All-against-all Approximate Substring Matching

Marina Barsky

Department of Computer Science

University of Victoria

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

Master of Science

©Marina Barsky, 2006

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

All-against-all Approximate Substring Matching

by

Marina Barsky

M.Sc., University of Victoria, 2006

Supervisory Committee:

Dr.Alex Thomo, Supervisor
(Department of Computer Science)

Dr.Chris Upton, Co-supervisor
(Department of Biochemistry and Microbiology)

Dr.Ulrike Stege, Department Member
(Department of Computer Science)

Dr.John S. Taylor, External Examiner
(Department of Biology)

Supervisory Comitee:

Dr.Alex Thomo, Supervisor
(Department of Computer Science)

Dr.Chris Upton, Co-supervisor
(Department of Biochemistry and Microbiology)

Dr.Ulrike Stege, Department Member
(Department of Computer Science)

Dr.John S. Taylor, External Examiner
(Department of Biology)

Abstract

Finding local regions of high similarity in a set of strings is of great importance in biological sequence analysis. This problem is far from being efficiently solved.

In this thesis we study the best known solutions to this problem. We present a new and efficient algorithm to solve the "threshold all vs. all" variant of the problem, which involves searching two strings (with length N and M respectively) for all maximal approximate substring matches of length at least S, with up to K differences. The algorithm is based on a novel graph model and solves the problem in time $O(NMK^2)$.

We also explore the possibility of extending our approach to the local alignment problem for multiple strings. Our developed program is a practical solution that detects similar regions in a set of strings in a feasible time, for cases of practical importance.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Figures vi

Acknowledgement viii

1 Introduction 1

1.1 Motivation 2

1.2 Local Alignment 4

1.3 The Alternative Approach to the Problem of Local Similarity 5

1.4 Previous Work 6

1.5 Our Contribution 7

1.6 Structure of the Thesis 7

2 A New Algorithm for All-against-all Approximate Substring Matching 9

2.1 A Graph Model 9

2.2 Solving "All Paths Below the Threshold" 16

2.2.1 Constructing the Matching Matrix 16

2.2.2 A Single-Step Path Expansion 17

2.2.3 Interdependence of Paths in the Graph 23
2.3 Experimental Evaluation 27
2.3.1 Setup 29
2.3.2 Input Sequences 29
2.3.3 Comparative Performance 31
3 A Practical Application: All-against-all for Multiple Strings 36
3.1 Motivation 36
3.2 A Practical Approach 36
3.2.1 Program Flow 37
3.2.2 Program Evaluation 42
3.2.3 Final Remarks about the Program 44
4 Conclusions and Future Directions 46
References 48
Appendix A. Comments about the Smith-Waterman Algorithm 51
A.1 Algorithm Description 51
A.2 Deficiencies of the Smith-Waterman Algorithm 52
Appendix B. Optimized Baeza-Yates and Gonnet's Algorithm. 54
B.1 Error-bounded Edit Distance in Linear Time 54
B.2 Optimized Algorithm by Baeza-Yates and Gonnet 55
Nomenclature 62

List of Figures

2.1 Matching matrix and partial induced graph 11
2.2 Examples of legal and illegal edit transcripts 12
2.3 Target square 18
2.4 Diagonals of the matching matrix 19
2.5 Optimization Theorems 19
2.6 Search space reduction 22
2.7 Saving the best error number for each expanded path 26
2.8 Pseudocode of APBT. 28
2.9 Running time for pair of similar viral RNA sequences. 32
2.10 Running time for pair of dissimilar viral RNA sequences 33
2.11 Effect of alphabet size on the performance of APBT algorithm . . 35

3.1 Algorithm *Filter*. 39
3.2 Generating the final output. 41
3.3 GUI to the program. 42
3.4 Multiple strings program output example 43
3.5 Input sets. 43
3.6 Running time statistics for set 1 44
3.7 Running time statistics for set 2 45

A.1 Mosaic effect of the Smith Waterman algorithm. 53
A.2 Shadow effect of the Smith Waterman algorithm 53

B.1 Error-bounded calculation of an edit distance between two strings. 55

B.2 Algorithm *BYG_U*. 56
B.3 Algorithm *continue_compare*. 57
B.4 Sample suffix trees 57
B.5 An example of algorithm $BYG + U$ 58

Acknowledgements

I would like to begin this thesis with some words of gratitude to all people who helped me and were close to me during this year of research. First of all, the people that made all this possible: my supervisors Dr.Alex Thomo and Dr.Chris Upton, who provided inspiration, encouragement, and support at all stages of this research. In particular, the great expertise and enthusiasm of Dr.Alex Thomo has been the key for the outcome of current thesis.

Alex also introduced me to Dr.Ulrike Stege, with whom we have spent a lot of fruitful time working our way through the interdisciplinary field of bioinformatics. I would like to warmly thank Ulrike for teaching me how to think and communicate with clarity and precision, and for her useful comments concerning this text.

I also offer my thanks to the Department of Computer Science, here at the University of Victoria, for providing an environment in which this research could be conducted.

Thanks are also due to all members of the Virus Bioinformatics lab, especially to Gord Brown, Aijazuddin Syed and Daniel Horspool, for their helpful advice concerning the design and the implementation of the multiple local alignment program.

And finally, I should like to thank my son, Andrew, for his patience and understanding.

Chapter 1

Introduction

The amount of data collected and stored in databases is growing with increasing speed. While computers were initially designed and used for numerical computations, a large proportion of the data is nowadays collected and processed in textual form. The sources of textual data can be very different, varying from documents in natural languages to the sequences of biological macromolecules.

Biological sequences represent the order of molecules (nucleotides or aminoacids) combined into long polynucleotide (DNA, RNA) or polypeptide (Protein) chains. Nucleic acids contain 4 different kinds of molecules, while proteins contain 20, and therefore biological sequences can be represented as strings over the alphabets of sizes 4 and 20 respectively. We will refer to such strings as *biosequences*.

The well established solutions for the efficient processing of natural language documents cannot be always applied directly to biosequences (10). Such sequences differ from natural language texts in the following aspects:

- Biological "texts" are continuous strings without clear division into segments ("words").
- Biological "words" are much longer than words in any natural language text. For example, the average protein length is around several hundreds amino acids, while large proteins can reach over a thousand amino acids. DNA chains are much longer than polypeptide sequences. For example, the

human genome contains about 3×10^9 base pairs, organized as 46 chromosomes. The 24 different chromosomes range from 50×10^6 to 250×10^6 bp (21). In comparison, the longest word in English is 45 letters long (*pneumonoultramicroscopicsilicovolcanoconiosis*)[1], while the average English word length is around 4.5 letters (15).

- Another difference is that even a "word" with the same meaning may be spelled in many different ways. A *mutation* is defined as a heritable change in the nucleotide sequence of DNA, caused by a faulty replication process. There are several kinds of point mutations: *substitution* - a change of one nucleotide into another in the DNA sequence; *insertion* - an addition of one or more nucleotides to the DNA sequence; *deletion* - a removal of one or more nucleotides from the DNA sequence. Mutations occur with every replication, and this leads to the state, when the parts of the sequence carrying similar information are undetectable by exact matching algorithms, since they contain a significant proportion of differences.

Effective algorithms for approximate string matching are crucial in this field, and such algorithms should take into account the nature of the input sequences.

Biological data mining, being a topic of extensive research in the last few decades, still faces a great deal of open problems. One of those problems can be described as the extraction of the local regions of high similarity from a set of biosequences, and this problem takes an important place in the field of comparative sequence analysis.

1.1 Motivation

We are motivated to solve the problem of finding local similar regions (in biosequences) because such information is crucial in gaining insights about many biological problems of great practical significance.

[1] A factitious word alleged to mean 'a lung disease caused by the inhalation of very fine silica dust'(12)

Namely, the biological theoretical motivation behind such a search is based on two simple assumptions:

1. The high degree of similarity implies common function; therefore similar regions inside long DNA molecules can be known functional units.

2. The most important and successfully evolved elements of genomes are preserved in the course of evolution, therefore we can discover unknown important functionality.

In the event that we find similar regions among a set of biosequences, we can consider them (the similar regions) as candidates for genes, promoters and other important functional units. If we take a single common region and study its function in detail, then we can estimate the functionality for the rest of the regions, which are similar to that one.

The availability of fully sequenced genomes makes it possible to initially localize genes by comparing the local similarity degree of two un-annotated genomes (e.g. human and mouse)(11). The regions of high local similarity are potential exons. Those areas are the place to start a search for new genes.

The most classic example of the search for similarities in biosequences is the establishment of an association between cancer and uncontrolled cell growth (5). This discovery was enabled by finding the high-degree similarity between a sequence of the cancer associated gene and a sequence of the protein involved in cell growth. The high correlation between the two sequences showed the connection between cancer and cellular growth.

Another example can be taken from the field of virology. Highly conserved subsequences of 40 bp were found in virtually all Poxvirus species. These conserved subsequences were shown to function as a Poxvirus promoter element in infected cells (4). This means that these regions may be targeted when designing anti-viral drugs.

These impressive examples leave no doubt that efficient solutions to the problem of extracting highly similar regions from a set of biological sequences can be

of great help in gene prediction, regulatory sites identification, drug design and many other theoretical and practical biological fields.

In contrast to finding exact patterns, the extraction of approximate patterns is computationally much more demanding. Notably, Pevzner and Sze (13) consider this a "challenging problem." We analyze next two common approaches to this problem.

1.2 Local Alignment

The most widely used algorithm in molecular sequence comparison is the "local alignment" algorithm by Smith and Waterman (14). Their specialized dynamic programming method was designed to obtain highly conserved regions of two compared sequences.

Recent research shows, however, that this algorithm does not report all the highly similar regions, due to *mosaic* and *shadow* effects (1). The *mosaic effect* is the inability of the algorithm to discard poorly preserved regions. As a result, these poorly preserved regions end up with a score higher than more similar regions. On the other hand, *the shadow effect* of Smith Waterman algorithm is its tendency to lengthen long alignments with a high score, incorporating or "shadowing" during this process shorter alignments with a lower score but higher degree of similarity. A more detailed explanation and examples can be found in Appendix A.

In short, the Smith-Waterman algorithm is able to find some of the local similarities (among biosequences), but unfortunately a great deal of them may end up undiscovered.

Different and sometimes very complex solutions were proposed to improve the Smith-Waterman algorithm, mainly by normalizing the score of the local alignment by the length of the induced substrings (6; 14). The similarity score varies from method to method. However, it is easy to find examples, where none of the proposed scores can help to discover important similarities among

biosequences. Another problem is that all these methods are based on a dynamic programming variant, which runs in quadratic time and space (6), which is almost impossible to apply for very long sequences.

These deficiencies of the local alignment methods are best acknowledged by the statement of Egesioglu, Arslan, and Pevzner(1): "Surprisingly enough, we still do not have an efficient algorithm, that finds the local alignment with the best degree of sequence similarity."

1.3 The Alternative Approach to the Problem of Local Similarity

The alternative approach, which we study in this thesis, is the so called "all-against-all approximate matching," which asks for the edit distance between each pair of substrings in two given strings (3; 7). In a Computer Science parlance, we are given a set of strings $\{s_1, \ldots, s_n\}$, and we are asked to find all the n-tuples $(s'_1, \ldots, s'_n)$, where s'_i is a substring of s_i, for $i = 1, \ldots, n$, and such that the "distance" between s'_i and s'_j for each $i, j = 1, \ldots, n$ is small enough to consider them "similar." Here the similarity is measured by the well-known edit distance (7), which is given by the least number of "edit operations" needed to transform one (sub)string into another. The edit operations are: insertion of a letter, deletion of a letter, and substitution of a letter by another one. Often, these operations are "visualized" as "typing errors," and so, when s'_i and s'_j have a distance of K, we say that they approximately match with up to K errors. In practice, we set two constraints for the sought solutions to all-against-all approximate matching problem. One is naturally the maximum allowed number of errors for each match, and the other is the minimum allowed length of substrings in a solution.

