemma 4]), the relation $\equiv$ is preserved by *nextdcol*. That is, for each character b and each distance column f and f' we have: $f \equiv f'$ implies $nextdcol(f, b) \equiv nextdcol(f', b)$. Note that $f(m) \leq k$ if and only if the last essential index of f is m.

Let $l = lei(f)$. The essential entries of $f_b = nextdcol(f, b)$ do not depend on the entries $f(l+1), f(l+2), \ldots, f(m)$ since these are larger than k. Hence it is not necessary to calculate f_b completely, as done by Sellers' algorithm. The calculation of f_b can be modified as follows. Compute $f_b(0), f_b(1), \ldots, f_b(l)$ according to Definition 1. If $l < m$, then compute

$$\begin{array}{lcl} f_b(l+1) & = & \min\{f_b(l) + \delta(p_{l+1} \to \varepsilon), f(l) + \delta(p_{l+1} \to b)\}, \\ f_b(l+2) & = & f_b(l+1) + \delta(p_{l+2} \to \varepsilon), \\ f_b(l+3) & = & f_b(l+2) + \delta(p_{l+3} \to \varepsilon), \\ & \vdots & \end{array}$$

until an entry $f_b(h)$ is reached such that either $h = m$ or $f_b(h) > k$ holds. Thus the computation of f_b is *cut off* at index h. The last essential index of f_b is the maximal $i, 0 \leq i \leq h$ such that $f_b(i) \leq k$. This modification leads to a cutoff variation of Sellers' method which was suggested by Ukkonen [17] in the context of the unit weight function. Chang and Lampe [3] showed that Ukkonen's cutoff trick leads to an expected running time of $\mathcal{O}(k \cdot n)$. Our empirical measurements suggest that this result holds for arbitrary weight functions, too. However, we have no proof for this. Note that the cutoff variation does not improve on the worst case efficiency of $\mathcal{O}(m \cdot n)$.

4 Essential Suffixes

In the previous section, we have seen that for solving the approximate string searching problem, it suffices to compute the essential entries of a distance column. In this section, we consider the strings which determine these entries.

Definition 3. Let v be a string. A suffix s of v is *essential* if $dcol(s) \equiv dcol(v)$. $ses(v)$ denotes the *shortest essential suffix* of v. □

Note that $ses(v)$ depends on δ, p, and k. Since δ, p, and k are arbitrary but fixed, we omit them in our notation. In the terminology of [18], $ses(t_1 \ldots t_j)$ is the "viable (k-approximate) prefix (of p) at j."

Example 1. Let δ be the unit weight function and $p = abbb$. Suppose $v = bbaba$. Then we have $dcol(bbaba) = (0,0,1,1,2)$, $dcol(baba) = (0,0,1,1,2)$, $dcol(aba) = (0,0,1,1,2)$, $dcol(ba) = (0,0,1,2,3)$, $dcol(a) = (0,0,1,2,3)$, and $dcol(\varepsilon) = (0,1,2,3,4)$. If $k = 0$, then $bbaba$, $baba$, aba, ba, and a are the essential suffixes of v. Hence $ses(v) = a$. If $k > 0$, then $bbaba$, $baba$, and aba are the essential suffixes of v. Hence $ses(v) = aba$. □

$ses(v)$ determines the essential entries of $dcol(v)$. That is, if $ses(v) = ses(v')$ holds, then $dcol(v) \equiv dcol(ses(v)) = dcol(ses(v')) \equiv dcol(v')$ (see [18, Theorem 1]). This property makes the shortest essential suffixes very interesting for the approximate string searching problem. All solutions to this problem can be enumerated according to the following idea:

> Determine the shortest essential suffixes of all prefixes of t. If the last entry in the distance column of a shortest essential suffix, say s, is at most k, then output the positions in t where s ends.

In fact, the algorithms developed in [18, 5] follow this idea. In particular, they determine the nodes in $stree(t)$ which correspond to a shortest essential suffix. From these nodes, the solutions to the approximate string searching problem can be read, provided $stree(t)$ has been annotated appropriately.

Before we describe how the idea above can be realized using DFAs, we give some more properties of shortest essential suffixes. Lemma 4 states that the set of all shortest essential suffixes can be represented by a trie. This is a new result. Lemma 6 states that shortest essential suffixes can be computed using length columns.

