Media Engineering and Technology Faculty
German University in Cairo

Approximate pattern matching under generalised edit distance and extensions to suffix array library

Bachelor Thesis

Author:	Abdallah El Guindy
Supervisors:	Prof. Rolf Backofen
	Dr. Michael Beckstette
	Dr. Sebastian Will
Reviewer:	Name of the Reviewer
Submission Date:	12 July, 2009

Media Engineering and Technology Faculty
German University in Cairo

Approximate pattern matching under generalised edit distance and extensions to suffix array library

Bachelor Thesis

Author: Abdallah El Guindy
Supervisors: Prof. Rolf Backofen
Dr. Michael Beckstette
Dr. Sebastian Will
Reviewers: Name of the Reviewer
Submission Date: 12 July, 2009

This is to certify that:

(i) the thesis comprimises only my original work toward the Bachelor Degree

(ii) due acknowlegement has been made in the text to all other material used

---

Abdallah El Guindy
12 July, 2009

Abstract

The approximate pattern matching problem is the problem of finding all occurences of a certain pattern in a usually much longer text allowing for a fixed error threshold in the matching. The problem has been studied extensively and many very good solutions were found. However, significant instances of the problem, namely those allowing for generalised edit-distance error functions, remain without satisfactory algorithms. This thesis suggests a promising solution. The new method provided relies on the suffix array data structure to preprocess the text linearly and allow later for fast queries. The new algorithm has the very desirable feature of having a fairly simple description and implementation while performing well in practice. Furthermore, the new algorithm handles wildcards quite well while retaining the same time and space worst-case complexities. The algorithms are compared on genuine genetic data from Zebrafish genome and the results are presented. In addition to presenting the new algorithm, the thesis discusses several extensions to an existing affix array library.

Contents

1 Introduction **1**
- 1.1 Importance of the problem . . . 1
- 1.2 Problem description . . . 1
- 1.3 Previous results . . . 2
  - 1.3.1 Unit-cost-error solutions . . . 2
  - 1.3.2 General-cost-error solutions . . . 2
- 1.4 Goals of the project . . . 3
- 1.5 Overview of the thesis . . . 3

2 Background **5**
- 2.1 Definitions and conventions . . . 5
  - 2.1.1 Basic string definitions . . . 5
  - 2.1.2 Notations and conventions . . . 5
  - 2.1.3 Distance function . . . 6
  - 2.1.4 Models of computation . . . 6
  - 2.1.5 Time and space complexities . . . 6
- 2.2 Problem statement . . . 7
  - 2.2.1 Assumptions . . . 7
- 2.3 Data structures and algorithms . . . 8
  - 2.3.1 Suffix trees . . . 8
  - 2.3.2 Suffix arrays . . . 9
  - 2.3.3 Enhanced suffix arrays . . . 9
  - 2.3.4 Range minimum query . . . 9
  - 2.3.5 Tree-array equivalence . . . 10
  - 2.3.6 Bounded-error global aligment . . . 12
- 2.4 Naive solution of the approximate pattern matching . . . 13

3 Extensions to affix array library **14**
- 3.1 Additional tables . . . 14
  - 3.1.1 Child tables . . . 14
  - 3.1.2 Calculating child intervals . . . 14
- 3.2 Traversals . . . 14
  - 3.2.1 Suffix links . . . 15

4 New algorithm **16**
4.1 Basic idea 16
4.2 Speedup 16
4.3 Further optimizations 17
4.4 Space complexity 17
4.5 Time complexity 20

5 Implementation details and results **22**
5.1 Test environment 22
5.2 Implementations 22
5.3 Practical considerations 23
5.4 Test data 23
5.5 Performance 23
5.6 Test results 24

6 Conclusion **26**
6.1 Future work 26

References **28**

Chapter 1

Introduction

1.1 Importance of the problem

The applications of the approximate pattern matching extend to reach numerous fields. Most notably, these applications include looking up a genome data base for some DNA sample allowing for mutation, searching signals after being transmitted in noisy channels and searching in text allowing for typing errors [16]. Additional applications also include handwriting recognition, virus detection, pattern reccognition and optical character recognition [16]. In bioinformatics for example, the typical sizes of for example genome databases are in the order of gigabytes, efficiency of search is a must.
The importance of generalizing the error function is however less obvious. The problem itself is the very natural generalization to the unit-cost edit distance. The importance is perhaps best demonstrated by our everyday use of the keyboard. It is quite clear that a typist is much more likely to mistake an "s" for an "a" than mistaking an "o" for "a". This follows from the positions of the keys on the keyboard. Therefore, it makes sense to introduce weights on the different types of errors that can occur in the pattern. Generalizing the error function also has its uses in bioinformatics. For example, it is natural that gene insertions or deletions can be more common that transformations. Furthermore, some mutations may be more common than others. Recent research shows that some DNA mutation operations indeed are more common than others [18] (thus, deserving a different cost).

1.2 Problem description

Pattern matching is the problem of finding a pattern in a usually much longer text. This is also sometimes referred to as the exact pattern matching problem. The problem has its clear applications in computation and bioinformatics. Searching for subtexts in texts is studied since a while ago and elegant and efficient solutions for almost every variation of the problem have been formulated. Perhaps the most notable results include Knuth-Morris-Pratt algorithm [9], Aho-Corasick algorithm [2] and a theoretically interesting unification of many techniques presented by Gusfield [7].
A related problem, is the approximate pattern matching problem. Informally speaking, it is the problem of finding occurences of a small pattern in a much longer text allowing

for some small bounded error in the matching process. The problem can be formalised in several ways, mainly depending on how "error" is interpreted and what assumptions are made about the text and the pattern. Again, this problem has been well studied and several very interesting results, both theoretically and practically have been found. Yet, a general enough result that is good in practice is not yet available up to the writer's best knowledge. In this thesis, a very flexible solution is presented. The algorithm performs very well in practice and adapts well to the problem's structure and generality/specificity.

1.3 Previous results

This section explores the existing solutions to the problem. Throughout the discussion below, the length of the text and the pattern will be denoted by n and m respectively. In addition, the alphabet size will be called α and the machine word size (in bits) w. Finally the error is given the symbol k. Those solutions generally fall into two main categories:

1.3.1 Unit-cost-error solutions

Those are solutions that assume that all different types of errors incur the same cost (usually taken without loss of generality to be unity). Among the first results, was a result due to Ukkonen proven later to have an expected running time of $O(kn)$ [3]. Landau and Vishkin gave later an algorithm that runs in worst case in $O(kn)$ and a parallelized version that runs in $O(k + logm)$ [11]. Later, two new approaches to solving the problem emerged. The first was using bitwise parallelism to gain speedup factors proportional to the computer word size. The second was employing filtering techniques to heuristically speed up the matching process by quickly excluding positions in which a match is not possible. Myer presented an algorithm that uses bitwise parallelism that runs in $O(mn/w)$ [15] and together with filtering techniques, algorithm vmatch was implemented and remains among the fastest (if the not the fastest) practically. Our algorithm will be compared later against vmatch.