All-against-all approximate matching is computationally even more demanding than the local alignment approach, and despite past attempts it is still not efficiently solved. In the following discussions, we focus on the case, where only two strings are given as input to the problem. We discuss the extension to the

case of more strings in Chapter 3.

First, let us discuss why the problem is difficult. Observe that for two given strings s and t of lengths N and M respectively, a naive approach to "all-against-all approximate substring matching" is to exhaustively test each pair of substrings from s and t respectively. This naive approach requires the computation of $O(NM)$ cells of dynamic programming table for each NM pair of possible starting locations, yielding a time complexity of $O(N^2M^2)$.

1.4 Previous Work

The best known solution to the "all-against-all" problem was proposed by Baeza-Yates and Gonnet in (2) and (3). Their solution significantly improved the average time complexity of the naive approach, by avoiding the examination of repeating substrings. In their method, the two input strings are organized into a suffix tree structure, and the order of substrings comparisons is guided by a depth-first traversal of the suffix tree nodes. The time complexity based on their practical results lies between NM (best case) and N^2M^2 (worst case), but closer to N^2M^2 (see (7)). Due to the significant length of compared strings, the algorithm can not be considered feasible.

Setting threshold criteria bounding the error number, e.g., allowing at most K differences in an approximate substring match, significantly improves the performance of the algorithm by Baeza-Yates and Gonnet. This is because the value of K can be directly incorporated into their algorithm to cut down the depth of the suffix tree traversal. As we verify through detailed experiments, for small values of K, Baeza-Yates and Gonnet's algorithm performs very well. However as K increases, the number of suffix tree nodes that are examined grows almost exponentially in K as observed also by Ukkonen in (17).

1.5 Our Contribution

In this thesis, we propose a new algorithm, which is computationally better than Baeza-Yates and Gonnet's algorithm by an order of magnitude.

We cast the original problem into the problem of finding "maximal paths" (to be defined precisely later) in a special "matching" graph. Via a careful study of this graph, we are able to derive interesting and useful properties that help us in devising a highly optimized depth-first search procedure for finding "maximal paths," which correspond to the solutions of the original string problem.

Our proposed algorithm runs in $O(NMK^2)$ time, which is a significant improvement over Baeza-Yates and Gonnet's algorithm. Moreover, we experimentally show that our algorithm scales linearly in K (without reaching our quadratic upper bound), and it outperforms Baeza-Yates and Gonnet's algorithm by an order of magnitude.

Finally, we stress that our algorithm has an additional nice feature: it reversely depends on the alphabet size. As the size of the alphabet grows, the running time of our algorithm decreases considerably. This is contrary to the behavior of Baeza-Yates and Gonnet's algorithm. Its running time worsens with the increase of the alphabet size.

Our contribution is a fast algorithm for solving the problem of "all-against-all approximate substring matching" for two strings. Clearly, the solution for two strings can be extended to more strings (see Chapter 3).

1.6 Structure of the Thesis

The remainder of this paper is organized as follows. Chapter 2 introduces the new graph model and describes the details of our new algorithm. We present performance results and compare them with an optimized variant of Baeza-Yates and Gonnet's algorithm. Chapter 3 describes the implementation of the algorithm for finding approximate matches for multiple strings. Chapter 4 closes

with conclusions and future work.

Chapter 2

A New Algorithm for All-against-all Approximate Substring Matching

2.1 A Graph Model

Let Σ be a finite alphabet. A sequence of letters $a_1 a_2 \ldots a_N$, where $a_i \in \Sigma$ (for $1 \leq i \leq N$), is called a *string* over Σ. We denote strings with $s, t, \ldots$. Given a string s, we denote its i-th letter with $s[i]$, and we denote a *substring* of s starting at position i and ending at some position j (where $i \leq j$) with $s[i, j]$. The letters of s at positions i and j are included in the substring $s[i, j]$, and thus $s[i, j]$ has length $j - i + 1$.

For any two strings s and t, we can "transform the first string into the second" by applying a sequence of the classical *edit operations*: insertion of a letter, deletion of a letter, and substitution of a letter by another letter. These edit operations are referred to as *errors*.

Let the *edit distance* for two strings s and t be the minimum number of edit operations needed to transform s into t, as in (7). Further, we say pair (s, t) is a *K-bounded approximate match* if the edit distance between s and t is at most K.

Now consider two strings s and t over Σ, with lengths M and N respectively. The problem we are trying to solve is to find all approximate substring matches

with at most K errors and of length greater than some threshold S.

Formally we have:

Problem 1 All error-bounded approximate matches

INPUT: *Strings s and t over alphabet Σ, and positive integers K and S.*

OUTPUT: *All error bounded approximate maximal matches $(s[i,j], t[k,l])$ such that*

1. *the edit distance between $s[i,j]$ and $t[k,l]$ is at most K and*
2. *the lengths of both $s[i,j]$ and $t[k,l]$ are at least S.*

Observe that in the above definition we are looking for maximal matches, with respect to the length. In other words, we do not report "sub-matches" satisfying the above two conditions. This is not a limitation, because the sub-matches can be easily obtained from the maximal matches.

We solve Problem 1 by casting it to an equivalent problem on graphs induced by a "matching matrix," which is defined as follows.

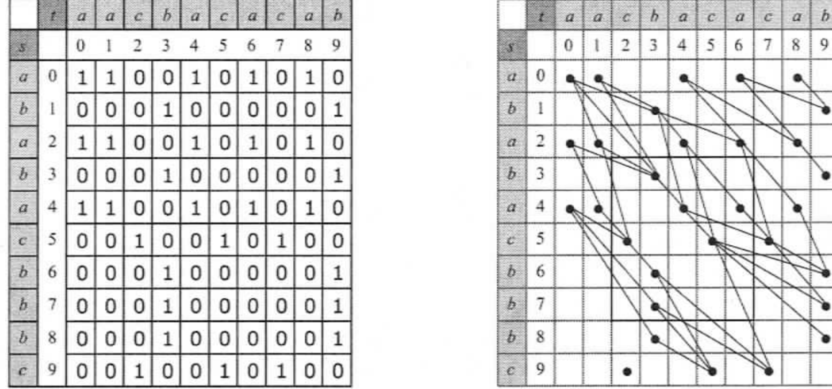
Let s and t be two given strings. We define the *matching matrix of s and t* ($\mathcal{M}_{s,t}$) as

$$\mathcal{M}_{s,t}[i,j] = \begin{cases} 1 & \text{if } s[i] = t[j] \\ 0 & \text{otherwise.} \end{cases}$$

We omit the subscripts whenever they are clear from the context. An example of a matching matrix for strings *ababa* and *aacbaca* is shown in Figure 2.1 (left).

Based on matching matrix $\mathcal{M}$, we define a weighted directed graph $G_{\mathcal{M}}$ with vertices v_{ij} corresponding to the 1-elements of the matrix, and with (directed) edges defined in a "top-down" and "left-right" fashion as follows: there is an edge $e(v_{ij}, v_{kl})$ if $i < k$ and $j < l$. Figure 2.1 (right) illustrates the nodes and some of the edges of the $G_{\mathcal{M}}$ graph based on the matrix $\mathcal{M}$ in the same figure (left).

We define the *cost* $c(v_{ij}, v_{kl})$ of an edge $e(v_{ij}, v_{kl})$ to be $c(v_{ij}, v_{kl}) = \max(k - i, l - j) - 1$. A path in graph $G_{\mathcal{M}}$ is a sequence of vertices connected by edges. For a path in $G_{\mathcal{M}}$, we define two characteristic properties: the "match length" and the "error number," which are as follows.



	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	1	1	0	0	1	0	1	0	1	0
b	1	0	0	0	1	0	0	0	0	0	1
a	2	1	1	0	0	1	0	1	0	1	0
b	3	0	0	0	1	0	0	0	0	0	1
a	4	1	1	0	0	1	0	1	0	1	0
c	5	0	0	1	0	0	1	0	1	0	0
b	6	0	0	0	1	0	0	0	0	0	1
b	7	0	0	0	1	0	0	0	0	0	1
b	8	0	0	0	1	0	0	0	0	0	1
c	9	0	0	1	0	0	1	0	1	0	0

Figure 2.1: Matching matrix and partial induced graph. Some edges of cost at most 3 are shown; the directions of edges are left out.

Definition 1 *Let π be a path starting at v_{ij} and ending at v_{kl}. Then:*

- *The* match length *of π is defined as $ML(\pi) = min(k - i + 1, l - j + 1)$.*
- *The* error number*, $EN(\pi)$, is defined as the cost of the path π, that is the sum of all costs of edges in π.*

In essence, a path from v_{ij} to v_{kl} outlines a sequence of edit operations that realizes an (approximate) match of $s[i, k]$ with $t[j, l]$. Note that $G_{\mathcal{M}}$ is *not* a dynamic programming (induced) graph (also called *edit graph* in (7)); DP graphs have been very well studied in the literature. However, to the best of our knowledge there is no work that formally studies the properties of the $G_{\mathcal{M}}$ graph.

Interestingly, graph $G_{\mathcal{M}}$ possesses a very desirable property. Namely, for any two vertices v_{ij} and v_{kl}, the error number of the shortest path in $G_{\mathcal{M}}$, going from v_{ij} to v_{kl}, equals the edit distance between the substrings $s[i, k]$ and $t[j, l]$.

Note that, although intuitively right, a formal proof needs special care.

In order to show the claimed property, we use the notion of an *edit transcript* (see (7), and Figure 2.2 on page 12 as an illustration). Each possible way to transform $s[i, k]$ into $t[j, l]$ using edit operations can be expressed by some edit transcript. In general, for two strings, there may be many different edit transcripts transforming one string into another. Clearly, for any possible pair

s	abce--d	abce-d	a-bced
t	a-ebced	aebced	aebced
Transcripts	MIFFDDM	MFFFDM	MFMMMM

Figure 2.2: Examples of legal and illegal edit transcripts. M stands for *match*, F stands for *substitution*, I stands for *insertion*, and D stands for *deletion*.

of substrings in s and t, we are looking for an edit transcript with a minimum possible number of edit operations.

Consider the substrings $s[i,k]$ and $t[j,l]$ with $s[i] = t[j]$ and $s[k] = t[l]$. Such substrings correspond to the path between two matches in the matching matrix (i.e. $\mathcal{M}[i,j]$ and $\mathcal{M}[k,l]$ are equal to 1).

Lemma 1 *There exists an edit transcript for $s[i,k]$ and $t[j,l]$ with cost at most $max(k-i, l-j)-1$.*

PROOF. Without loss of generality let $s[i,k]$ be shorter than $t[j,l]$. Then $max(k-i, l-j)-1 = l-j-1$, i.e. equal to the length of $t[j+1, l-1]$. Now, an edit transcript with the claimed cost can be obtained to reflect the following procedure.

First align $s[i,k-1]$ with the prefix of $t[j,l]$ for a cost of at most $(k-1)-i+1-1 = k-i-1$ substitutions (recall that $s[i] = t[j]$). Then, insert spaces at the end of $s[i,k-1]$ to fully align it with $t[j,l-1]$. Since $s[k] = t[l]$, we get a full alignment of $s[i,k]$ with $t[j,l]$. It is easy to see that the total cost of the transcript corresponding to this alignment is at most $l-j-1$ edit operations. □

In line with the above lemma, we define the concept of a legal edit transcript as follows. For this let *consecutive matches* in an edit transcript be two matches ("M's") without any other match in between.