Lemma 4. *The set* $\{ses(v) \mid v \in \Sigma^*\}$ *is not empty, prefix-closed, and finite.* □

Definition 5. For all strings v we define $lcol(v) \in C$ as follows. For $i, 0 \leq i \leq m$, $lcol(v)(i)$ is the length of the shortest suffix of v, whose weighted edit distance to $p_1 \ldots p_i$ is $dcol(v)(i)$. $lcol(v)$ is the *length column* of v. $g \in C$ is a length column if $g = lcol(v)$ for some $v \in \Sigma^*$. □

Lemma 6. *Let* $l = lei(dcol(v))$. *Then* $lcol(v)(l) = |ses(v)|$. □

In analogy to *nextdcol*, we introduce a function for computing length columns.

Definition 7. For $f, g \in C$ and $b \in \Sigma$, the function $nextlcol : C \times C \times \Sigma \to C$ is defined as follows: Let $f_b = nextdcol(f, b)$. Then $nextlcol(f, g, b) = g_b$ where $g_b(0) = 0$ and

$$g_b(i+1) = \begin{cases} g_b(i) & \text{if } f_b(i+1) = f_b(i) + \delta(p_{i+1} \to \varepsilon) \\ g(i) + 1 & \text{else if } f_b(i+1) = f(i) + \delta(p_{i+1} \to b) \\ g(i+1) + 1 & \text{else if } f_b(i+1) = f(i+1) + \delta(\varepsilon \to b) \end{cases}$$

This recurrence is also given in [18]. It is easy to show that $lcol(\varepsilon)(i) = 0$ holds for all $i, 0 \leq i \leq m$. Moreover, for each $v \in \Sigma^*$ and each $b \in \Sigma$, $lcol(vb) = nextlcol(dcol(v), lcol(v), b)$. As can easily be verified, a length column can be computed in $\mathcal{O}(m)$ steps. Since length columns depend on the corresponding distance columns, both are computed simultaneously, as described in [18]. The notion of equivalence extends as follows: Two pairs (f, g) and (f', g') of distance and length columns are equivalent, denoted by $(f, g) \equiv (f', g')$, if for all $i, 0 \leq i \leq m$, $f(i) = f'(i)$ and $g(i) = g'(i)$ whenever $f(i) \leq k$ or $f'(i) \leq k$.

5 Algorithm SESA and its Variations

DFAs find wide application in string processing. For instance, they were used to solve the k-differences problem. Ukkonen [17] described an algorithm which preprocesses Σ, k, and p into a DFA whose states depict the possible distance columns. Each transition in Ukkonen's DFA represents the computation of a

distance column from a previous distance column. After precomputing the DFA, t can be processed in linear time. However, the preprocessing time can be $\mathcal{O}(3^m)$. Wu, Manber, and Myers [19] have shown how to reduce the preprocessing effort for Ukkonen's algorithm, by applying the "Four Russian's" paradigm. Their algorithm achieves a running time of $\mathcal{O}(k \cdot n/\log n)$ in the expected case. It is important to note that the algorithms in [17, 19] exploit geometric properties of distance columns, in order to efficiently store and retrieve these. Unfortunately, these geometric properties do not hold for arbitrary weight functions. Therefore it seems unlikely that the ideas of [17, 19] lead to efficient solutions for the approximate string searching problem. Instead of directly precomputing distance columns, our approach is to precompute the strings which determine the essential entries of distance columns. This can be done efficiently for arbitrary weight functions.

Definition 8. The *SES-automaton* for Σ, δ, k, and p is the deterministic finite automaton $(SES, \mathcal{F}, s^0, nextstate)$ where

1. $SES = \{ses(v) \mid v \in \Sigma^*\}$ is the set of *states*,
2. $\mathcal{F} = \{s \in SES \mid dcol(s)(m) \leq k\}$ is the set of *accepting states*,
3. $s^0 = \varepsilon$ is the *initial state*, and
4. $nextstate : SES \times \Sigma \to SES$ is the *transition function* defined as follows: for each $s \in SES$ and each $b \in \Sigma$, $nextstate(s, b)$ is the longest suffix of sb that occurs in SES. □

Example 2. Let δ be the unit weight function, $\Sigma = \{a, b\}$, $k = 1$, and $p = abba$. The corresponding *SES*-automaton is given by the following table. Accepting states are underlined.