1.3.2 General-cost-error solutions

Very recently, Gonzalo Navarroa and Edgar Chvez started tackling the problem under generalized error functions. They noticed that the problem can be formulated as an adjacency problem in a metric space (more on that later). They produced two results that give unprecedented time bounds. The results however are theoretical in nature and are very complicated (even the authors did not produce any implementation and did not compare against previous results). The first runs in expected time $O(mlog^2n + m^2 + t_{occ})$ and requires $O(n)$-space and $O(nlog^2n)$-time for preprocessing [4]. The algorithm was the first to achieve a sublinear (in n) time bound for the problem in the average case. The second runs in expected time $O(m^{1+\epsilon} + t_{occ})$ as long as the error is $o(m/log_\alpha m)$ and $m > ((1 + \epsilon)/\epsilon)log_\alpha n$. It requires $O(m^{1+\sqrt{2}+\epsilon} nlogn)$-time and $O(m^{1+\sqrt{2}+\epsilon} n)$-space preprocessing resources [17]. This algorithm achieves the more crucial result, namely independence from n altogether.
The algorithms above are however not very practical. Only few practical algorithms solve the general problem. Besides the naive $O(mn)$ dynamic programming, which is generally

very good with small texts, an $O(\alpha mq)$, where $q \leq n$, algorithm due to Cobbs [5] seems to be the best existing result so far.

1.4 Goals of the project

The first goal of the project is to provide a practical solution to the approximate pattern matching problem under generalised edit distance. As the typical sizes of the texts is very big, the algorithm is allowed at most linear preprocessing time and space. Furthermore, the preprocessing space should have very low constants so that the preprocessing structures can fit in memory. Throughout, the space consumption in terms of bytes is kept track of. The algorithm should also run queries in very short time and light memory foot print. The algorithm is always compared against Myer's algorithm (described above) despite the fact that it solves a more general class of problems. One of the main aims is that the algorithm is competitive with the best performing algorithms. The algorithm should also be flexible to adapt to problem instances and scale up/down with the problem instance's complexity.
In terms of code, the algorithm should be simple to describe and implement. To quote Dijkstra: "simplicity is prerequisite for reliability." Simpler algorithms are easier to optimize, maintain and parallelize.
On a technical level, the algorithm should allow the patterns to have character classes, especially wildcards without causing the performance to deteriorate significantly (as long as the number of wildcards remains small). It should in addition allow for a more general class of scoring (error) functions rather than the very specific, heavily studied, unit-cost edit distance function. Finally, no assumptions or restrictions should be placed on the pattern lengths or the amount of error (other than the reasonable restrictions). That is, the algorithm correctness and efficiency should be unconditional with respect to the text length, pattern length or amount of error.
A second(ary) goal of the project is to extend and enhance the existing affix array library by implementing additional structures and routines needed to port most of suffix tree algorithms to suffix arrays. Extensions to the suffix array library are an essential component for almost any use of the suffix arrays (other than exact matching). With a huge legacy of algorithms using suffix trees, augmenting the suffix arrays to fully replace suffix arrays means automatically allowing all those algorithms to run with better performance using the simpler suffix arrays. Such applications include solutions to classical problems such as finding maximal palindromes, super maximal repeats, longest common extensions, ...etc.

1.5 Overview of the thesis

Chapter 2 introduces the necessary definitions, notations and data structures used throughout the thesis. Most importantly, it describes the crucial result of suffix tree-array equivalence and presents the necessary data structures to esablish this complete equivalence. It also presents two previous results, namely the naive dynamic programming and Cobbs' algorithm. Chapter 3 presents the affix array library extensions needed for the development of the algorithm. In chapter 4, a full development of the new algorithm (along

with necessary structures) is presented. Finally, time and space analysis of the algorithm are given. Chapter 5 discusses the implementation details and the results of testing the algorithm. A comparison with vmatch is provided in terms of preprocessing and running space and times. Chapter 6 concludes the thesis and discusses possible future research to be done in the field.

Chapter 2

Background

In this chapter, fundamental definitions and conventions are presented for later use throughout. In addition, the assumptions about the nature of the problem are explicitly specified. Finally, some preliminary results regarding the data structures and algorithms used later are presented for completeness.

2.1 Definitions and conventions

This section, the basic definitions and conventions are introduced. In addition, the distance function is defined and the assumed models of computation are discussed.

2.1.1 Basic string definitions

We follow the usual convention of defining the *alphabet* Σ as a finite non-empty set of symbols. The set Σ^k is the set of k-tuples from Σ. The set Σ^0 is the set containing single element ε. The set $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ is the set of all tuples over the alphabet Σ. A *string* S is an element of Σ^*. The length of S, denote by $|S|$ is the number k such that $S \in \Sigma^k$, i.e. the number of characters in the string.

2.1.2 Notations and conventions

A string S is simply denoted by $s_1 s_2 .. s_k$ instead of $(s_1, s_2, .., s_k)$. $S[i..j]$ is the substring of S starting at position i and ending at position j, i.e. $s_i s_{i+1} .. s_j$ if $i \leq j$ and ε otherwise. The i_{th} suffix of a string S, namely the suffix starting at position i is denoted by S_i. Notice that $S_0 = S$. In addition, throughout the paper, we will refer to two important strings P (the pattern) and T (the text). We will reserve the two variables n and m to stand for $|T|$ and $|P|$ respectively. We denote $|\Sigma|$ by α. Finally, we refer throughout the thesis to a special blank symbol $-$. It is assumed that $- \notin \Sigma$. Finally, we use the letters k and κ. The former is used when the error function is the unit-cost edit-distance while the latter is used when the function is generalised-edit-distance.

2.1.3 Distance function

The class of functions under considerations is a slight generalization of the edit-distance function. The allowed operations are insertion, deletion and substitution. Instead of each assuming a unit cost, each operation is given a weight. Formally the (character) distance function is defined as a function $d : \Sigma \cup \{-\} \times \Sigma \cup \{-\} \rightarrow \mathbb{R}$ that obeys the properties of a metric. The following definition restates the definition of a metric.

Definition 1 *A function $f : X \times X \rightarrow \mathbb{R}$ is said to be a* metric *if $\forall x, y, z \in X$:*

1. $d(x, y) \geq 0$ (non-negative)
2. $d(x, y) = d(y, x)$ (symmetric)
3. $d(x, y) = 0 \leftrightarrow x = 0$
4. $d(x, y) + d(y, z) \leq d(x, z)$ (triangle inequality)

In addition, with $x \in \Sigma \cup \{-\}$ and Y a nonempty subset of Σ, we define $d(x, Y) = min\{d(x, y)|y \in Y\}$. The definition is of interest when dealing later with character groups. A wildcard is just defined to be the character group Σ.
Using the character distance, a (generalised) edit-distance between two strings S_1 and S_2 can be defined. We define it as the sum of costs of the distance between individual characters in the optimal alignment of the two strings. That is, it is the minimum over all alignments of the strings (padded by blank symbols). The definition is identical to that of the edit-distance, except that the distance between characters assumes any value (not just unity). We abuse the notation and use d for both character and string distances.