Definition 2 *Consider an edit transcript for substrings $s[i,k]$ and $t[j,l]$. Further, let $s[x] = t[v]$ and $s[y] = t[w]$ for $i \leq x < y \leq k$ and $j \leq c < w \leq l$ be* consecutive matches *in this edit transcript. We call the edit transcript* legal *if the number*

of edit operations between any two consecutive matches in the edit transcript for $s[i, k]$ *and* $t[j, l]$ *does not exceed* $max(y - x, w - v) - 1$.

To illustrate, Figure 2.2 on page 12 (left) shows an example of an illegal edit transcript for the strings *abced* and *aebced*. The transcript is illegal because the number of edit operations (equal to 5) exceeds the total number of letters between the first and last positions of the longer string. The same figure (middle) shows a legal edit transcript for the same strings, while on the right is shown an edit transcript with the minimum possible amount of edit operations (edit distance).

We also observe that the number of edit operations between two consecutive matches $s[x] = t[v]$ and $s[y] = t[w]$ in an edit transcript cannot be less than (surprisingly the same bound as before) $max(y-x, w-v)-1$, regardless whether the transcript is legal or not. The reason is that there are no other matches between two consecutive matches. Therefore, the total number of edit operations cannot be less than the larger number of characters between two matches.

Thus, in a legal transcript, the number of edit operations between two consecutive matches $s[x] = t[v]$ and $s[y] = t[w]$ is in fact equal to $max(y-x, w-v)-1$.

Lemma 2 *Let s and t be two strings over alphabet* Σ. *For any legal edit transcript between two substrings* $s[i, k]$ *and* $t[j, l]$, *there exists a path* π *between vertices* v_{ij} *and* v_{kl} *in* $G_{\mathbb{M}}$ *such that* $EN(\pi)$ *is equal to the number of edit operations in this edit transcript.*

Proof. Let us consider some legal edit transcript for $s[i, k]$ and $t[j, l]$. We show that we can find a path in $G_{\mathbb{M}}$, which passes through vertices corresponding to the transcript matches, and whose error number is equal to the transcript cost (number of edit operations).

Let $s[x] = t[v]$ and $s[y] = t[w]$ be two consecutive matches in this transcript. From the above discussion, we know that the number of edit operations between these two matches (in the transcript) is equal to $max(y - x, w - v) - 1$.

By the construction of $G_{\mathcal{M}}$, we have that v_{xv} and v_{yw} are vertices in $G_{\mathcal{M}}$, and $e(v_{xv}, v_{yw})$ is an edge in $G_{\mathcal{M}}$. The cost of this edge, by definition, is equal to $max(y-x, w-v)-1$.

Hence, we can always find a path π from v_{ij} to v_{kl} with error number equal to the number of edit operations in our edit transcript. Namely, π consists of the edges connecting the vertices corresponding to the matches in the edit transcript. □

Now we are ready to prove our *characterization theorem.*

Theorem 1 *The edit distance between $s[i,k]$ and $t[j,l]$ is equal to the error number of the cheapest path(s) from v_{ij} to v_{kl} in $G_{\mathcal{M}}$.*

PROOF. The edit distance between $s[i,k]$ and $t[j,l]$ corresponds to the number of edit operations in some edit transcript. Such an edit transcript surely is a legal one, and it has the minimum possible number of edit operations. From this and Lemma 2, we conclude that the path corresponding to this edit transcript is a cheapest path from v_{ij} to v_{kl} in $G_{\mathcal{M}}$. □

Clearly, only the cheapest paths from v_{ij} to v_{kl} have an error number which is equal to the edit distance between $s[i,k]$ and $t[j,l]$. We call a path with error number less than or equal to K a *path below (error) threshold.*

Problem 2 ***All paths below threshold***
INPUT: *The graph $G_{\mathcal{M}}$ for two strings s and t, and positive integers K and S.*
OUTPUT: *All maximal paths below threshold K, and with match length at least S.*

Based on Theorem 1 we conclude that:

Corollary 1 *The problem* all bounded approximate matches *can be reduced to the* all paths below threshold *problem in a linear time with respect to the size of the output.*

We show in Subsection 2.2.1 how to construct, in linear time and space, an instance for *all paths below threshold* from an instance of *all bounded approximate matches*.

What do solutions for *all paths below threshold* look like? We remark that in $G_{\mathcal{M}}$ there can exist multiple maximal paths below threshold connecting the same two vertices v_{ij} and v_{km}. However, we need to detect only one such path in order to produce $(s[i,k], t[j,m])$ as an approximate match. Based on the properties of $G_{\mathcal{M}}$ that we show in the next section, we devise a highly optimized depth-first search approach, which takes special care of path expansions and overlap. In this way we avoid shortest path methods, which are associated with high overhead and rigidity of path expansions. Still we produce best maximal paths, even without using shortest path methods.

Further, observe that every solution path π is a path between two character matches. Thus, the transcript for a substring match $(s[i,k], t[j,m])$ starts and ends with a character match, and solutions having an error number less than K can be extended by up to $K - EN(\pi)$ end gaps. It is easy to see that, from the solution set for Problem 2, all these solutions for Problem 1 can be produced in linear time.

We conclude this section with a remark on the maximality of the paths versus the maximality of the solutions to Problem 1. Namely, each maximal solution to Problem 1 has a corresponding maximal path, which is a solution to Problem 2. However, there are maximal paths produced as solution to Problem 2, which correspond only to sub-solutions (i.e. non-maximal solutions) to Problem 1. We illustrate such cases in Subsection 2.2.3. Also, although sub-solutions can be eliminated as a post-processing step, we show in Subsection 2.2.3 how to eliminate the paths producing sub-solutions during the main processing of our algorithm.

2.2 Solving "All Paths Below the Threshold"

In this section we present algorithm All Paths Below the Threshold (APBT) that is time-sensitive to threshold K and solves Problem 2 in time $O(NMK^2)$.

In our search for all maximal paths below threshold K, we use an optimized depth-first search. We scan the matching matrix in row-major order, i.e., we scan the first row from left to right, then the second etc. When a vertex of $G_{\mathcal{M}}$ is encountered during the scan of $\mathcal{M}$, we initialize a path π with $EN(\pi) = 0$ and $ML(\pi) = 1$. The algorithm then builds all the possible expansions of this initial path by adding one vertex at a time and by keeping track of the best paths found so far. Such an optimal path is used as a bound for future candidates. As paths are constructed, the algorithm examines each partially completed path π: If no more vertices can be added without exceeding threshold K, then we stop the further expansion of π, and check whether $ML(\pi) \geq S$. If true, then we consider path π *completed* and report it as a solution. Otherwise, path π is *aborted.* Note that we also stop expanding a path when the last row or last column of the matching matrix $\mathcal{M}_{s,t}$ is reached.

2.2.1 Constructing the Matching Matrix

Although the comparison of two characters of given strings takes a constant time, we explicitely build $\mathcal{M}_{s,t}$, since the scan of a boolean array speeds up the computation.

We build $\mathcal{M}_{s,t}$ in linear time and space (so the preprocessing is as cheep as possible) as following.

Let $|\Sigma|$ be the size of the alphabet. We create $|\Sigma|$ bit arrays, BA_a, for each letter a in alphabet Σ. Each such array has N bits (elements) initially set to 0. Now, we scan the string t from left to right. When we read a letter a, say at position i of t, we set to 1 the ith bit in the corresponding BA_a array. At the end of scanning each array BA_a will record all the positions in t where the

corresponding letter a occurs. Formally,

$$BA_a[i] = \begin{cases} 1 & \text{if } t[i] = a \\ 0 & \text{otherwise.} \end{cases}$$

After this, we scan the other string s. For each letter a that we read from s, we create a pointer to the previously constructed array BA_a. At the end of scanning we obtain an array of M pointers, each pointing to some bit array. Note that we only scan once both s and t, and the total memory we need for $\mathcal{M}_{s,t}$ is $O(|\Sigma| \cdot N + M)$, where M (the length of s) is the number of pointers to the bit vectors.

Let's turn our attention to graph $G_{\mathcal{M}}$. We stress here that we never explicitly construct and store it, remaining so linear with respect to space. Rather, as we show, we traverse it by constructing the needed paths "on the fly."

2.2.2 A Single-Step Path Expansion

Next we describe in detail a single-step path expansion. Since the error number of a path cannot exceed K, an edge to be appended to a path clearly has to have a cost of at most K. As a consequence, all edges in $G_{\mathcal{M}}$ of cost higher than K are excluded from further consideration.

Consider a path π with error number $EN(\pi)$, which ends at vertex v_{ij}. From the above discussion, it is clear that for a single-step expansion of π we need to search (in $\mathcal{M}$) for a possible "next vertex" only inside square $ABCD$, where $A = (i+1, j+1)$, and $C = (i+1+\kappa, j+1+\kappa)$, for $\kappa = K - EN(\pi)$. We call square $ABCD$ the *target square* for path π at vertex v_{ij} (see Figure 2.3 on page 18 left).

Note, that as the *error number* of the growing path increases, the area of the corresponding target square decreases (see Figure 2.3 on page 18 right).

On the first sight it seems that for any vertex v_{ij} in $G_{\mathcal{M}}$ there are at most $(\kappa+1)\times(\kappa+1)$ outgoing edges to be considered. Next, we show how to reduce the number of edges for consideration even further. Toward this end, we introduce the following definitions regarding diagonals in the matching matrix $\mathcal{M}$.

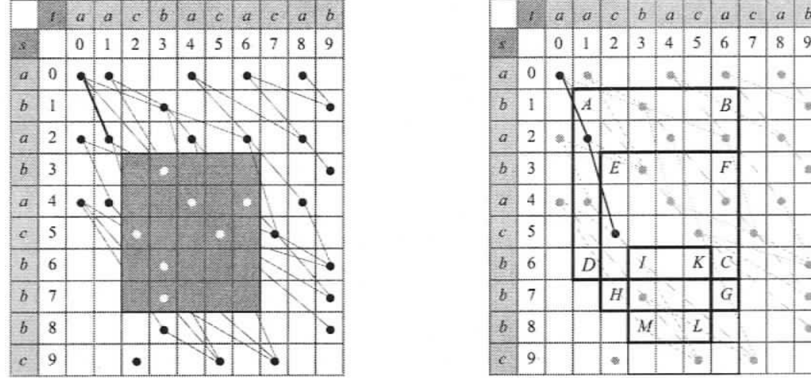


Figure 2.3: **[Left]** Target square for path π going from v_{00} to v_{21}, $K = 5$ and $\kappa = 4$. **[Right]** The area of the target square decreases as the error number increases. $ABCD$ is the target square for the path starting at v_{00}; $EFGH$ is the target square for the expanded path ending at v_{21}; $IKLM$ is target square for the further expanded path ending at v_{52}.

Definition 3 *Let (i, j) be an arbitrary cell in the matching matrix $\mathcal{M}$.*

1. *The (i, j)-*main diagonal *for $\mathcal{M}$ is the sequence of $(i + p, j + p)$-cells in $\mathcal{M}$, where $0 \leq p \leq min\{M - i, N - j\}$.*

2. *Let q be a value between 0 and $N - j$. The (i, j)-q-*upper diagonal *is the $(i, j + q)$-main diagonal.*

3. *Let r be a value between 0 and $M - i$. The (i, j)-r-*lower diagonal *is the $(i + r, j)$-main diagonal.*

Figure 2.4 illustrates the above definitions. Note that the (i, j)-main diagonal can be considered as (i, j)-0-upper diagonal as well as (i, j)-0-lower diagonal.

For the further development of our method, it is useful to visualize the cells on a given diagonal as points on a two-dimensional plane. Then, the (given) diagonal can be visualized as a straight line passing through these points, and forming a 45 degree angle with the horizontal and vertical axes. (cf. Figure 2.4).

Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) < K$. Now, assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, which

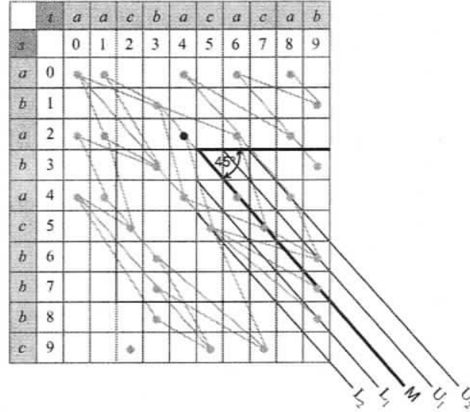


Figure 2.4: Diagonals for vertex v_{24}. M stands for *main diagonal*, L_2 for (2, 4)-2-*lower diagonal*, and U_1 for (2, 4)-1-*upper diagonal.*

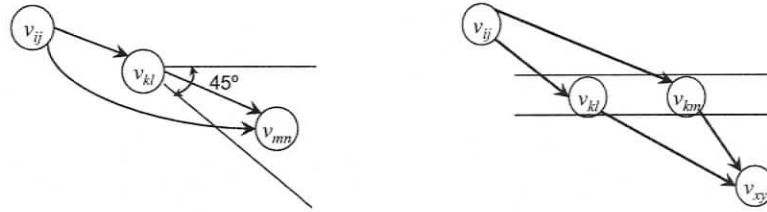


Figure 2.5: Optimization Theorems. **[Left]** Reverse Triangle Inequality: $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{mn}) < c(v_{ij}, v_{mn})$ **[Right]** Two-edge paths between v_{ij} and v_{xy}. We show that $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = c(v_{ij}, v_{km}) + c(v_{km}, v_{xy})$.

satisfy the following conditions:

(C1) $i < k < m$ and $j < l < n$, **(C2)** $l - j \geq k - i$, and **(C3)** $n - l \geq m - k$.

From condition **C1**, we know that the edges $e(v_{ij}, v_{kl})$, $e(v_{kl}, v_{mn})$, and $e(v_{kl}, v_{mn})$ do exist in $G_{\mathcal{M}}$. Condition **C2** is equivalent to $\frac{k-i}{l-j} \leq 1$ and therefore to $\frac{k-i}{l-j} \leq \tan(45°)$. This means, that the line segment connecting (k, l) and (i, j) forms an angle less or equal to 45° with the horizontal axis (see Figure 2.5 on page 19 left). We conclude that position (k, l) lies on an upper diagonal with respect to (i, j).

Reasoning similarly, condition **C3** implies that position (m, n) lies on an upper

diagonal with respect to (k, l). We show the following theorem.

Theorem 2 *(Reverse triangle inequality) Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) \leq K$. Further assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, satisfying conditions* **C1**, **C2**, *and* **C3**. *Then*

$$c(v_{ij}, v_{kl}) + c(v_{kl}, v_{mn}) < c(v_{ij}, v_{mn}).$$

PROOF. From the definition of $c(_,_)$ and based on **C2**, we get

$$c(v_{ij}, v_{kl}) = max\{k - i, l - j\} - 1 = l - j - 1.$$

Similarly, **C3** yields

$$c(v_{kl}, v_{mn}) = max\{m - k, n - l\} - 1 = n - l - 1.$$

By adding up **C1** and **C2** we obtain $m - i \leq n - j$. Thus,

$$c(v_{ij}, v_{mn}) = max\{m - i, n - j\} - 1 = n - j - 1.$$

Finally, adding up the expressions for $c(v_{ij}, v_{kl})$, $c(v_{kl}, v_{mn})$, and $c(v_{ij}, v_{mn})$ yields

$$l - j - 1 + n - l - 1 < n - j - 1 \equiv \; -1 < 0.$$

□

Based on the above theorem, we state the following corollary, which captures our first optimization regarding the single-step path expansions.

Corollary 2 *Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) \leq K$. Further assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, satisfying conditions* **C1**, **C2**, *and* **C3**. *Then extending path π to vertex v_{kl} and then to v_{mn} is cheaper than directly extending π to v_{mn}.*

The above corollary implies that if we build an edge from v_{ij} directly to v_{mn}, we "ignore" vertex v_{kl}, and unnecessarily increase $EN(\pi)$. Rather, we better expand path π to v_{kl} and later on, in the next round, continue to v_{mn}.

Notably, we obtain analogous results for the symmetrical conditions when the vertices v_{kl} and v_{mn} are in the lower part of the target square.

Let us take a more careful look at Corollary 2. Practically, it says that if we find a vertex v_{kl}, which lies on an upper diagonal of the target square, then we can exclude from the search for expansion all the triangular area of the target square, which is bounded by

1. row k (exclusive), and
2. the upper diagonal passing through v_{kl} (inclusive).

Symmetrically, if vertex v_{kl} lies on some lower diagonal, then we can exclude from the search for expansion all the triangular area of the target square, which is bounded by

1. column l (exclusive), and
2. the lower diagonal passing through v_{kl} (inclusive).

If vertex v_{kl} lies on the main diagonal of the target square, then both triangular areas are excluded at once.

We strengthen the search space reduction for path expansions even further. Consider two vertices v_{kl} and v_{km} in the same row of the target square for a path π at vertex v_{ij}, such that

1. $l - j \geq k - i$, i.e. v_{kl} lies on an upper diagonal, and
2. $m > l$, i.e. v_{kl} is closer than v_{km} to the main diagonal (see Figure 4 middle).

Now, let v_{xy} be any vertex inside the target square for some path π', an extension of π that is ending at vertex v_{km}. We show that

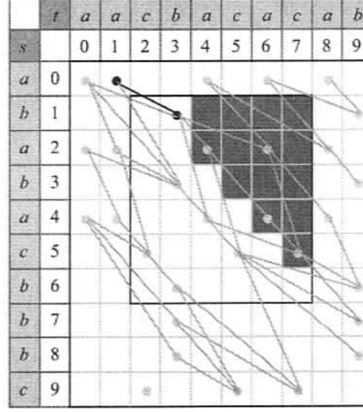


Figure 2.6: Search space reduction: None of the grey cells of the target square for v_{01} is tested after v_{13} is encountered.

Theorem 3 *Let vertices v_{ij}, v_{kl}, v_{km}, and v_{xy} be as above. The error number of the two-edge path from v_{ij} to v_{xy}, which passes through v_{km}, is equal to the error number of the two-edge path from v_{ij} to v_{xy}, which passes through v_{kl}, i.e.*

$$c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = c(v_{ij}, v_{km}) + c(v_{km}, v_{xy}).$$

Hence, it is sufficient to collect only the path closer to the main diagonal.

PROOF. $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = (l - j - 1) + (y - l - 1) = y - j - 2$, and
$c(v_{ij}, v_{km}) + c(v_{km}, v_{xy}) = (m - j - 1) + (y - m - 1) = y - j - 2.$ □

Observe the following implication based on Theorem 3. If a vertex v_{kl} is detected on an upper diagonal while scanning the target square for vertex v_{ij}, then the cells on the right of v_{kl} in row k can be safely excluded from further search for expansions.

Symmetrically, if a vertex v_{kl} is detected on a lower diagonal while scanning the target square for vertex v_{ij}, then the cells below v_{kl} in column l can be safely excluded from further search for expansions.

Based on Corollary 2 and the above discussions about Theorem 3 (and its symmetrical case), we present the following optimization.

Optimization 1 *In search for expansions, we scan the cells of the target square in a diagonal-major order, that is: First scan the main diagonal, possibly excluding parts of the target square from further scan. Next, scan the remaining of the target square through the 1-upper diagonal and the 1-lower diagonal, possibly excluding other areas of the target square. Then, continue with the 2-upper diagonal and the 2-lower diagonal and so on.*

Observe that, the scanning of a target square in this order guarantees that the exclusion of triangular areas takes place *as early as possible.* Figure 2.6 illustrates this search space reduction.

We state two important corollaries that follow from the above discussion.

Corollary 3 *Single path extension from an arbitrary vertex v_{ij} in $G_{\mathcal{M}}$ is performed at most once for each of the $2K+1$ diagonals surrounding v_{ij}, and therefore the number of possible extensions for v_{ij} is bounded by $2K+1$.*

Corollary 4 *An arbitrary cell of a matrix $\mathcal{M}[i,j]$ is accessed at most once from each of $2K+1$ diagonals. This also implies that an arbitrary vertex v_{ij} serves as a path extension for at most $2K+1$ vertices.*

2.2.3 Interdependence of Paths in the Graph

We show next how the information from previously explored paths can be reused. Consider the situation that, while scanning the matching matrix, two encountered (sub)paths start at different vertices end at the same vertex.

Let π_1 be the previously explored path, which connects vertex v_{ij} with v_{mn}. Now, let π_2 be another path that we are currently exploring, which originates in v_{kl}, and is built up to vertex v_{mn}.

The question is whether we should further expand π_2, or safely abort it without loosing any solution. We distinguish three possible cases for π_1 and π_2 with respect to their match length and error number and decide whether or to expand or abort π_2.

Case 1. $EN(\pi_2) < EN(\pi_1)$.

We expand π_2, since π_2 offers a further or better solution than the one discovered so far.

Case 2. $EN(\pi_2) \geq EN(\pi_1)$ and $ML(\pi_2) \leq ML(\pi_1)$

In this case all possible expansions of π_2 from v_{mn} are subsets of already tested expansions of π_1. The part π_2 up to v_{mn} has shorter match length that π_1, and therefore no new information can be obtained by expanding π_2. Path π_2 can be aborted.

Note that in this way we may ignore some maximal path, but such a path would only correspond to a non-maximal solution for Problem 1.

To illustrate please consider again Figure 2.1 (right). The path v_{00}, v_{33}, v_{44} serves as π_1 ($EN(\pi_1) = 2$), which is explored earlier in a row-major order. On the other hand the path v_{01}, v_{13}, v_{44} serves to exemplify π_2 ($EN(\pi_2) = 3$). Clearly, path π_2 will only offer a sub-solution to the solution corresponding to π_1, since the substring $t[1, 4]$ is a substring of $t[0, 4]$.

Case 3. $EN(\pi_2) \geq EN(\pi_1)$ and $ML(\pi_2) > ML(\pi_1)$

In this case π_2 follows the trails of some already tested expansions of π_1, but by extending π_2 we can obtain a longer path, and possibly contribute a new solution.

Case 3 is the only case where both a path expansion or a path abortion maybe the right decision. Recall, that the match length of a path is defined as the length of the shorter of the vertical ($s[_, _]$) and horizontal ($t[_, _]$) corresponding substrings.

We consider two different scenarios for Case 3 with respect to the match length.

Scenario 1. $ML(\pi_2)$ is defined by the vertical substring. Since the matching matrix is processed in a row-major order, $ML(\pi_2)$ cannot be greater than

$ML(\pi_1)$. This is because both paths end at the same vertex, and π_2 starts at the same or later (greater) row than π_1.

Scenario 2. $ML(\pi_2)$ is defined by the horizontal substring. In this case only, $ML(\pi_2)$ can be greater than $ML(\pi_1)$. This is because both paths end at the same vertex, and π_2 may start at a smaller column than π_1.

Notably, Scenario 2 becomes Scenario 1, if we repeat the computation in a column-major order. Thus, we abort the expansion of π_2 in Case 3. To avoid missing solutions we repeat the whole algorithm in column-major order. It is clear that by taking the union of the solution sets of the two runs of the algorithm yields the final solution set.

Optimization 2 *For each vertex v_{mn} we remember the smallest error number of the paths that reached this vertex. Each expansion from a particular vertex starts by checking, whether the path constructed so far has an error number greater or equal to an error number stored for v_{mn}. If so, the current path is aborted.*

Then, the computation is repeated in column-major order.