	ε	a	ab	$\underline{aba}$	$\underline{abaa}$	$\underline{abb}$	$\underline{abba}$	$\underline{abbaa}$	$\underline{abbab}$	$\underline{abbb}$	b	bb	$\underline{bba}$
a	a	a	aba	$abaa$	a	$abba$	$abbaa$	a	aba	bba	a	bba	a
b	b	ab	abb	ab	ab	$abbb$	$abbab$	ab	abb	bb	bb	bb	ab

Note that for the same Σ, δ, k, and p, the algorithm of Ukkonen [17] computes a DFA with only 11 states. This is due to the fact that the shortest essential suffixes $abaa$, $abbaa$, and bba have the same distance column $(0, 0, 1, 2, 1)$. In fact, one can show that Ukkonen's DFA is never larger than the corresponding *SES*-automaton. □

Algorithm SESA Construct the *SES*-automaton $M = (SES, \mathcal{F}, s^0, nextstate)$ for Σ, δ, k, and p. For each $j, 0 \leq j \leq n-1$, compute $s^{j+1} = nextstate(s^j, t_{j+1})$. Output j if $s^j \in \mathcal{F}$. □

The *SES*-automaton for Σ, δ, k, and p accepts a prefix $t_1 \ldots t_j$ of t if and only if there is an approximate match ending at position j. This is shown in the following theorem.

Theorem 9. *Algorithm SESA correctly solves the approximate string searching problem.*

Proof. Let $0 \leq j \leq n$. By induction on j one shows that s^j is the longest suffix of $t_1 \dots t_j$ that occurs in *SES*. Let $s = ses(t_1 \dots t_j)$. Since $s \in SES$ and s is a suffix of $t_1 \dots t_j$, we conclude that s is a suffix of s^j. Hence we have $dcol(s)(i) \geq dcol(s^j)(i) \geq dcol(t_1 \dots t_j)(i)$ for all $i, 0 \leq i \leq m$. Using this inequality, one easily shows $dcol(s^j) \equiv dcol(t_1 \dots t_j)$. That is, s^j is an essential suffix of $t_1 \dots t_j$. This implies $s = s^j$ and we can conclude: $s^j \in \mathcal{F}$ iff $dcol(s^j)(m) \leq k$ iff $dcol(s)(m) \leq k$ iff $dcol(t_1 \dots t_j)(m) \leq k$ iff j is a solution to the approximate string searching problem. □

Note that Algorithm SESA can easily be modified such that with each solution j it additionally outputs $ses(t_1 \dots t_j)$, which is the shortest approximate match ending at position j in t. All the algorithm needs, is to remember the last $|s^j|$ characters of $t_1 \dots t_j$ whenever it is in state s^j.

We now consider the construction of the *SES*-automaton M. Lemma 4 suggests a construction which begins with state $s^0 = \varepsilon$ and obtains longer and longer states of *SES*. Let $s \in SES$ and assume that we have computed $dcol(s)$, $lcol(s)$, and a transition $failure(s)$ yielding the longest *proper* suffix of s that occurs in *SES*. $failure(s)$ is the *failure*-transition of the pattern matching machine of [1]. It is well-defined, whenever $s \neq s^0$. According to Lemma 6, $sb \in SES$ if and only if $|sb| = lcol(sb)(l)$ where $l = lei(dcol(sb))$. To check the latter condition, we evaluate the distance column $dcol(sb) = nextdcol(dcol(s), b)$ and the length column $lcol(sb) = nextlcol(dcol(s), lcol(s), b)$, and proceed according to the following cases:

(*i*) If $|sb| = lcol(sb)(l)$, then we construct a state sb and add a transition $nextstate(s, b) = sb$. By definition, $sb \in \mathcal{F}$ if and only if $dcol(sb)(m) \leq k$. $failure(sb)$ is obtained as follows. If $s = s^0$, then $failure(sb) = s^0$. Otherwise, $failure(sb) = nextstate(failure(s), b)$.

(*ii*) If $|sb| \neq lcol(sb)(l)$, then $nextstate(s, b)$ is the longest *proper* suffix of sb that occurs in *SES*. $nextstate(s, b)$ is obtained as follows. If $s = s^0$, then $nextstate(s, b) = s^0$. Otherwise, $nextstate(s, b) = nextstate(failure(s), b)$.