2.1.4 Models of computation

Throughout the discussion, the standard unit-cost serial computation model is assumed. All arithmetic and bitwise operators are assumed to take constant time as long as the numbers fit in the machine registers. The registers are assumed to have size $O(logn)$, where n is the maximum size of a problem (for example length of text).

2.1.5 Time and space complexities

The time and space complexities of all algorithms are given in the big/small O notation as customary. The operations assumed to run in constant time were discussed in section 2.1.4. Otherwise, almost nothing is assumed to be a constant. Most notably, the alphabet size dependence is made explicit in the time/space analysis as in some applications the alphabet size is considerably large (for example, amino acids in protein chains).

Remark The preprocessing space is not given in asymptotic notation, but rather as number of bytes per input text symbol. The reason for this is that the text size is usually very big. Saying that the space is linear is not always practical if the constant is big. Therefore, it is absolutely necessary for the preprocessing space constant to be small (usually less than 20).

2.2 Problem statement

With all prerequisite background concepts defined, we are now in position to define the treated problem rigoursly.

Definition 2 *For strings P and T, a position i,* $1 \leq i \leq n$ *is said to be an* approximate match *of P in T with a threshold* κ *under some string distance function* μ *if* $\mu(P, T[i..i + L]) \leq \kappa$ *for some* $L \leq n - i$*.*

Defintion 2 defines the intuitive notion of an approximate match of a small string in a longer one. It basically defines matches to be those starting positions of substrings of T that do not differ that much from P under our assumed metric. The next defintion defines our problem to be finding all approximate matches of the pattern in the text.

Definition 3 *The* approximate pattern matching problem *is defined as given* (T, P, κ, σ) *where* $T, P \in \Sigma^*$*,* $\kappa \in \mathbb{R}$ *and d is a (character) generalized edit-distance function, find all* appromixate matches *i of P in T with threshold* κ *under the string distance function induced from* σ*.*

The problem can be equivalently defined to report ending positions of matches. Both definitions are reducible to one another by just reversing both strings. Note that the definition above does not account for the presence of wild cards or character groups. This can be accounted for by allowing the pattern to be a tuple of non-empty sets of symbols instead of just symbols. The symbols in the pattern are just singleton sets and the character groups are just non-singleton sets.

2.2.1 Assumptions

What is meant by an efficient solution varies significantly depending on what assumptions we make about the problem. In this section, all non-obvious assumptions made about the problem are explicitly listed. Some assumptions are necessary for the validity of some later claimed facts and some are necessary if presented solutions are to be considered efficient or effective. Some other assumptions exist to simplify later discussion.

Remark The following assumptions are made about the problem tackled:

1. The text and the pattern have size that fits in one (or a constant number of) computer word(s), i.e. the size of computer word is $O(log|T|)$.
2. The long text is available offline for preprocessing. Furthermore, we are allowed an initial linear number of accesses to the text during our proprocessing phase. The allowed preprocessing space is also linear.
3. The d function (distance between symbols) is well-defined, obeys the metric properties and takes constant time to compute for any pair of characters.
4. $|P| \leq |T|$, in fact usually $|P| \ll |T|$.

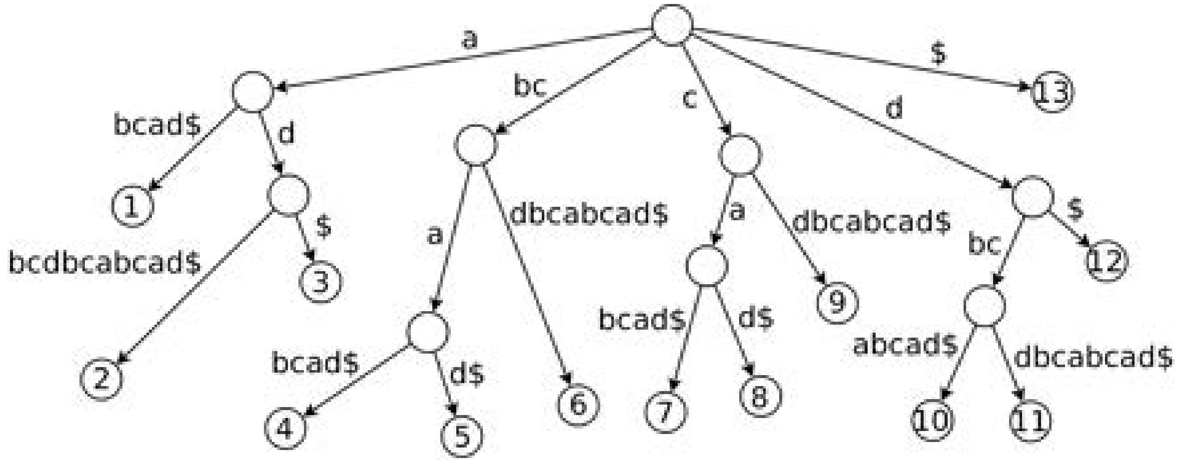


Figure 2.1: Suffix tree for string adbcdbcabcad$

5. Notice that the behaviour of the distance function is unaffected by scaling up/down. Thus, for convenience, we assume that the distance function is normalized such that the minimum cost of insertion or deletion is unity.

Note that not all assumptions may hold in practice. Assumptions 1 and 4 are quite realistic. 2 and 3 define what variation of the problem we are trying to solve, and in case either does not hold, another solution than the ones presented should be seeked. Assumption 5 causes no loss of generality as the scoring function can be scaled together with the error threshold to obtain an equivalent problem.

2.3 Data structures and algorithms

2.3.1 Suffix trees

The suffix tree data structure is the most central data structure in the field of string algorithms. Introduced in 1973 under the name "position tree" [19], the data structure was used in an amazing variety of applications. Some of the most elementary algorithms on suffix trees solve problems on strings in time bounds that were conjectured to be unachievable.
A suffix tree for a string S is a directed tree with edges labeled by strings. The path from the root to each node spells out a substring of S. The tree has $|S|$ leaves each representing a (non-empty) suffix of S. No two edges out of the same node carry strings starting with the same characters. In addition, it is required that each internal node of the tree is branching (i.e. has an out degree > 1), i.e. the tree is compact. Figure 2.1 shows an example of suffix trees.
For a string of length n, the suffix tree has at most $2n - 1$ nodes. This follows from the facts that the tree has n leaves and each internal node is branching, thus the number of internal nodes can be at most $n - 1$. Suffix trees can be constructed in linear time and space [14] and have an amazing variety of applications such as finding supermaximal repeats, maximal palindromes and many others. The suffix tree can also be extended to contain the suffixes of more than one string.