A last note is about storing the information regarding the "so far" best error number of paths ending at the examined vertices. As we are looking for local similarities, we solved this problem by setting an (artificial) upper bound $S_{max} > S$ for the matches. In such a way, we can use $N \cdot S_{max}$ bounded memory in the form of a rotating two-dimensional array A. Namely, a row of A corresponds to a row of $\mathcal{M}$. Initially, rows 0 to S_{max} of A correspond to rows 0 to S_{max} of $\mathcal{M}$, i.e. the best path error numbers for a vertices $v_{i_}$, where $i < S_{max}$, are stored in row i of A. When we finish processing row 0 of $\mathcal{M}$, then we recycle row 0 of the array to store the best path errors for the vertices of row S_{max}, and so on. In general, when we want to ask for the best path error for a vertex v_{ij} we consult the row $i \ mod \ S_{max}$ of A (see Figure 2.7 as an example).

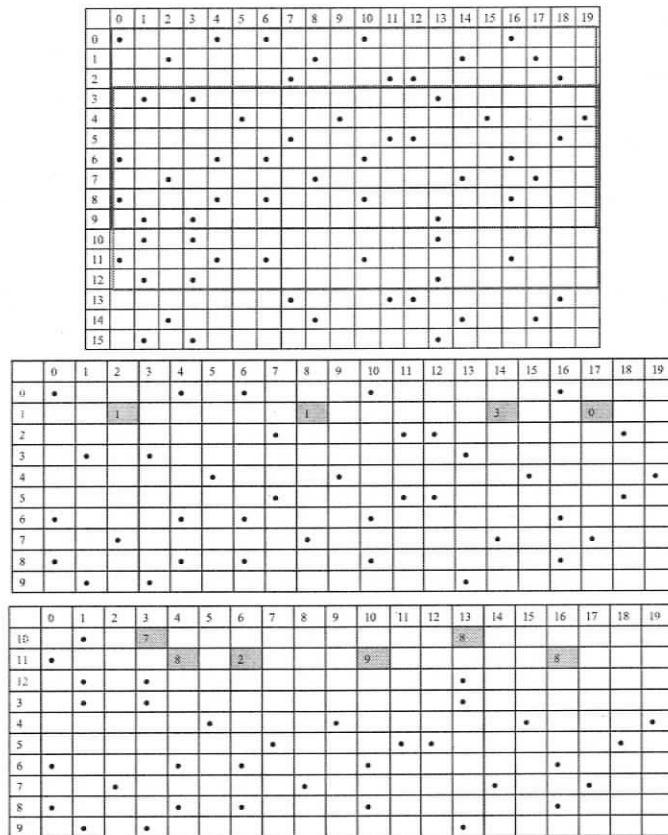


Figure 2.7: An example of a rotational array A for saving the best error number for each expanded path. $S_{max} = 10$. When row 2 has been processed, the first 3 rows of an array can be used to save the best error numbers for vertices in rows 10-12.

We stop expanding the solution if the length exceeds S_{max}. In practice, we choose $S_{max} = 500$ which seems high enough, since the matches never exceeded it in our experiments.

In practice, we process the matrix $\mathcal{M}$ by chunks of size $1000 \times N$, yielding the space for A to be $(1000 + S_{max}) \times S_{max}$, in order to never exceed the linear space while processing two long strings.

We stress that setting an artificial upper bound S_{max} is not really a limitation of our algorithm. If there exist oversize matches, then they can be easily obtained as a post processing step in linear time with respect to the size of output.

From the above, we can formulate the following lemma.

Lemma 3 *Each vertex in the graph is expanded at most* $2(K + 1)$ *times.*

PROOF. $K + 1$ is the maximum number of different possible error numbers, and factor 2 stems from the two runs of the algorithm. □

Theorem 4 *Algorithm All Paths Below Threshold has a time complexity of* $O(NMK^2)$.

PROOF. Since during path extension, any cell of the matrix is accessed only once from at most $2K + 1$ vertices (cf. Corollary 4) and each of these cells, if it is a vertex, is expanded at most $2(K+1)$ times (cf. Lemma 3), the upper bound for traversing a particular cell of the matrix is at most $2(K+1)(2K+1)$. Since there are at most MN many cells in $\mathcal{M}$, the total time complexity is $O(NMK^2)$. □

The pseudocode of our algorithm is given is Figure 2.8.

2.3 Experimental Evaluation

In this section, we present an experimental evaluation of our *All Paths Below Threshold* algorithm as it compares with the well-known algorithm by Baeza-Yates and Gonnet (3).

```
All_paths_below_threshold(M,K,S)
  scan M in row major order
  if M[i,j] = 1 then
    create a single-vertex v_ij path π
    EN(π): = 0
    Expand_path(π)

  scan M in column major order
  if M[i,j] = 1 then
    create a single-vertex v_ij path π
    EN(π) := 0
    Expand_path(π)

Expand_path(π)
  if ML(π) ≥ S then
    add π to the set of solutions

  if a path with error number EN(π)
    has already been extended through
    v_kl then abort π and return

  Do a single-step expansion (if possible) of π
    creating new expanded path π_exp

  Expand_path (π_exp)
```

Figure 2.8: Pseudocode of APBT.

In fact, we implemented the Gusfield's (7) variant of the Baeza-Yates and Gonnet algorithm.[1] Also, we optimized it by an order of magnitude using Ukkonen's (16) error bounded dynamic programming method, which has linear time with respect to the length of the compared (sub)strings.

Briefly described, Gusfield's variant of the algorithm by Baeza-Yates and Gonnet is as follows. First, suffix trees are built for strings s and t. Then, these suffix trees are traversed, and for each pair of nodes, the edit distance between the substrings labeling the nodes is computed using dynamic programming. Notably, the last row and column of the dynamic programming table for two substrings corresponding to a pair of nodes (from the suffix trees of s and t respectively), are used to initialize the dynamic programming table for the substrings of the child nodes.

In an error-bounded variant, if all the values in the last column and last row of the dynamic programming table for the parent nodes exceed K, then we can safely abort traversing and testing the child nodes. This is true because of the non-decreasing nature of the dynamic programming table values. Early stopping

[1] The original code of (3) is unfortunately not available anymore (Personal communication with (2), (18)).

in this way tremendously improves the running time of the algorithm for small K. Without it, the use of suffix trees still involves the comparison of NM different pairs of suffices by calculating values of at most NM dynamic programming tables, having an $O(N^2M^2)$ algorithm in the worst case.

As mentioned above, since we are interested in an error-bounded version of the problem, we enhanced the original algorithm by using Ukkonen's linear time calculation of the error-bounded edit distance (see (16) and Appendix 2 for more details). In our implementation of Baeza-Yates and Gonnet algorithm, we compute cells only in the $2K + 1$ strip around the main diagonal of the dynamic programming table, and this reduces the worst case time to $O(N^2MK)$. We abbreviate this version of the algorithm as algorithm $BYG + U$. A more detailed description of the $BYG + U$ algorithm can be found in Appendix 2.

2.3.1 Setup

We have measured and compared the running time for different pairs of input sequences, on the same 1.2 GHz PC with 312 MB of RAM. We implemented both the *All Paths Below Threshold*($APBT$) algorithm and the $BYG + U$ algorithm in Java 1.5.

2.3.2 Input Sequences

In this sub-section we present a detailed description of the test data and the rationale for our choices. In total, six different pairs of strings were tested. These pairs were chosen from the following sequence types:

- RNA and DNA viral sequences. The alphabet size of such sequences is 4.
- Protein sequences. The alphabet size of such sequences is 20.
- Random strings over two alphabet sizes: 4 and 20. This because 4 and 20 present practically important alphabet sizes.

In detail, we studied the performance of $APBT$ and $BYG+U$ algorithms on the following pairs of sequences from (19):

Pair 1. RNA genomic sequences of two Human Coronaviruses 229E (27317 bp) and OC43 (30738 bp). We expect that they have a lot of local similarities (i.e. a large output size).

Pair 2. RNA genomic sequences of two viruses belonging to the different families: Coronaviridae (SARS coronavirus Tor2 - 30504 bp) and Togaviridae (Sleeping disease virus strain 45 - 11900 bp). Here we expect to get a small output.

Pair 3. Sequences of two proteins from the same protein family (Phospholipase), and belonging to two viruses: EEVBovine papular stomatitis virus strain BV-AR02 (378 aminoacids) and Canarypox virus strain ATCC VR111 (378 aminoacids). We expect a bigger output size than in pair 4.

Pair 4. Protein sequences of the Myxoma virus belonging to different protein families: MYXV-Lau-m004L (237 aminoacids) and MYXV-Lau-MT1 (260 aminoacids). We do not see why they should have local similarities.

Pair 5. Very long DNA's of two poxviruses: Myxoma virus strain Lausanne (161773 bp) and Swinepox virus strain Nebraska 17077-99 (146454 bp). We consider them in order to estimate the running time for bigger genomes.

Also, we generated two pairs of pseudo-random strings of length $1,000$, and $10,000$ respectively. The generated random strings were saved in order to compare the running time of both algorithms on exactly the same (random) input. We observed that the random strings represent the most dissimilar string pairs with the comparatively small output size.

2.3.3 Comparative Performance

In this sub-section we present the behavior of the $APBT$ and $BYG{+}U$ algorithms on the above sequences.

Let us start with Figure 2.9 on page 32, which represents the running times of $BYG+U$ and $APBT$ on **Pair 1** of RNA sequences belonging to viruses from the same family. Notably $APBT$ outperforms the $BYG+U$ algorithm for values of $K \geq 6$. We also show the size of the output, and this clearly shows that in order to obtain any output at all, even for similar RNA sequences, one has to set a bigger or equal to 6 value of K.

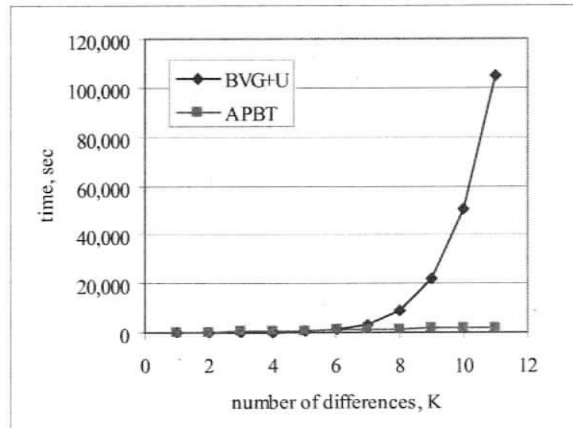
Interestingly, for K less or equal to 5, the $BYG+U$ algorithm outperforms $APBT$. This is because the $BYG+U$ algorithm benefits from the early stop of deeply going in the suffix trees, when the accumulated error exceeds K. However, as K grows the $BYG+U$ algorithm goes deeper in the suffix trees, and we observe an almost exponential in K increase in the running time. In contrast, our $APBT$ algorithm scales almost linearly with K although our worst case bound is quadratic with respect to K.

Now, let us focus on the behavior of **Pair 2** of the dissimilar RNA sequences (shown in Figure 2.10 on page 33). We observe very similar behavior as previously. For this input, algorithm $APBT$ outperforms $BYG+U$ starting from $K = 5$. This case demonstrates that $APBT$'s behaviour is not sensitive with respect to the output size. In contrast, $BYG+U$ is quite sensitive to the similarity degree of two strings, which is expected due to the use of suffix trees.

Algorithm $APBT$ performs even better on random strings, which represent the extreme case of dissimilarity. This can be explained by the fact that the less is the number of matches between two strings, the more sparse becomes our matrix $\mathcal{M}$ and the graph based on $\mathcal{M}$. In this case, the non-vertex cells of $\mathcal{M}$ constutite the majority, and according to Corollary 4, each non-vertex cell of $\mathcal{M}$ is accessed at most $O(K)$ times, and does not get expanded.