The computation of the transitions, as described above, requires a certain evaluation order. In particular, before $failure(sb)$ and $nextstate(s, b)$ are computed, the transitions $failure(s)$ and $nextstate(failure(s), b)$ have to be constructed. An evaluation strategy respecting this order was already described in [1]. It works in two phases: In the first phase, all states and all transitions of the form $nextstate(s, b) = sb$ are constructed depth first. This means that a state s' is constructed before a state s, whenever s' is a *proper* prefix of s. If $|sb| \neq lcol(sb)(l)$, then the transition $nextstate(s, b)$ is left undefined in this phase. In the second phase, the *failure*-transitions and all remaining transitions of M are computed in a breadth first traversal over the states. This means that all transitions (including *failure*-transitions) outgoing from a state s' are constructed before the transitions outgoing from a state s, whenever $|s'| < |s|$.

Theorem 10. *SESA runs in* $\mathcal{O}(|\mathrm{SES}| \cdot |\Sigma| \cdot m + n)$ *time and* $\mathcal{O}(|\mathrm{SES}| \cdot |\Sigma| + m^2)$ *space.*

Proof. M can be represented in $\mathcal{O}(|SES| \cdot |\Sigma|)$ space. During the first phase of the construction extra space for the distance and length columns is required. If s is the state for which the transitions are to be constructed, then it suffices to store the distance and the length columns for the prefixes of s. Since the length of s, as well as the size of the columns is $\mathcal{O}(m)$ in the worst case, $\mathcal{O}(m^2)$ space is required for storing the columns. The first phase of the construction takes $\mathcal{O}(|SES| \cdot |\Sigma| \cdot m)$ time, since for each of the $|SES| \cdot |\Sigma|$ transitions, the functions *nextdcol* and *nextlcol* are called. Consider the second phase. The breadth first traversal can be implemented using a queue of pointers to states. For details see [1]. The queue requires $\mathcal{O}(|SES|)$ space. Each transition which was left undefined in the first phase, can be constructed in constant time. Hence each state is processed in $\mathcal{O}(|\Sigma|)$ steps. After M is constructed, t is scanned in $\mathcal{O}(n)$ time, without using extra space. □

As noted in [18], the length of a shortest essential suffix is $\mathcal{O}(m)$. This gives a very rough upper bound for the size of SES. For a very general class of weight functions (which includes all integer weight functions), we can prove a useful property relating length and distance columns. This property can be exploited to derive a tighter upper bound.

Lemma 11. *Suppose* δ *is a weight function such that* $\delta(\varepsilon \to b) \geq 1$ *for all edit operations of the form* $\varepsilon \to b$. *Then we have* $lcol(v)(i) \leq i + dcol(v)(i)$ *for each* $v \in \Sigma^*$ *and each* $i, 0 \leq i \leq m$. □

Theorem 12. *Let* δ *be as in Lemma 11. Then* $|\mathrm{SES}| \leq 2 \cdot |\Sigma|^{m+k} - 1$.

Proof. Let $s \in SES$ and $l = lei(dcol(s))$. Then $|s| = lcol(s)(l) \leq l + dcol(s)(l)$, by Lemmas 6 and 11. Now $l \leq m$ and $dcol(s)(l) \leq k$. This implies $|s| \leq m + k$. □

The exponential preprocessing effort limits the applicability of Algorithm SESA. Therefore we have developed two variations of Algorithm SESA which reduce the number of states and transitions computed, while still retaining efficient search for approximate matches.

5.1 Variation 1: Depth Restriction

An important observation for Algorithm SESA is that in a typical situation most of the states in M are visited very rarely. In particular, the longer a state, the more unlikely it is to be visited. In this section, we show how to exploit this observation. The idea is to precompute only a part of M consisting of all states, whose length is $\leq r$ for some fixed integer $r \geq 1$. This limits the number of states to $2 \cdot |\Sigma|^r - 1$. The key of the method lies in taking care to reenter the restricted automaton as soon as possible.

Definition 13. Let $r \geq 1$ and $M = (SES, \mathcal{F}, s^0, nextstate)$. The *restricted SES-automaton* for Σ, δ, k, and p is the DFA $M_r = (SES_r, \mathcal{F}_r, u^0, nextstate_r)$ where

1. $SES_r = \{u \in SES \mid |u| \leq r\}$ is the set of states,
2. $\mathcal{F}_r = \mathcal{F} \cap SES_r$ is the set of accepting states,
3. $u^0 = \varepsilon$ is the initial state,
4. $nextstate_r : SES_r \times \Sigma \to SES_r$ is the transition function defined as follows. If $|u| < r$, then $nextstate_r(u, b)$ is the longest suffix of ub that occurs in SES_r. If $|u| = r$, then $nextstate_r(u, b)$ is undefined. □