2.3.2 Suffix arrays

A suffix array for a string S is simply a list of sorted suffixes of the string . It was introduced as an asymptotically less efficient alternative to suffix trees [13]. Later however, it was shown that suffix arrays can be constructed in linear space and time [10]. Table 2.1 shows an example of suffix arrays (for the same string of the suffix tree). Similar to suffix trees, suffix arrays can be constructed for a group of strings.

i	Suff[i]	Lcp[i]	T_i
0	7	0	abcad$
1	0	1	adbcdbcabcad$
2	10	2	ad$
3	5	0	bcabcad$
4	8	3	bcad$
5	2	2	bcdbcabcad$
6	6	0	cabcad$
7	9	2	cad$
8	3	1	cdbcabcad$
9	4	0	dbcabcad$
10	1	3	dbcdbcabcad$
11	11	1	d$
12	12	0	$

Table 2.1: Suffix array for string adbcdbcabcad$ (same string as of figure 2.1)

2.3.3 Enhanced suffix arrays

The suffix array on its own can not solve the same set of problems solved using suffix trees. The suffix array can be augmented with many additional tables, most importantly the *longest-common-prefix table* (lcp for short). The entry lcp[i] is defined as the length of the longest common prefix between suffix at entry i in the table and that at entry i-1 (note that lcp[0] is just defined to be zero). Figure 2.1 shows also the lcp-table. Once the suffix array is constructed, the lcp table can be constructed in linear time and space [8].

2.3.4 Range minimum query

For purposes discussed later, we are interested in answering questions about the LCP table. The main example of such question is knowing the index of the minimum LCP value in a contiguous range in the table. The problem is generally called Range Minimum Query (RMQ) as is defined as follows:

Definition 4 *For an array A of N numbers, the* RMQ(i, j) *for* $1 \leq i \leq j \leq N$ *to be the minimum k, such that* $A[k] \leq A[s]$ *for all* $i \leq s \leq j$.

In our case, the array A is the LCP value table. The problem has been well studied and many solutions were found with optimal linear preprocessing and constant query time.

The solution of choice [6] uses $2n + o(n)$ bits preprocessing space and linear preprocessing time. After the preprocessing, any subsequent RMQ query can be answered in $O(1)$ time. The RMQ data structure is central to our later discussion of the suffix tree suffix array equivalence.

2.3.5 Tree-array equivalence

Very recently, the enhanced suffix array data structure was found to be totally equivalent to the suffix tree of the corresponding string. This means that for every algorithm using the suffix tree, a corresponding algorithm using the suffix array with the same time and space complexities exists [1]. This allows for completely replacing suffix trees with suffix arrays gaining the following advantages:

1. Suffix arrays use less space than suffix trees.
2. Suffix arrays have better locality of reference behaviour.
3. Suffix arrays are simpler to construct.
4. Suffix arrays are easier to store/load to/from disk.

The first step towards establishing the equivalence is to define lcp-intervals.

Definition 5 *An interval $[i, j]$ in the suffix array is called a* λ-interval *and denoted by $[i, j]$-λ if the following conditions hold:*

1. $\text{lcp}[i] < \lambda$
2. $\text{lcp}[j + 1] < \lambda$ if $j + 1 < n$
3. $\text{lcp}[k] \geq \lambda$ for all $i < k \leq j$
4. $\text{lcp}[k] = \lambda$ for at least one k, $i < k \leq j$

Remark Singletons are eliminated from being λ-intervals as a side-effect of condition 4.

What is the significance of such LCP-intervals? They some how carry the same information as suffix tree nodes. Figure 2.2 illustrates this equivalence. An LCP-interval $[i, j]$-λ basically represents a group of suffixes sharing a common prefix of length λ. In addition, the common prefix is maximal, i.e. this group of suffixes shares a prefix no longer than λ. This is very similar to an internal node in the suffix tree. If the length of the label of the node is λ, it means that suffixes corresponding to the leaves in the subtree rooted at this node share a longest common prefix of length λ, since every internal node is branching. The following notion is formalised in the following theorem.

Theorem 2.3.1 *A bijection exists between the LCP-intervals and the internal nodes of the suffix tree of the same string.*

Proof *Sketch:* For an lcp interval $[i, j]$-λ, consider the string $\beta = T[\text{suff}[i] .. \text{suff}[i]+\lambda-1]$. The string is a substring of T and this exists in the suffix tree either at a node or in the middle of an edge. Since this is the longest common prefix, $T[\text{suff}[i] + \lambda - 1]$ must be different from $T[\text{suff}[j] + \lambda - 1]$ (remember that the suffixes are listed in the suffix array in lexicographic order). This means that after the substring β the tree has to branch. Therefore, the substring β must end at a node in the suffix tree. The reverse argument can be applied proving that the function is indeed a bijection. A detailed discussion can be found in [1]. ∎

Corollary 2.3.2 *LCP-intervals and singleton intervals (rows in the suffix array) have one to one correspondence with all nodes of the suffix tree.*

Proof Notice that singleton intervals correspond to leaves of the suffix tree. ∎

One to one correspondence between LCP-intervals (and singleton intervals) and suffix tree nodes is not sufficient for full replacement of suffix trees with suffix arrays. It still remains to be shown that the tree structure exists among LCP-intervals. Indeed, this correspondence is an isomorphism with respect to the relations of siblings, parents and children. That is, if an interval I_1 and I_2 correspond to nodes v_1 and v_2 respectively, then if v_1 is the parent of v_2, then interval I_1 is the smallest λ-interval that includes I_2. It is also the same for sibling and child relations. It suffices to show the child relation amongst intervals and the other two relations will follow. The following defintions and theorem establish the child relations.

Definition 6 *For an interval $[i, j]$-λ, an index k, such $i \le k \le j$ and lcp$[k] = \lambda$ is said to be a* λ-index *of the interval.*

Definition 7 *For an interval $[i, j]$-λ, define the* children intervals *to be $[i, k_1 - 1]$-λ_1, $[k_1, k_2 - 1]$-λ_2, .., $[k_N, j]$-λ_{N+1} where k_1, k_2, .., k_N are the λ-indices of the interval and each of the λ_i's is the minimum LCP value in the range defined by the adjacent brackets.*

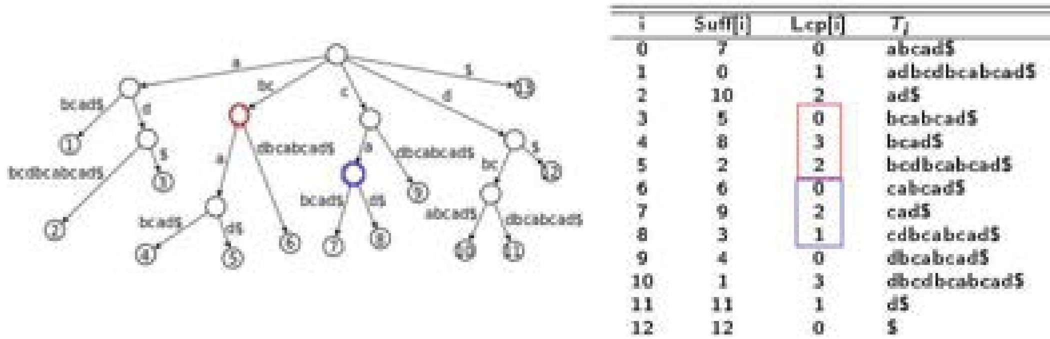
Theorem 2.3.3 *The bijection described in theorem 2.3.1 between the LCP and singleton intervals and the suffix tree nodes is an isomorphism with respect to the relation "child".*

Proof See [1].