Also, we emphasize the fact that for alphabets of larger size, such as 20,

K	time, BYG+U	time, APBT	output, S=50
1	3 sec	2.9 min	0
2	5 sec	4.8 min	0
3	19 sec	7.4 min	0
4	1.6 min	10.3 min	0
5	6.3 min	13.7 min	0
6	19.2 min	16.4 min	2
7	55.0 min	19.7 min	5
8	2.5 hours	23.7 min	12
9	6.2 hours	27.9 min	28
10	14.1 hours	32.5 min	51
11	29.1 hours	37.2 min	104



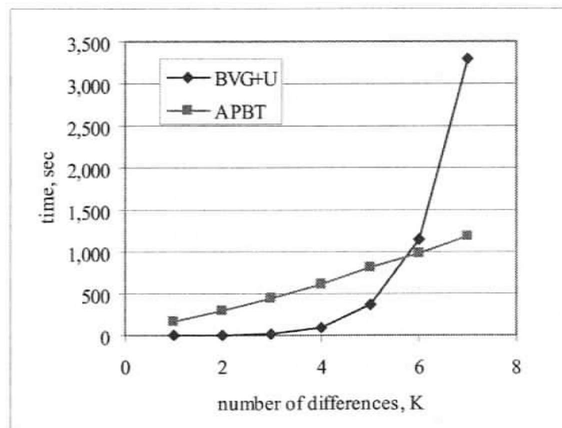
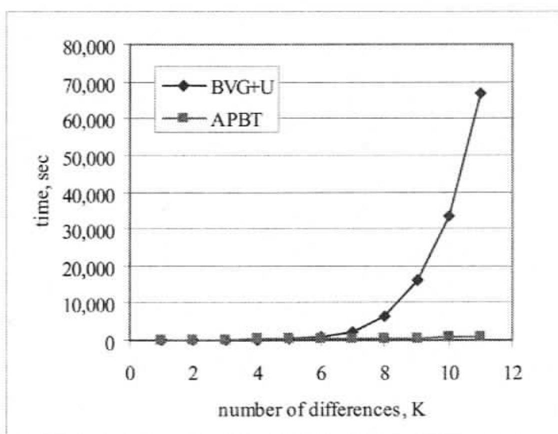


Figure 2.9: Running time for Pair 1 of viral RNA sequences: The last figure is a zooming of the figure in the middle for $K < 8$.

K	time, BYG+U	time, APBT	output, S=30
1	4 sec	65 sec	0
2	8 sec	1.8 min	0
3	18 sec	2.8 min	0
4	1.1 min	4.0 min	0
5	3.7 min	5.1 min	0
6	13.0 min	6.1 min	0
7	37.9 min	7.5 min	3
8	1.8 hours	9.0 min	89
9	4.4 hours	10.6.min	821
10	9.3 hours	12.8 min	5,680
11	18.6 hours	14.0 min	33,175



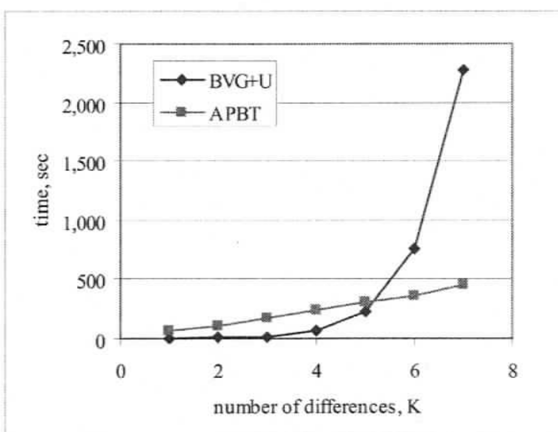
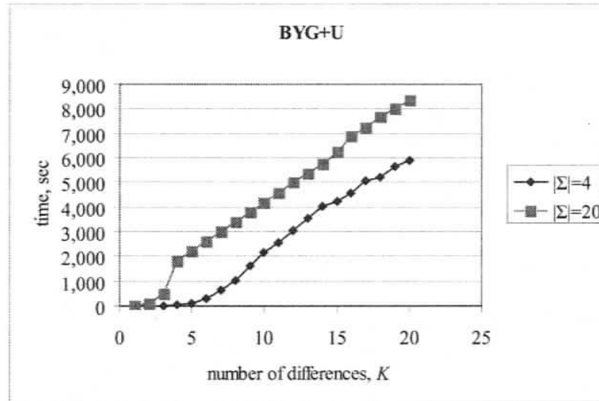
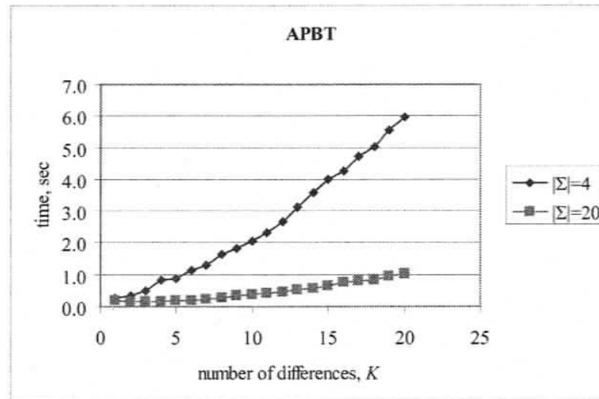


Figure 2.10: Running time for Pair 2 of viral RNA sequences: The last figure is a zooming of the figure in the middle for $K < 8$.

algorithm $APBT$ performs better than $BYG + U$, starting from K as low as 3. This can be explained by the greater "bushiness" of the suffix trees (used by algorithm $BYG + U$) close to the root, and by the fact that with the increase of the alphabet size, our matching matrix used in the $APBT$ algorithm becomes even more sparse. We observe this behavior in both the pairs of proteins as well as in the pair of random strings on an alphabet of size 20. Notably, the $APBT$ algorithm behaves so much better than the $BYG + U$ algorithm that we had to plot their behavior in different scales in order to have readable graphs.

Finally, for **Pair 5** of two long biological sequences of size 161,773 (Myxoma virus genome DNA) and 146,454 (Swinepox virus genome DNA), all matches of length at least 50 with up to $K = 10$ errors were produced by algorithm $APBT$ in 6 hours on the same (simple) machine. For this input set the $BYG+U$ algorithm was unable to produce results in reasonable time (i.e. 53 hours). Thus, we ran the $BYG+U$ algorithm on prefixes of length 1/10 of the **Pair 5** sequences. The output was produced after 29 hours.





K	BYG+U, dissimilar proteins	APBT, dissimilar proteins	BYG+U, similar proteins	APBT, similar proteins
1	5	0.02	3	0.03
2	12	0.02	7	0.02
3	19	0.02	19	0.03
4	32	0.02	45	0.04
5	37	0.02	61	0.04
6	42	0.02	64	0.07
7	47	0.03	69	0.05
8	52	0.04	71	0.06
9	60	0.03	84	0.07
10	63	0.04	96	0.13

Figure 2.11: Effect of alphabet size on the performance of APBT algorithm.[**Top, Middle**] Running time for two random strings of lengths 1000. *APBT* (middle) improves considerably when $|\Sigma| = 20$. This is in contrast with $BYG+U$ (top), which worsens when $|\Sigma| = 20$. [**Bottom**] Running time for two dissimilar (Pair 4) and two similar viral proteins (Pair 3).

Chapter 3

A Practical Application: All-against-all for Multiple Strings

3.1 Motivation

In the context of molecular biology, multiple strings comparison (of DNA, RNA, or protein strings) is more than a technical exercise. It is an effective tool for extracting biologically important, but faint, commonalities from a set of strings. Most of those commonalities can not be apparent when comparing two strings alone but may become clear when comparing a set of strings. Quoting from Arthur Lesk (8):"One or two homologous sequences whisper...a full multiple alignment shouts out loud."

The global multiple alignment for n strings is an NP-complete problem (20). The local multiple alignment problem is even more expensive than the global one, and it is also proven to be NP-complete (9). We develop a practical solution to the "all-vs-all" variant of local multiple alignment.

3.2 A Practical Approach

Based on our efficient and comparatively fast APBT algorithm for two strings, we developed a simple practical solution for a set of strings.

Formally, we solve the following problem:

Problem 3 *Given a set of strings* $\mathcal{T}=\{s_1, s_2, s_3, \ldots s_n\}$ *find all* n*-tuples of substrings (one from each string) having length at least* S*, and pairwise edit distance at most* K*.*

Next we outline the logic flow of the program and present its pseudocode.

3.2.1 Program Flow

Reducing Potential Starting Points.

We use the APBT algorithm for pairwise computation of approximate matches for each pair (s_i, s_j) in the input set. After the output - all maximal approximate matches - is produced, the positions along both s_i and s_j are marked as potential starting points of approximate matches for the entire set of strings. We have selected this simple marking technique in order to avoid storing the output of each pairwise computation, which can be quadratic in the worst case.

Based on marked positions, we find all maximal approximate matches for each pair of strings in the set, each time using only previously marked positions as an input. Note, that only for pair (s_1, s_2) the APBT algorithm is performed through the entire length of s_1 and s_2. The next comparison (s_1 with s_3) is performed starting only from previously marked positions. The matching matrix, used by the APBT algorithm, is well suited for such restricted computation. When starting to expand a path from a matrix cell, say (i, j), one needs to ask first, whether or not this cell is such that both i and j are marked positions in the corresponding strings.

After performing the comparison between each pair of strings in a set, input strings are reordered in ascending order of remaining starting points, We do this in order to reduce the start positions for the next iterations of the APBT algorithm. Then the computation repeats.

We continue to restrict the candidate regions by running the APBT algorithm for each pair of strings, until the number of start positions in all strings does not decrease significantly. The order of the pairwise calculations for reordered strings is as following: first string s_1 against $s_2, s_3, \ldots s_n$, then s_2 against $s_3, s_4, \ldots s_n$ and so on. The pseudocode of procedure *filter* which performs the initial reduction of starting points is given in Figure 3.1. The procedure *zero_Start_Positions()* simply checks if at least one start position in each $s_i \in \mathcal{T}$ is left after the current iteration. The computation may end at this step if there is no candidate start positions in one of the strings in $\mathcal{T}$, which means that for these threshold parameters, there are no matches among all the strings in $\mathcal{T}$. The procedure *no_Positions_Count_Decreased()* compares candidate positions counters for each $s_i \in \mathcal{T}$ before and after the current iteration. If no significant decrease in all counters values has occured, then the algorithm proceeds to the next step.

Generating a Set of Candidate Patterns.

The final set of starting points for each string is reordered for the last time according to the number of the marked positions, and algorithm APBT finds all maximal substrings for the two first strings in the ordered set. All substrings, which are potential multiple string matches of length at least S, are generated from this maximal solutions set. The duplicates are removed by hashing in order to avoid repeating computation for identical substrings. Then, each substring is tested against all strings, but only for the marked starting points in each string.

Generating the Final Output

The final output consists of n-tuples of substrings (one from each string) having length at least S, and pairwise edit distance at most K. On the other hand, from the candidate generation phase, we have only a set of candidate substrings, from which we have to generate the n-tuples. Thus, each candidate pattern is tested against each string s_i in the input set, by simple Pattern-against-all computation (7).