The restricted *SES*-automaton processes most of the characters in t just like the *SES*-automaton does. However, if $u = ses(t_1 \dots t_j)$ and $|u| = r$, then there is no transition outgoing from state u. In this case, the next characters are processed by some dynamic programming steps starting with $dcol(u)$ and $lcol(u)$. This leads to a sequence of pairs $(f^{j'}, g^{j'})$ of distance and length columns. Once it can be inferred from these columns that $|ses(t_1 \dots t_{j'})| \leq r$, the restricted *SES*-automaton is reentered. In order to reenter the appropriate state, our method maintains for each $j, 0 \leq j \leq n$, the longest suffix u^j of $t_1 \dots t_j$ that occurs in SES_r.

Algorithm rSESA Preprocess the restricted *SES*-automaton for Σ, δ, k, and p. For each state $u \in SES_r$ with $|u| = r$, store $dcol(u)$, $lcol(u)$, and a transition $failure_r(u)$ yielding the longest *proper* suffix of u that occurs in SES_r. Compute a sequence $v^0, v^1, \dots, v^n$ of values $v^j \in SES_r \cup (SES_r \times C \times C)$ where $v^0 = u^0$ and v^{j+1} is determined as follows:

1. Suppose $v^j = u^j$ for some $u^j \in SES_r$. If $|u^j| < r$, then let $u^{j+1} = nextstate_r(u^j, t_{j+1})$ and $v^{j+1} = u^{j+1}$. Assume that $|u^j| = r$. Then let $u^{j+1} = nextstate_r(failure_r(u^j), t_{j+1})$, $f^{j+1} = nextdcol(dcol(u^j), t_{j+1})$, and $g^{j+1} = nextlcol(dcol(u^j), lcol(u^j), t_{j+1})$. If $g^{j+1}(lei(f^{j+1})) \leq r$, then let $v^{j+1} = u^{j+1}$. Otherwise, let $v^{j+1} = (u^{j+1}, f^{j+1}, g^{j+1})$.
2. Suppose $v^j = (u^j, f^j, g^j)$ for some $u^j \in SES_r$ and some $f^j, g^j \in C$. Let u^{j+1} be as in the previous case. Moreover, let $f^{j+1} = nextdcol(f^j, t_{j+1})$ and $g^{j+1} = nextlcol(f^j, g^j, t_{j+1})$. If $g^{j+1}(lei(f^{j+1})) \leq r$, then let $v^{j+1} = u^{j+1}$. Otherwise, let $v^{j+1} = (u^{j+1}, f^{j+1}, g^{j+1})$.

For $0 \leq j \leq n$ output j if either $v^j = u^j$ and $u^j \in \mathcal{F}_r$ or $v^j = (u^j, f^j, g^j)$ and $f^j(m) \leq k$. □

Theorem 14. *Let q be the number of indices j such that $|ses(t_1 \dots t_j)| \geq r$. Algorithm rSESA correctly solves the approximate string searching problem in $\mathcal{O}(|\Sigma|^{r+1} \cdot m + q \cdot m + n)$ time and $\mathcal{O}(|\Sigma|^{r+1} + |\Sigma|^r \cdot m)$ space.*

Proof. Let $0 \leq j \leq n$ and $s^j = ses(t_1 \dots t_j)$. It is straightforward to show that u^j is the longest suffix of $t_1 \dots t_j$ that occurs in SES_r. Moreover, if $v^j = u^j$, then $|s^j| \leq r$ which implies $u^j = s^j$. If $v^j = (u^j, f^j, g^j)$, then $|s^j| > r$ and $(f^j, g^j) \equiv (dcol(s^j), lcol(s^j))$. These properties imply the correctness of rSESA. M_r can be obtained in a similar way as M, except that the depth first construction of the states backtracks whenever a state of length r has been constructed. M_r has $2 \cdot |\Sigma|^r - 1$ states in the worst case. For each state of length $< r$, an array of size $|\Sigma|$ is stored. For each state of length r, two columns of size $\mathcal{O}(m)$ are stored. If $|ses(t_1 \dots t_j)| \geq r$, then *nextdcol* and *nextlcol* are called. Each call takes $\mathcal{O}(m)$ time. □

The choice of r is very important for the efficiency of rSESA. On the one hand, the preprocessing effort grows exponentially with r. One the other hand, q decreases with increasing r. Thus r is a parameter that allows to balance preprocessing versus search efficiency. Expressing q as a function of r, and obtaining analytical results to guide the best choice for r, remains a subject of future work.