To summarise, the full structure of the suffix tree is indeed implicit in the enhanced suffix array data structure.

Corollary 2.3.4 *There exists a bijection between λ-intervals $\cup$ singleton intervals and nodes of the suffix tree for the same string. In addition, this bijection is an isomorphism for the relations parent, child and sibling.*

Although this asserts that both structures contain the same data, it does not directly imply that all algorithms that operate on suffix trees can use suffix arrays efficiently instead. This however follows once the routines of traversing the tree top-down and bottom-up are written for lcp-intervals of the suffix array. This was also done in [1]. Optimal linear time bottom up traversal was given earlier by Kasai et al. [8]. In addition,



i	Suff[i]	Lcp[i]	T_j
0	7	0	abcad$
1	0	1	adbcdbcabcad$
2	10	2	ad$
3	5	0	bcabcad$
4	8	3	bcad$
5	2	2	bcdbcabcad$
6	6	0	cabcad$
7	9	2	cad$
8	3	1	cdbcabcad$
9	4	0	dbcabcad$
10	1	3	dbcdbcabcad$
11	11	1	d$
12	12	0	$

Figure 2.2: Equivalence of suffix tree and suffix array for string adbcdbcabcad$. The blue and red interval correspond to the blue and red nodes respectively.

two methods of getting children lcp-intervals was also provided in [1] allowing for optimal linear time top-down traversal. The first method was fully implemented and explained using an additional structure called child tables (which encode the parent/child relations among intervals). The second was hinted in the same paper and relies on RMQ data structure. Although the former runs faster, the latter is the method of choice as it uses much less space, takes shorter preprocessing time and is much cleaner to code. However, both methods are implemented and compared for the sake of completeness. The following theorem summarises those results.

Theorem 2.3.5 *For every algorithm using suffix trees, including traversing the tree and using auxiliary structures (such as suffix links), there exists an equivalent algorithm that has the same time and space bounds and uses suffix arrays but not suffix trees.*

2.3.6 Bounded-error global aligment

A final prerequisite to presenting our algorithm is the global alignment problem. It is the problem of calculating the edit-distance (as described before) between two strings. The problem has been solved for general reasonable scoring functions (those that are edit-like and metric) by the Needleman-Wunsch algorithm. The algorithm is a straight forward dynamic programming using the following recurrence:

$$D[0,0] = 0$$

$$D[i,j] = min \begin{cases} D[i-1,j] + d(T[i-1], -), \\ D[i,j-1] + d(P[j-1], -), \\ D[i-1,j-1] + d(T[i-1], P[j-1]) \end{cases}$$

Remark We assume that whenever i or j becomes negative the value is infinite and that the addition of an infinite with a finite is infinite.

The problem can be modified to match the strings allowing for an upper-bound on the distance between the strings. If the strings have a distance not more than the bound, the distance is calculated, otherwise the algorithm reports that the distance is bigger than

the bound. Although the above recurrence can solve this modified version as well, it is only necessary to compute only $2k+1$ band around the main diagonal. This is done, by assuming that $D[i,j] = \infty$ if $|i-j| > k$ (the error bound). This improves the time complexity to $O(kn)$ instead of $O(n^2)$.

2.4 Naive solution of the approximate pattern matching

The naive solution of the problem follows from a very simple modification of the simple Needleman-Wunsch algorithm presented in the previous section. The main difference is that spaces at the beginning of the pattern are free. We account for this by setting $D[i,0] = 0$ for all $0 \leq i \leq n$. All solutions (ending positions) are those i such that $D[i,m] \leq \kappa$.
The dynamic programming solution is very important as many of the more sophisticated algorithms rely on properties of the DP table to compute the result faster. In fact, in the special case when $\sigma(x,y)$ is either 0 or 1, it is clear that no two adjacent entries in the table differ by more than 1. Thus, computation of the table can be accelerated as complete batches of the table are computed at once using bitwise operators. This is known as the Four Russians Trick.

Chapter 3

Extensions to affix array library

As a prerequisite to the implemented algorithms, additional structures were implemented and added to the existing affix array library.

3.1 Additional tables

3.1.1 Child tables

The child tables hold the parent/child relations among lcp-intervals. They consist of three main tables: up, down and next-l. Up and down tables are used to locate the first child interval. Afterwards, following the next-l table entries the other child intervals are found one by one. All three tables can also be saved in only one table to save space [1]. The naive implementation requires 12 bytes per input symbol, while the space efficient one requires 4 bytes per input symbol. The implementation followed the pseudo-code implementations described in [1].

3.1.2 Calculating child intervals

Implementing the top-down traversal of the lcp-interval tree simply reduces to implementing the *GetChildren* function, which is given an lcp-interval, it returns a list of all children lcp-intervals. The function can be implemented in one of two ways. The first uses child tables (with the exception of the children of the root interval) as described in [1]. The second is much easier conceptually and uses the RMQ data structure to quickly find l-indices following the brief description at the end of [1] and the description of [6] of the RMQ data structure. Both implementations run in $O(\alpha)$, where α is the number of children lcp-intervals.

3.2 Traversals

The bottom up traversal was implemented as described in [8] and the top-down traversal was implemented following the description in [1]. The top-down traversal utilizes the GetChildren function implemented in either of the two ways described above, this leads to two different implementations.

3.2.1 Suffix links

Suffix links are an additional structure augmented to the suffix tree. They are links attached to each node that link a node to the node representing the longest suffix of the substring represented by the node. The suffix links can also be constructed for the lcp-intervals of the suffix array. The suffix links can be constructed in linear time and space using two traversals of the lcp-interval tree [12]. The generic traversals are directly used for the construction of the suffix links as described in [12].

Chapter 4

New algorithm

This section presents the algorithms implemented. This section only provides a theoretical discussion, for the implementation details and running times, see the next section.

4.1 Basic idea

As explained in the previous section, the bounded-error global alignment problem of two strings can be solved in $O(\kappa L)$ space and time, where L is the length of the shorter string. This inspires a very simple algorithm for the approximate pattern matching problem. Try for every position to solve the bounded-error global alignment problem. Algorithm 1 demonstrates the idea. The correctness of the algorithm is very straight-forward. Each iteration of the algorithm takes $O(\kappa m)$ time, thus, the total time complexity is $O(\kappa mn)$.