If there are no K-approximate matches for a pattern (substring from the

```
Filter(Input Set T,S,K): boolean
  n := size(T)
  for each s_i in T
   label(s_i) := 0
   for pos from 1 to length(s_i)
    start_positions(s_i)[pos]: = label(s_i)
  stop:=false
  do until (stop)
   for i from 1 to n − 1
    for j from i + 1 to n
     M:=build_Matching_Matrix(s_i,s_j)
     Solution_i_j:=APBT(M,start_positions(s_i),start_positions(s_j),K,S)
     for each pattern in Solution_i_j
      for pos from pattern(i_part).start to pattern(i_part).end − S
       if start_positions(s_i)[pos] ≥ label(s_i) then
        start_positions(s_i)[pos]: = label(s_i) + 1
     label(s_i) := label(s_i) + 1
      for pos from pattern(j_part).start to pattern(j_part).end − S
       if start_positions(s_j)[pos] ≥ label(s_j) then
        start_positions(s_j)[pos]: = label(s_j) + 1
     label(s_j) := label(s_j) + 1
   if zero_Start_Positions()
    return false
   stop:= no_Positions_Count_Decreased()
   do sort T in ascending order of start positions counters
  return true
```

Figure 3.1: Algorithm *Filter*.

previous phase) in one of the strings, the pattern together with all its previously collected matches is discarded. If there are ≥ 1 matches, the bunches of growing tuples are added rooted at the original candidate substring. Before the new match from s_i is added to the existing $i-1$-tuple, it is tested against all substrings already in the tuple, and if the edit distance between the new match and at least one substring exceeds K, the new i-tuple is not created. The entire procedure can be viewed as creating bunches of exactly n substrings, one from each string in $\mathcal{T}$. At each step i we have bunches of $i-1$ substrings. When the match is found in the ith string, the new bunch of i substrings is created. The previous bunches of $i-1$ substrings are discarded and replaced by the new ones of i substrings, until all n strings in $\mathcal{T}$ are tested.

As one can see in Figure 3.2, this process is exponential in the number of repetitions found for each candidate substring in each of n strings of $\mathcal{T}$. Therefore, the efficiency of the algorithm depends on the size of an initial candidate patterns set, and later on the number of patterns for each $s_i \in \mathcal{T}$, which approximately match all substrings already in the bunch.

Postprocessing of Discovered Patterns

It is clear, that since we test non-maximal substrings in order to not to miss any solution, we end up with non-maximal substrings in each group. It may happen, that substrings in two solution groups all start at the same positions in the input strings. The simple hashing by first position helps us to remove duplicate solutions, and we leave for each start position the solution with the biggest length of substrings.

In order to make the program as modular as possible, we did not use elaborate post-processing techniques. Instead, the final solution is saved as a binary object, and any further post-processing of the solution may be done easily taking this persistent object as an input.

We implemented a simple Compressor program, which compresses an initial solution set into a smaller set: two n-tuples are considered different only if start

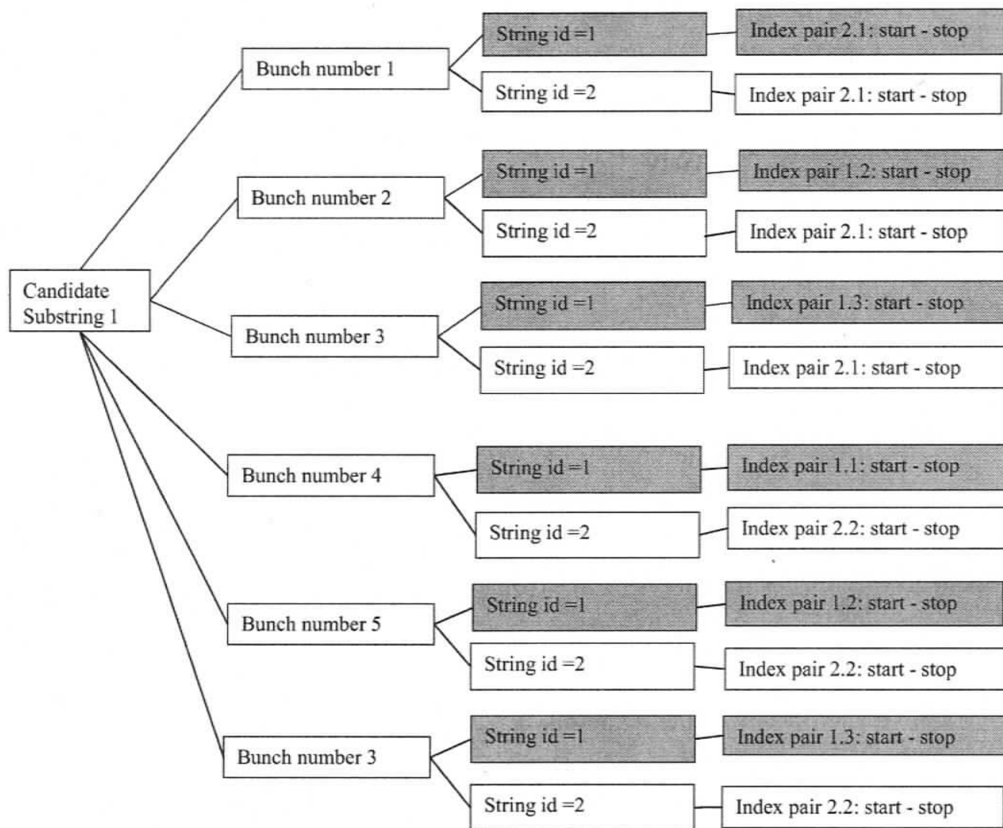


Figure 3.2: Generating the final output.

Figure 3.3: GUI to the program. "Select Parameters" window.

positions of each substring in two tuples differ more than the number of characters, specified by the user.

User Interface

The user inteface is intuitive and the "Select Parameters" window is shown in Figure 3.3. In general, the input sequences are supplied in a file in FASTA format. Then the user specifies the threshold parameters S-minimum length of the patterns, and K-maximum number of allowed errors. At this step the user also specifies the name of two different files for an output: one where the binary Hash table object is stored for further programming post-processing, and the second - in a readable tab-delimited format. An example of such a file (in Excel format) is shown in figure 3.4.

3.2.2 Program Evaluation

Experiments were performed on the same simple desktop machine with 1.2 GHz processor and 312 MB of RAM.

We have tested the program on 2 sets of viral genomes.

<table>
<tr><th>Pattern</th><th>Organism</th><th>Start</th><th>End</th><th>Length</th><th>Substring</th></tr>
<tr><td>pattern 1</td><td>>lcl|5|MYXV-Lau|NC_001132.2|Myxoma virus strain Lausanne</td><td>47952</td><td>47981</td><td>30</td><td>agacgatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 1</td><td>>lcl|49|DPV-W848_83|NC_006966|Deerpox virus strain W-848-83</td><td>55310</td><td>55340</td><td>31</td><td>agcactatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 1</td><td>>lcl|22|CPXV-BR|NC_003663.2|Cowpox virus strain Brighton Red</td><td>87466</td><td>87495</td><td>30</td><td>agacgatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 1</td><td>>lcl|33|YMTV-UNK|NC_005179|Yaba monkey tumor virus strain Unknown</td><td>42486</td><td>42515</td><td>30</td><td>aaactatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 1</td><td>>lcl|14|LSDV-Nee|NC_003027.1|Lumpy skin disease virus strain Neethling 2490</td><td>46790</td><td>46820</td><td>31</td><td>agaactatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 1</td><td>>lcl|16|SWPV-Neb|NC_003389.1|Swinepox virus strain Nebraska 17077-99</td><td>45610</td><td>45639</td><td>30</td><td>atactatcatttactaaggagtaaaatagg</td></tr>
<tr><td>pattern 2</td><td>>lcl|5|MYXV-Lau|NC_001132.2|Myxoma virus strain Lausanne</td><td>54154</td><td>54185</td><td>32</td><td>atatggatattaattgtagctatttaaatggg</td></tr>
<tr><td>pattern 2</td><td>>lcl|49|DPV-W848_83|NC_006966|Deerpox virus strain W-848-83</td><td>61635</td><td>61666</td><td>32</td><td>atatggttattaattgtagctatttaaatggg</td></tr>
<tr><td>pattern 2</td><td>>lcl|22|CPXV-BR|NC_003663.2|Cowpox virus strain Brighton Red</td><td>93727</td><td>93758</td><td>32</td><td>atatggatactaattgtagctatttaaatggg</td></tr>
<tr><td>pattern 2</td><td>>lcl|33|YMTV-UNK|NC_005179|Yaba monkey tumor virus strain Unknown</td><td>48779</td><td>48810</td><td>32</td><td>atatggatattaattgtggctatttaaatggg</td></tr>
<tr><td>pattern 2</td><td>>lcl|14|LSDV-Nee|NC_003027.1|Lumpy skin disease virus strain Neethling 2490</td><td>53125</td><td>53156</td><td>32</td><td>atatgggtattaattgtagctatttaaatggg</td></tr>
<tr><td>pattern 2</td><td>>lcl|16|SWPV-Neb|NC_003389.1|Swinepox virus strain Nebraska 17077-99</td><td>51992</td><td>52023</td><td>32</td><td>atgtggatattaattgtagctatttaaatggg</td></tr>
<tr><td>pattern 3</td><td>>lcl|5|MYXV-Lau|NC_001132.2|Myxoma virus strain Lausanne</td><td>104287</td><td>104325</td><td>39</td><td>taataacttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 3</td><td>>lcl|49|DPV-W848_83|NC_006966|Deerpox virus strain W-848-83</td><td>110382</td><td>110420</td><td>39</td><td>taataacttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 3</td><td>>lcl|22|CPXV-BR|NC_003663.2|Cowpox virus strain Brighton Red</td><td>143000</td><td>143038</td><td>39</td><td>taatatcttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 3</td><td>>lcl|33|YMTV-UNK|NC_005179|Yaba monkey tumor virus strain Unknown</td><td>96827</td><td>96865</td><td>39</td><td>taataacttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 3</td><td>>lcl|14|LSDV-Nee|NC_003027.1|Lumpy skin disease virus strain Neethling 2490</td><td>101902</td><td>101940</td><td>39</td><td>taataacttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 3</td><td>>lcl|16|SWPV-Neb|NC_003389.1|Swinepox virus strain Nebraska 17077-99</td><td>100172</td><td>100210</td><td>39</td><td>tagtaacttaaataactcatttatatattaaaaaatgtc</td></tr>
<tr><td>pattern 4</td><td>>lcl|5|MYXV-Lau|NC_001132.2|Myxoma virus strain Lausanne</td><td>104298</td><td>104328</td><td>31</td><td>ataactcatttatatattaaaaaatgtctgt</td></tr>
<tr><td>pattern 4</td><td>>lcl|49|DPV-W848_83|NC_006966|Deerpox virus strain W-848-83</td><td>110393</td><td>110423</td><td>31</td><td>ataactcatttatatattaaaaaatgtctat</td></tr>
<tr><td>pattern 4</td><td>>lcl|22|CPXV-BR|NC_003663.2|Cowpox virus strain Brighton Red</td><td>143011</td><td>143041</td><td>31</td><td>ataactcatttatatattaaaaaatgtcact</td></tr>
<tr><td>pattern 4</td><td>>lcl|33|YMTV-UNK|NC_005179|Yaba monkey tumor virus strain Unknown</td><td>96838</td><td>96868</td><td>31</td><td>ataactcatttatatattaaaaaatgtccgt</td></tr>
<tr><td>pattern 4</td><td>>lcl|14|LSDV-Nee|NC_003027.1|Lumpy skin disease virus strain Neethling 2490</td><td>101913</td><td>101943</td><td>31</td><td>ataactcatttatatattaaaaaatgtctat</td></tr>
<tr><td>pattern 4</td><td>>lcl|16|SWPV-Neb|NC_003389.1|Swinepox virus strain Nebraska 17077-99</td><td>100183</td><td>100213</td><td>31</td><td>ataactcatttatatattaaaaaatgtctat</td></tr>
</table>

Figure 3.4: An example of a compressed output for 6 complete genomes of poxviruses. $S = 30$, $K = 2$.

Coronaviruses		Poxviruses	
Sequence Name	Length	Sequence Name	Length
SARS coronavirus Tor2	29,729	Cowpox virus strain Brighton Red	224,499
Avian infectious bronchitis virus strain Cal99	27,693	Yaba monkey tumor virus strain Unknown	134,721
Human coronavirus OC43	30,738	Swinepox virus strain Nebraska 17077-99	146,454
Human coronavirus 229E	27,317	Lumpy skin disease virus strain Neethling 2490	150,773
		Deerpox virus strain W-848-83	166,259
		Myxoma virus strain Lausanne	161,773

Figure 3.5: Input sets.