5.2 Variation 2: Lazy Construction

The second variation, called lazySESA, applies the technique of lazy evaluation to reduce the number of states and transitions computed. This technique was already used by Aho in the implementation of *egrep* (see [2, Section 3.7]). The idea is to interleave the construction of M with the scanning of the input string. That is, a transition is constructed immediately before it is to be traversed for the first time. lazySESA works as follows. Let $s = s^j$, $b = t_{j+1}$ and assume that the transition $nextstate(s, b)$ has not been constructed yet. It can be constructed as described in case (i) and (ii) above. If $s \neq s^0$, then in both cases a transition $nextstate(failure(s), b)$ is to be traversed. This may itself have not been constructed yet. In other words, in order to perform a transition $s^{j+1} = nextstate(s^j, t_{j+1})$, there may be one or more transitions to be constructed. Let q_s be the number of states and q_t be the number of transitions constructed by lazySESA. Obviously, $q_s \leq q_t$. Recall that the construction of a transition $nextstate(s, b)$ leads to a *failure*-transition. Traversing a *failure*-transition consumes at least one character of t. Hence there can be at most n *failure*-transitions. As a consequence, we have $q_t \leq n$.

Theorem 15. *Algorithm lazySESA correctly solves the approximate string searching problem in $\mathcal{O}(q_s \cdot |\Sigma| + q_t \cdot m + n)$ time and $\mathcal{O}(q_s \cdot (|\Sigma| + m))$ space.*

Proof. The correctness of lazySESA is clear. For each constructed state s, a Σ-indexed array is used to store the possible transitions for s. The array requires $\mathcal{O}(|\Sigma|)$ space. Allocation takes $\mathcal{O}(|\Sigma|)$ time. Note that it is required to store $dcol(s)$ and $lcol(s)$, whenever there is a character b such that $nextstate(s, b)$ has not been constructed yet. Hence, for each state $\mathcal{O}(|\Sigma| + m)$ space is required. To construct a transition $nextstate(s, b)$, we need to evaluate $dcol(sb)$ and $lcol(sb)$. This takes $\mathcal{O}(m)$ time. □

Since lazySESA requires $\mathcal{O}(n \cdot (|\Sigma| + m))$ time in the worst case, it does not improve on Algorithm SEL, in general. However, if k is not too large, then q_s and q_t can be considerably smaller than n, thus leading to an improved running time for lazySESA. Note that the automaton constructed by lazySESA can be reused for other input strings.

6 Practical Measurements

Algorithms SEL, lazySESA, and rSESA were coded in C. In all implementations we applied the cutoff trick to reduce dynamic programming steps. Moreover, we implemented a space optimized variation of Ukkonen's Algorithm A [18]. This variation, called UKKA$'$, does *not* assume that $stree(t)$ is precomputed. Instead it processes t online and constructs, in a lazy way, a trie representing those parts of $stree(t)$ which are actually traversed by Ukkonen's Algorithm A (for details see [8]). UKKA$'$ is easy to implement and it can handle the large input strings we used in our measurements. This is not true for the algorithms which require a precomputed suffix tree: To implement $stree(t)$, roughly 4 integers per character of t are required (see [10]). The additional annotation of $stree(t)$, as used in Ukkonen's [18] and in Cobbs' algorithms [5], takes at least one extra integer and one boolean per character of t. On a typical current architecture an integer requires 4 bytes and a boolean 1 byte. This means that the precomputed suffix tree structure alone requires $21 \cdot n$ bytes. This is too much when $n = 10^6$, as in our measurements.