Algorithm 1 Naive algorithm

```
for i = 0 to n do
  solve the bounded-error global alignment problem for strings P and T_i.
end for
```

4.2 Speedup

The basic idea of the algorithm does not any more complicated than described in the previous section. Great speedups however are obtained by using suffix arrays to prevent redundant computations. The algorithm exploits the fact that when the text is repetitive or long enough, the different suffixes share common prefixes. The main idea is to compute entire sections of the dynamic programming table one time for shared prefixes of different T_i's. Recalling that the suffix array is the array of the suffixes of a string sorted in lexicographic order, it is straight forward to see that it seems wise to consider the suffixes in the order of their appearance in the suffix array. With the suffixes now rearranged, we can compute parts of the dynamic programming table for whole lcp-intervals. Algorithm 2 shows the improved version.

Algorithm 2 LCP-interval algorithm

```
function fill_table (start, end, lcp, already_calculated)
if start = end then
  ROW = length of suffix starting at suf[start]
else
  ROW = lcp.
end if

Fill dynamic programming table for P and T_{suf[start]} from row already_calculated + 1
until ROW.

if start = end then
  Report a match starting at suf[start] if DP[i, m] ≤ κ for some i, m−κ ≤ i ≤ m+κ.
else
  for each [i-j] λ-interval child to [start-end] do
    fill_table (i, j, λ, lcp)
  end for
end if
end function
fill_table(0, n, 0, 0)
```

4.3 Further optimizations

The optimization described above, by itself, is a good improvement. A lot of redundant work is saved. Full traversal of the lcp-interval tree is still required however, this leads to doing work proportional to at least the length of the text (remember that the lcp-interval tree has N nodes, where $|T| \leq N < 2|T|$). We can do several simple improvements that lead to very good speedups in practice.
First, remembering that our scoring function is normalized so that the minimum cost of insertion is 1, it is easy to see that we need to only inspect the first $m + \kappa$ letters of each suffix. We need not go beyond that, since we are only allowed κ insertions at most. This improvement can be expressed as follows in the pseudocode:
One final optimization is obtained by noticing that once a row of the DP table contains only entries greater κ, no more rows need to be computed as the values of the cells in subsequent rows will always be greater than κ (since $d(a, b) \geq 0$). Algorithm 4 illustrates the final version of the algorithm.

4.4 Space complexity

The algorithm consumes linear preprocessing space. First the suffix array and LCP tables have to be constructed. Both together consume exactly $5n$ bytes [1], plus additional space for the exceptions table (which is very small in practice). In addition, either RMQ or child tables have to be constructed. The former consumes $n/4 + o(n)$ bytes while the latter uses $4n$ bytes and can be compressed in n bytes [1]. This gives a total of not more $6n$ for either case.

Algorithm 3 LCP-interval algorithm (modification 1)

function fill_table (*start*, *end*, *lcp*, *already_calculated*)

if $start = end$ **then**
  ROW = length of suffix starting at $suf[start]$
else
  $ROW = lcp$.
end if

Fill dynamic programming table for P and $T_{suf[start]}$ from row $already_calculated + 1$ until ROW.

if $start = end$ or $lcp \geq m + \kappa$ **then**
  for $j = start$ to end **do**
    Report a match starting at $suf[j]$ if $DP[i, m] \leq \kappa$ for some i, $m - \kappa \leq i \leq m + \kappa$.
  end for
else
  for each [*i*-*j*] λ-interval child to [*start*-*end*] **do**
    fill_table (i, j, λ, lcp)
  end for
end if

end function

Algorithm 4 LCP-interval algorithm (modification 2)

function fill_table (*start*, *end*, *lcp*, *already_calculated*)

if $start = end$ **then**
  ROW = length of suffix starting at $suf[start]$
else
  $ROW = lcp$.
end if

$LAST_ROW$ = Fill dynamic programming table for P and $T_{suf[start]}$ from row $already_calculated + 1$ until ROW. If the minimum of a row exceeds κ return the index of this row and do not continue, otherwise return ROW.

if $start = end$ or $lcp \geq m + \kappa$ **then**
  for $j = start$ to end **do**
    Report a match starting at $suf[j]$ if $DP[i, m] \leq \kappa$ for some i, $m - \kappa \leq i \leq LAST_ROW$.
  end for
else
  for each [*i*-*j*] λ-interval child to [*start*-*end*] **do**
    fill_table (i, j, λ, lcp)
  end for
end if

end function

The space complexity of search is rather straightforward to compute. It is stated in the following lemma.

Lemma 4.4.1 *The algorithm's space complexity is $O(\kappa m + \kappa^2)$.*

Proof The algorithm consumes space for two purposes, namely, maintaining recursion stack and for the DP matrix. The recursion depth is bounded by $O(m + \kappa)$. The DP solves the global alignment for strings of length $m + \kappa$, thus needing space $O(\kappa(m + \kappa)$. Therefore, a total of $O(\kappa m + \kappa^2)$ space is used by the algorithm. ∎

4.5 Time complexity

The preprocessing time is clearly linear because ll structures discussed previously can be constructed in linear time [1] [6].
The time complexity of the search algorithm is by far less obvious. In fact, no satisfactory time bounded was found, only loose bounds that are thought to be not a very good representative of the real performance of the algorithm. We present several time bounds that might give an indication about the absolute worst case of the algorithm. Note that for any given instance the minimum of the given time bounds is an upper bound on the algorithm.

Definition 8 *In the interval tree, call a node v* terminal *if the algorithm does not proceed to the children of v, i.e. process function returns false on input v.*

Lemma 4.5.1 *The algorithm runs in $O(v + \kappa mv + \kappa^2 v + t_{occ})$ where v is the number of terminal nodes.*

Proof Since every internal node is branching, and the algorithm either visits all children or none, then the total number of nodes visited by the algorithm is at most $2v - 1$. The traversal requires $O(v)$ time and assuming that all paths to the v terminal nodes are disjoint (which is quite impossible), then v different global aligment problems are solved each taking $O(\kappa(m + \kappa))$ time. Reporting the matches just involves iteration over them which takes $O(t_{occ})$. This gives a total complexity of $O(\kappa(m+\kappa)v+v) = O(v+\kappa mv+\kappa^2 v)$. ∎

Corollary 4.5.2 *The running time of the algorithm is $O(\kappa mn + \kappa^2 n + t_{occ})$ regardless of the problem instance and the number of wildcards or character groups.*

Proof It suffices to notice that $v \leq$ number of nodes in interval tree $< 2n$. ∎

The time bound given by corollary 4.5.2 is very loose. It is of no interest other than to show that the algorithm in theory does not perform much worse that Cobbs' algorithm. In fact, the κ factor is still bad enough given the fact that the problem is solvable naively in $O(mn)$. As we will see later however, the bound given does not reflect at all the true performance of the algorithm. The next lemma gives the most useful time bound.