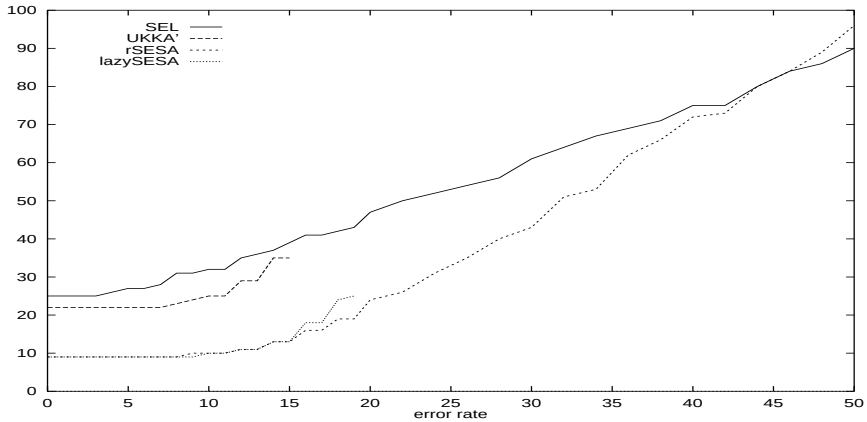
Our runtime measurements were performed on a SPARC 10/41 with 32 MB memory. We used biological sequences. The input string for the first test series was the complete DNA sequence of chromosome XI of the yeast S. cerevisiae [6]. Hence $n = 666,448$ and $\Sigma = \{A, C, G, T\}$. We used the symmetric transversion/transition weight function δ_1, represented by the following weight matrix:

δ_1	ε	A	C	G	T
ε					
A	3	0			
C	3	2	0		
G	3	1	2	0	
T	3	2	1	2	0

Bases A and G are called purine, and bases C and T are called pyrimidine. The transversion/transition weight function reflects the biological fact that a purine/purine and a pyrimidine/pyrimidine replacement is much more likely to occur than a purine/pyrimidine replacement. Moreover, it takes into account that a deletion or an insertion of a base occurs more seldom. We performed measurements for different error rates $\varrho = (100 \cdot k)/m$ between 0 and 50. Recall that k is a weighted threshold and not the *number* of possible errors. Since the average non-zero value of δ_1 is 2, an error rate $\varrho = 50$ allows $k/2 = (\varrho \cdot m)/(2 \cdot 100) = 0.25 \cdot m$ errors in an approximate match (on the average). Figure 1 shows the total running times when searching 24 cytochrome P450 sequences of length

between 7 and 51. These were selected from the TRRD database. For rSESA we chose $r = 8$. UKKA′ ran into space problems for $\varrho > 15$, and lazySESA for $\varrho > 18$. Therefore the corresponding measurements are missing. From Figure 1 we observe that for $\varrho \leq 45$, rSESA is faster than SEL. For $\varrho \leq 20$ it is more than twice as fast.

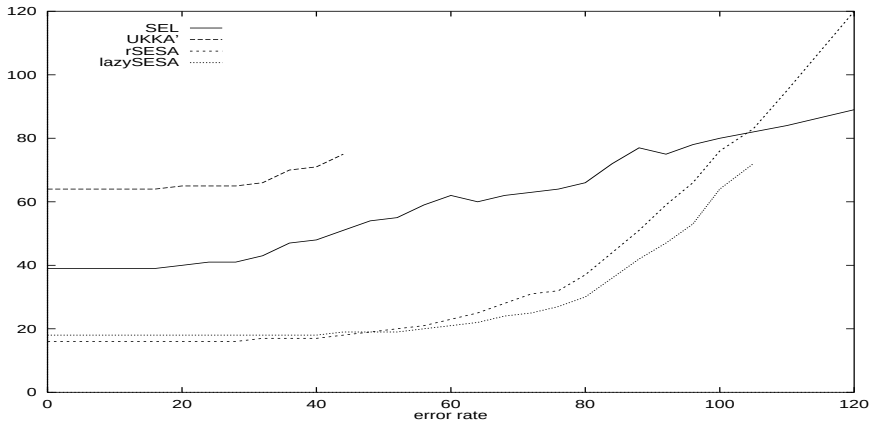
Fig. 1. Running times in seconds for varying error rates and DNA sequences



The input string for the second test series was a section of 1 million residues from the PIR database. The underlying alphabet is of size 21, including a wild card symbol X matching any residue. We transformed the PAM250 scoring matrix from [15] into a weight function δ_2. In particular, if $\sigma(a \to b)$ gives the PAM250-score of the replacement operation $a \to b$, $a \neq b$, then we defined $\delta_2(a \to b) = -\sigma(a \to b) + 8$. A replacement of the symbol X by any residue was weighted 0. Indels were weighted 12. We performed measurements for different error rates $\varrho = (100 \cdot k)/m$ between 0 and 120. Note that $\varrho \geq 100$ makes sense, since the weighted threshold k can be greater than m. Since the average non-zero value of δ_2 is 9.8, an error rate $\varrho = 120$ allows $k/9.8 = (120 \cdot m)/(9.8 \cdot 100) = 0.12 \cdot m$ errors in an approximate match (on the average). Figure 2 shows the total running times when searching 28 cytochrome P450 sequences of length between 10 and 30. These were selected from the PIR database. For rSESA we chose $r = 4$. UKKA′ ran into space problems for $\varrho > 48$, and lazySESA for $\varrho > 105$. Therefore the corresponding measurements are missing. From Figure 2 we observe that for $\varrho \leq 105$, Algorithms rSESA and lazySESA are faster than SEL. For $\varrho \leq 80$ they are twice as fast. UKKA′ is considerably slower than SEL. This may be due to the linked list implementation we used for the set of edges outgoing from a node in a trie.