Lemma 4.5.3 *The algorithm runs in $O((\kappa + 1)(\alpha - 1)^{\kappa+\omega+1} m^{\kappa+\omega+1} + (\kappa + 1)m + (\kappa + 1)^2 + t_{occ})$.*

Proof The number of wild cards can be thought of as some allowed mismatches. With this in mind, let's consider the problem with no wild cards and with error $\kappa + \omega$. The time complexity can be modelled by the recurrence: $T(d, \kappa) = T(d, \kappa - 1) + T(d - 1, \kappa) + (\alpha - 1)T(d - 1, \kappa - 1)$ where d denotes the depth in the tree and κ is the remaining error. The first term accounts for insertion, the second term accounts for a match and the third for a mismatch. Solving the recurrence yields the above expression (except for t_{occ} which comes from reporting matching positions). ∎

Lemma 4.5.3 shows that the algorithm can be used to solve exact matching (by having $\kappa = 0$) in optimal time $O(\alpha m + t_{occ})$ [1]. In addition, for small κ the algorithm performs very well. For example, for $\kappa = 1$, the complexity is $O(\alpha^2 m^2 + t_{occ})$, almost as good as in Cobbs' algorithm [5].
One final time bound is given. This one is very useful when both m and κ are small.

Lemma 4.5.4 *The algorithm runs in $O(\kappa\alpha^{m+\kappa} + t_{occ})$.*

Proof In the very worst case, The interval tree has all substrings of length $m + \kappa$. In this case, it is a complete α-ry tree with height $m + \kappa$ with each edge holding a single character from Σ. In the first level, a total of $O(\kappa\alpha)$ time is needed, in the second $O(\kappa\alpha^2)$, ...etc. Thus in total the complexity is $O(\kappa\Sigma_{i=1}^{m+\kappa}\alpha^i + t_{occ}) = O(\kappa\alpha^{m+k}(\alpha)/(\alpha - 1) + t_{occ}) = O(\kappa\alpha^{m+\kappa} + t_{occ})$ since $\alpha/(\alpha - 1) \leq 2$.

Chapter 5

Implementation details and results

This section provides an overview of actual implementations done, their analysis and how they performed when tested.

5.1 Test environment

All implementations described below were implemented in C and compiled using GCC version 4.1.2 with optimization level 2. The testing was performed on 64-bit cluster running Red Hat linux. The machine has 64 GB RAM memory and network-mapped drives.

5.2 Implementations

Firstly, the naive dynamic programming was implemented. The implementation was used as reference implementation for checking the correctness of the other algorithm. Furthermore, optimizing the naive implementation gave insight into several ways to optimize more sophisticated algorithms.
For the implementation of our algorithm, a suffix array data structure was needed. An affix array library written by Fernando Meyer was augmented with additional structures and used throughout our implementations. The library provided implementations of constructing the suffix array and the longest-common-prefix tables. First, an RMQ library based on [6] was integrated with the suffix array library. An implementation of child tables based on [1] was written. The previous implementations were used to write generic top-down and bottom-up traversals, again following the methods given in [1]. Finally, using the traversals, construction of suffix links was implemented as described in [12].
With all the needed structures in place, two separate versions of the algorithm were implemented. The first was implemented using child tables to simulate top down traversal. The second much neater version was implemented using the generic top down traversal to abstract all the details related to the traversal. Although the algorithm can be parallelized on a single pattern, the code was written to search for more than one pattern in parallel since it is frequently the case that multiple patterns are searched at once. Parallelizing across patterns (instead of parallelizing the algorithm itself) was noticed to be practically much better, incurring lower overheads.

5.3 Practical considerations

Throughout all implementations, large tables were loaded using the function memory map. This was done to overcome the delays arising from the network traffic overhead. This is does not cause the results to lose realism as the exact same performance can be obtained by loading the data on memory-mapped drives.
The running time of the applications was measured using time command and the results presented are the CPU times. Again, this is not the realistic time one would expect to get when running the application, however, this is the most realistic measure that excludes other overheads such as other processes running on the same machine.
The parallelization was implemented using the OpenMP library and its built-in support in the GCC compiler. This allowed parallelization of loops using minimal code and without the needed to consider any parallelization issues, such as spawning and maintaining threads.

5.4 Test data

The new algorithm was tested and compared with vmatch on three big data sets. The first two are the Danio rerio (zebrafish) genome. Each set was about 0.5 GB. 20 randomly patterns were chosen from probe sequences of agilent zebrafish. The patterns are each 60 characters long. The third data set is the Rfam RNA database consisting of about 150 MB. 16 randomly chosen patterns were produced by a RNA family descriptor generator. The patterns varied in length (7-37 characters).

5.5 Performance

As a first step, the index data structures (suffix arrays and auxiliary tables) were constructed for each data set. The preprocessing space used was competitive with vmatch implementation even when both the child tables and the RMQ tables were constructed at the same time. Table 5.1 shows the sizes of each constructed structure. Besides using much less space, the time needed for the preprocessing is slightly (sometimes significantly) better than vmatch's. Table 5.2 shows the time of constructing each structure.

Test	**Text**	**V. ESA**	**F. ESA**	**Child tables**	**RMQ tables**
Test 1 (ZF 1)	594.2 MB	12.4 GB	3.5 GB	6.8 GB	491.6 MB
Test 2 (ZF 2)	648.1 MB	13.5 GB	3.9 GB	7.4 GB	536.1 MB
Test 3 (Rfam)	197.5 MB	3.9 GB	1.2 GB	2.0 GB	139.6 MB

Table 5.1: Space needed for preprocessing. V. ESA stands for vmatch Enhanced Suffix Array, which refers to the implementation used in vmatch. F. ESA stands for Fernando's Enhanced Suffix Array, the implementation used by our algorithm. Child tables and RMQ tables are not used together but are rather alternative ways to achieve the same task (namely, top down traversal of the lcp-interval tree.

Test	V. ESA	F. ESA	Child tables	RMQ tables
Test 1 (ZF 1)	820.9s	708.7s	87.3s	23.2s
Test 2 (ZF 2)	881.1s	807.4s	95.2s	25.2s
Test 3 (Rfam)	231.7s	146.1s	41.9s	10.2s

Table 5.2: Time needed for preprocessing. V. ESA stands for vmatch's Enhanced Suffix Array, which refers to the implementation used in vmatch. F. ESA stands for Fernando's Enhanced Suffix Array, the implementation used by our algorithm. Child tables and RMQ tables are not used together but are rather alternative ways to achieve the same task (namely, top down traversal of the lcp-interval tree.

As shown, our preprocessing is by far superior in terms of space and time to that of vmatch. The space usage of the child tables can be even cut to a third by applying the trick mentioned in the end of [1]. Even though the RMQ tables where by far the least space consuming, the amount of space used was disappointing as it was much worse than the theoretical $2n + o(n)$ bits, the constant in the small o was quite big. If further space reduction is required, the RMQ tables can be replaced altogether by segment trees at a cost of logarithmic factor. Note that using segment trees, space requirement can be reduced by a factor of h at the cost of increasing the query time by the same factor.

5.6 Test results

Our algorithm was compared in a series of tests with vmatch. In each of the three data sets, the error value was varied in the range that is considered of practical interest, usually up to an error percentage of approximately 16%. Tables 5.3, 5.4 and 5.5 show the results of the tests. Two versions of our algorithm are compared. The former is the optimized version relying on the child tables for the traversal. The latter is implemented using the generic traversals implemented earlier, traversing the lcp-interval tree using RMQ queries.
The reader should be reminded that vmatch solves a restricted version of the problem solved by our algorithm, and both are compared at solving this version. With this in mind, it seems that our algorithm shows competitive performance with vmatch and even outperforms it drastically in some occasions (for example in test 3).

Error	0	1	2	3	4	5	6	7	8	9
T(s)	0.1	0.1	0.8	7.5	53.5	267.0	913.8	2193.6	4500.2	6604.3
T2(s)	0.1	0.2	1.6	16.2	114.6	568.3	1898.0	4458.1	7796.0	x
VT(s)	x	1.6	2.4	3.2	5.3	5.4	65.3	749.4	2223.6	5617.2

Table 5.3: Running times of our algorithm and vmatch's on the first data set (part one of the zebrafish genome). T stands for the optimized version of the algorithm, relying on child tables, T2 stands for the cleaner, yet slower, version of the algorithm that uses RMQ and generic traversal functions and V stands for vmatch. Entries having an x are either invalid, such as the first entry in the row of vmatch (vmatch does not accept query with 0 error) or are just regarded as uninteresting (while taking a lot of time to compute) since the pattern is clear from the rest of the table.

Error	0	1	2	3	4	5	6	7	8	9
T(s)	0.1	0.1	0.8	7.1	56.2	267.8	947.2	2368.0	4613.0	7106.5
T2(s)	0.1	0.2	1.5	15.9	115.4	579.9	2039.8	4781.3	8329.0	x
VT(s)	x	1.9	2.8	3.8	6.3	6.3	70.2	834.3	2699.9	6709.4

Table 5.4: Running times of our algorithm and vmatch's on the second data set (part two of the zebrafish genome). T stands for the optimized version of the algorithm, relying on child tables, T2 stands for the cleaner, yet slower, version of the algorithm that uses RMQ and generic traversal functions and V stands for vmatch. Entries having an x are either invalid, such as the first entry in the row of vmatch (vmatch does not accept query with 0 error) or are just regarded as uninteresting (while taking a lot of time to compute) since the pattern is clear from the rest of the table.

Error	0	1	2	3	4	5
T(s)	1.2	7.1	15.3	33.0	65.7	137.4
T2(s)	16.1	36.1	62.8	102.9	174.7	320.9
VT(s)	x	4.8	254.5	1046.5	2150.5	3226.2

Table 5.5: Running times of our algorithm and vmatch's on the third data set (Rfam database). T stands for the optimized version of the algorithm, relying on child tables, T2 stands for the cleaner, yet slower, version of the algorithm that uses RMQ and generic traversal functions and V stands for vmatch. Entries having an x are either invalid, such as the first entry in the row of vmatch (vmatch does not accept query with 0 error) or are just regarded as uninteresting (while taking a lot of time to compute) since the pattern is clear from the rest of the table.

Chapter 6

Conclusion

To summarise, the problem of approximate pattern matching problem is the problem of finding occurences of patterns in much longer texts. Restriced versions of the problem were solved satisfactorily, however, when slightly generalized, the problem seems to lack good enough practical solutions. In this thesis, an attempt was made to design an efficient algorithm that consumes acceptable preprocessing resources and provides efficient running time. The solution relies on the enhanced suffix array data structure to preprocess the long text and answer subsequent queries in fast time and little space.
First, several extensions were made to a very good suffix library, most notably, implementing additional tables and routines needed for the emulation of suffix tree algorithms using enhanced suffix arrays with no loss of performance. This leads to the immediate expansion of the class of problems solvable efficiently by the library. In addition, the new extensions lead immediately to an algorithm that solves the approximate pattern matching problem efficiently in practice. The algorithm is very simple and uses very little space. In addition, the algorithm handles character groups and wildcards very easily. Tests show that the algorithm competes with the best solutions to even more specific problems.

6.1 Future work

As for the suffix array library, full integration of the enhancements of the library is still needed. The extensions are implemented as a separate library that uses the suffix array library. Integrating the two libraries can lead to slightly better performance and much better usability. In addtion, several improvements for the suffix array library are suggested. Mainly, the limitation on the allowed text size should be removed leading to a much better running time of our algorithm. Furthermore, some other tables (for example, inverse suffix tables) should be implemented as some algorithms rely on their existence in the library code.
As for the algorithm, a tight time bound that reflects the real performance of the algorithm is still needed. The time bounds found were either linear in the text size or exponential in the error or the pattern length. This does not represent the real performance of the algorithm as shown by the tests. The algorithm should also be compared with Cobb's algorithm, which can be implemented to use suffix arrays (after the extension of the library) instead of suffix trees. Finally, it is conjectured that the suffix links

tables can be used to improve the performance of the algorithm by saving computations repeated in different branches of the lcp-interval tree. Further investigations are needed to verify or dismiss this conjecture.

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[3] William I. Chang and Jordan Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 175–184, London, UK, 1992. Springer-Verlag.

[4] Edgar Chvez, Gonzalo Navarro, and N+r Time. A metric index for approximate string matching. In *In LATIN*, pages 181–195, 2002.

[5] A. L. Cobbs. Fast approximate matching using suffix trees. 937:41–??, 1995.

[6] Johannes Fischer and Volker Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470, Hangzhou, China, April 7-9, 2007, 2007. Springer-Verlag.

[7] D. Gusfield. Simple uniform preprocessing for linear-time string matching. Technical report, CSE-96-5, UC Davis, Dept. of Computer Science, 1996.

[8] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. pages 181–192. Springer-Verlag, 2001.

[9] Donald E. Knuth, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[10] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Journal of Discrete Algorithms*, pages 200–210. Springer, 2003.

[11] G M Landau and U Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 220–230, New York, NY, USA, 1986. ACM.

[12] Moritz G. Maaß. Computing suffix links for suffix trees and arrays. *Inf. Process. Lett.*, 101(6):250–254, 2007.

[13] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[14] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[15] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46:1–13, 1999.

[16] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[17] Gonzalo Navarro and Edgar Chávez. A metric index for approximate string matching. *Theor. Comput. Sci.*, 352(1):266–279, 2006.

[18] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature Biotechnology*, 2008.

[19] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.