Note that in the first test series, Algorithms rSESA and lazySESA perform relatively better, in comparison to SEL. This may be explained by the fact that DNA sequences, due to the smaller alphabet, contain more repeated subwords than protein sequences.

Fig. 2. Running times in seconds for varying error rates and protein sequences



7 Conclusion

As already observed in measurements of algorithms solving the k-differences problem (see [7, 8]), no algorithm is the best in all cases. A reasonable solution is a method which selects the best algorithms on the bases of Σ, δ and the error rate ϱ: If DNA sequences are searched and δ_1 is the cost function, then rSESA is a good choice for $\varrho \leq 45$, and SEL for $\varrho > 45$. If protein sequences are searched and δ_2 is the cost function, then rSESA or lazySESA is a good choice for $\varrho \leq 105$, and SEL for $\varrho > 105$.

Note that the programs we measured only compute the positions, where an approximate match ends. In applications, however, it is often required to report approximate matches as well. While our implementation can easily be modified to do so without much extra effort, Sellers' method additionally has to compute the lengths columns. Thus, in such applications, our algorithms are likely to perform even better in comparison to Sellers' method.

Some questions concerning our methods remain open for research. These include theoretical analysis of the expected running time as well as comparisons against the methods in [18, 5]. Our algorithms may be improved further by applying techniques from [5], or the "Four Russian's" paradigm [19].

References

1. A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, **18**:333–340, 1975.

2. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1985.

3. W.I. Chang and J. Lampe. Theoretical and Empirical Comparisons of Approximate String Matching Algorithms. In *Proc. of* CPM 92*,* LNCS 644, pages 175–184, 1992.

4. W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, **12**(4/5):327–344, 1994.

5. A.L. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proc. of* CPM 95*,* LNCS 937, pages 41–54, 1995.

6. B. Dujon et al. The Complete DNA Sequence of Chromosome XI of Saccharomyces cerevisiae. *Nature*, **396**:371–378, 1994.

7. P. Jokinen, J. Tarhio, and E. Ukkonen. A Comparison of Approximate String Matching Algorithms. Technical Report A-1991-7, Department of Computer Science, University of Helsinki, 1991.

8. S. Kurtz. *Fundamental Algorithms for a Declarative Pattern Matching System.* Dissertation, Technische Fakultät, Universität Bielefeld, available as Report 95-03, 1995.

9. G.M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, **10**:157–169, 1989.

10. U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5):935–948, 1993.

11. E.W. Myers. A Sublinear Algorithm for Approximate Keyword Searching. *Algorithmica*, **12**(4/5):345–374, 1994.

12. E.W. Myers. Algorithmic Advances for Searching Biosequence Databases. In S. Suhai, editor, *Computational Methods in Genome Research*, pages 121–135. Plenum Press, 1994.

13. E.W. Myers. Approximate Matching of Network Expressions with Spacers. *J. Comp. Biol.*, **3**(1):33–51, 1996.

14. E.W. Myers and W. Miller. Approximate Matching of Regular Expressions. *Bulletin of Mathematical Biology*, **51**(1):5–37, 1989.

15. W.R. Pearson. Rapid and Sensitive Sequence Comparison with FASTP and FASTA. In Doolittle, R., editor, *Methods in Enzymology*, volume **183**, pages 63–98. Academic Press, San Diego, CA, 1990.

16. P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**:359–373, 1980.

17. E. Ukkonen. Finding Approximate Patterns in Strings. *Journal of Algorithms*, **6**:132–137, 1985.

18. E. Ukkonen. Approximate String-Matching over Suffix Trees. In *Proc. of* CPM 93*,* LNCS 684, pages 229–242, 1993.

19. S. Wu, U. Manber, and E.W. Myers. A Subquadratic Algorithm for Approximate Limited Expression Matching. *Algorithmica*, **15**, 1996.