- Set 1. 4 complete genomes of Coronaviruses (length approximately 30,000 bp).
- Set 2. 6 complete genomes of Poxviruses (length from 134,000 to to 224,500 bp).

The details of input sequences are shown in Figure 3.5.

Some running time statistics for set 1 are presented in Figure 3.6. The analysis of the presented results leads to the conclusion, that the most important factor in the program performance is not the percent of differences per se, but the probability of random occurence of common substrings with particular threshold

S	K	K/S, %	output size	time, min
15	1	7	0	8
15	2	13	101	34
20	2	10	1	21
20	3	15	109	25
20	4	20	772	37
30	5	17	21	31
30	6	20	246	35
30	7	23	1015	46
30	8	27	3232	59
30	9	30	7812	185
40	8	20	10	41
40	9	23	97	51
40	10	25	589	56
40	11	28	1530	70
40	12	30	4618	86
50	12	24	13	60
50	13	26	164	73
50	14	28	751	82
50	15	30	2503	97

Figure 3.6: Running time statistics for set 1.

parameters. Note, that the (somewhat) high running time for $S = 30$ with $K = 9$ differences is explained by a big size of the output. In such cases, in order to have a better running time, one can lower K or increase S.

Some performance results for set 2 are shown in Figure 3.7. For these very long strings the bottleneck remains the first run of algorithm APBT, which is quadratic on strings lengths.

In general, the program produces output in a reasonable time for most of the cases. The sequences for a comparison should not be too similar, and threshold parameters S and K should be adjusted according to a particular set.

3.2.3 Final Remarks about the Program

Advantages

S	K	output size	time, hours	compressed output	compressing parameter
30	2	691	5.4	4	10
30	3	5918	7.9	7	15
30	4	20491	11.0	14	15

Figure 3.7: Running time statistics for set 2.

1. The program produces the output in a feasible time (the maximum time on our simple machine did not exceed 3 hours for set 1, and 11.5 hours for set 2 with 8 sequences of several hundreds thousands bp length).
2. The multiple local alignments produced are more tractable: they are nothing else but the approximate matches with up to K mismatches, insertions and deletions.
3. The produced alignments represent the highest similarity regions between given strings.

Drawbacks

1. If the size of a candidate pattern set is larger than 1000 different substrings, it is practically impossible to build the final solution, due to exponential growth of the final output set. This happens only when the threshold is selected with too low minimum length S or/and with too big (more than 40 percent) allowed error number, or in case that all strings posses a great number of local similarities.
2. When the final output size is very big, further postprocessing methods should be applied in order to make use of discovered patterns.

Chapter 4

Conclusions and Future Directions

In this thesis, we studied the problem of the multiple local alignment - one of the most important problems in biological data mining. We focused on the all-against-all variant of the above problem. The main contributions of our work are the following:

- The new algorithm developed in this thesis presents a much more feasible solution for the All-against-all approximate substring matching problem for two strings. Namely the worst-case running time has been improved from $O(N^2M^2)$ to $O(NMK^2)$, where N and M are the lengths of strings, and K is a (comparatively small) constant.

- The application-driven program for the All-against-all approximate substring matching for a set of strings has been developed, and it is feasible and ready to use. This program is more suitable for uncovering homologous sequences, rather than for aligning homologous sequences.

Starting from the ideas and work presented in this thesis, several interesting issues for future research can be considered. These include:

- The *APBT* algorithm may be easily parallelized, and it does not require shared memory. The same linear in space matching matrix can be processed

independently at the same time for different sub-sets (starting indices) of the above matrix. This should make the new algorithm practical for any length of input strings.

- We conducted the performance experiments on viral genomes with the length of several hundreds of thousands bp. There may be need to search sequences much longer than that. Therefore, still quadratic in the lengths of input strings, the APBT algorithm requires the additional work in order to make it as efficient as possible (with the reasonable effort). For example, the dependance on the K threshold parameter can be reduced. We are working now on another algorithm, based on the same new graph model, which may help to improve the running time to $O(NM)$, where N and M are the lengths of two input strings.

- We left outside the scope of this thesis the question of the usefulness of all-against-all substring matching vs. local alignment in biological studies. This question should be answered by extensive experiments conducted by domain experts. The next steps in this direction can be:

  1. The determining of the statistical significance of the discovered patterns vs the rest of substrings pairs and vs the patterns generated for random strings.
  2. The analysis of the discovered patterns in order to determine which of them represent the new non-trivial similarities vs well-known common regions.

References

[1] Arslan A., Egesioglu O., and Pevzner P. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, **17**:327–337, 2001.[1] 4, 5, 52

[2] Baeza-Yates R. A. and Gonnet G. H. All-against-all sequence matching., 1990. Report Department of Conputer Science, Univ. de Chile. 6, 28, 62

[3] Baeza-Yates R. A. and Gonnet G. H. All-against-all sequence matching. *Proceedings SPIRE/CRIWG*, pages 16–23, 1999. 5, 6, 27, 28

[4] Brunetti C.R., Amano H., Ueda Y., Qin J., Miyamura T., Suzuki T., Li X., Barrett J.W., and McFadden G. Complete genomic sequence and comparative analysis of the tumorigenic poxvirus yaba monkey tumor virus. *Journal of Virology*, **77**:13335–13347, 2003. 3

[5] Doolittle R.F., Hunkapiller M., Hood L.E., Devare S., Robbins K., Aaronson S., and Antoniades H. Simian sarcoma virus onc gene v-sis, is derived from the gene encoding a platelet-derived growth factor. *Science*, **221**:275–277, 1983. 3

[6] Efraty N. and Landau G. Sparse normalized local alignment. *Algorithmica*, **43**:179–194, 2005. 4, 5

[1] The numbers at the end of a reference indicate the page numbers of the thesis, where this reference was cited.

[7] Gusfield D. *The Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997. 5, 6, 9, 11, 28, 38

[8] Hubbard T.G̃. P., Lesk A. M., and Tramontano A. Gathering them into the fold. *Natural Structural Biology*, **4**:313, 1996. 36

[9] Karp L. M. *Reducibility among combinatorial problems.* In: Complexity of Computer ComputationsSeries. The IBM Research Symposia Series, Plenum Press, New York, 1972. 36

[10] Navarro G., Baeza-Yates R.A., Sutinen E., and Tarhio J. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, **24:4**:19–27, 2001. 1

[11] Novichkov P.S., Gelfand M.S., and Mironov A.A. Prediction of the exon-intron structure by comparison sequences. *Molecular Biology*, **34**:200–206, 2000. 3

[12] *The Oxford English Dictionary* Oxford University Press, 2005. 2

[13] Pevzner P. and Sze S. H. Combinatorial approaches to finding subtle signals in dna sequences. *Proceedings ISMB*, pages 269–278, 2000. 4

[14] Smith T.F. and Waterman M.S. Identification of common molecular subsequences. *Journal of Molecular Biology*, **147**:195–197, 1981. 4, 51

[15] Twain M. *Mark Twain's Speeches.* Kessinger Publishing, 2004. 2

[16] Ukkonen E. Algorithms for approximate string matching. *Information and Control*, **64**:100–118, 1985. 28, 29, 54, 62

[17] Ukkonen E. Approximate string matching over suffix trees. *Lecture Notes in Computer Science*, **684**:228–242, 1993. 6

[18] Vilo J. *Pattern Discovery from Biosequences, PhD Thesis.* Series of Publications A, Report A-2002-3, University of Helsinki, Finland, 2002. 28

[19] Virus orthologous clusters database at *http://athena.bioc.uvic.ca.* Viral Bioinformatics Resource Center, University of Victoria, Canada. 30

[20] WANG L. AND JIANG T. On the complexity of multiple sequence alignment. *Journal of Compputational Biology*, **1**:337–348, 1994. 36

[21] WATSON J.D. *DNA: the Secret of Life.* Alfred A. Knopf, 2004. 2

Appendix A

Comments about the Smith-Waterman Algorithm

A.1 Algorithm Description

Problem 4 Local Alignment *Given two strings s and t, local alignment problem is defined as the problem of finding the substring s' of s and t' of t, whose similarity (optimal global alignment) is maximal (over all such pairs of substrings).*

The Smith-Waterman algorithm (14) solves this problem by dynamic programming as following.

Notation 1 *Let $\sigma(a, b)$ be the score (weight) of the alignment of character a with character b (including spaces).*

For the simplicity of the demonstration, we define the scores as following:

$$\sigma(a,b) = \begin{cases} 2 & \text{if a and b match} \\ -1 & \text{if a and b do not match or one of them is a space} \end{cases}$$

We use $V(i, j)$ to denote the value of the optimal local suffix alignment for a given pair i, j of indices.

The algorithm proceeds as following:

Base conditions: **for each** i, j: $V(i, 0) = 0$, $V(0, j) = 0$.

Recurrence relation:

$$V(i,j) = MAX \begin{cases} 0 \\ V(i-1,j-1) + \sigma(s[i],t[j]) \\ V(i-1,j) + \sigma(s[i],-) \\ V(i,j-1) + \sigma(-,t[j]) \end{cases}$$

Compute i^* and j^*: $V(i^*,j^*) = MAX_{1 \leq i \leq N, 1 \leq j \leq M} V(i,j)$.

The algorithm correctly outputs the pair of positions (i^*,j^*), where the largest total score in the dynamic programming table was registered.

A.2 Deficiencies of the Smith-Waterman Algorithm

In real life, there is a need to examine all sub-optimal local alignments with high score. For this problem the Smith-Waterman algorithm does not perform well due to *mosaic* and the *shadow* effects (1). The *mosaic effect* leads to an output, where the poorly preserved regions end up with a score higher than more similar regions. As an illustration of the mosaic effect see Figure A.1. Here the Smith-Waterman algorithm has found the pair of local regions with the highest score: *axcgxtaxct* and *acygtyacat*, which have the lengths of 10, and with edit distance between them equal to 6. The algorithm has scored this pair as 8. At the same time the pair *axxgxxacxt* and *acygtyacat* with the same lengths of 10, but with the edit distance 5 was scored only as 5, which made it indistinguishable from the poorly conserved regions with the same score.

The second weakness of the Smith Waterman algorithm is known as *shadow effect*. This term describes the tendency of the algorithm to lengthen long alignments with a high score, rather than shorter alignments with a lower score and a higher degree of similarity. The example is represented in figure A.2. Here the best local alignment is stretched though the entire strings, and this shadows more similar regions of the lesser length, such as *cgtg* and *cgagtg*.

s \ t		a	x	c	g	x	t	a	x	c	t	a	x	x	g	x	x	a	c	x	t
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	**2**	**1**	0	0	0	0	2	1	0	0	2	1	0	0	0	0	2	1	0	0
c	0	1	1	**3**	2	1	0	1	1	3	2	1	1	0	0	0	0	1	4	3	2
y	0	0	0	**2**	2	1	0	0	0	2	2	1	0	0	0	0	0	0	3	3	2
g	0	0	0	1	**4**	**3**	2	1	0	1	1	1	0	0	2	1	0	0	2	2	2
t	0	0	0	0	3	3	**5**	4	3	2	3	2	1	0	1	1	0	0	1	1	4
y	0	0	0	0	2	2	**4**	4	3	2	2	2	1	0	0	0	0	0	0	0	3
a	0	2	1	0	1	1	3	**6**	**5**	4	3	4	3	2	1	0	0	**2**	1	0	2
c	0	1	1	3	2	1	2	5	5	**7**	6	5	4	3	2	1	0	1	**4**	3	2
a	0	2	1	2	2	1	1	4	4	**6**	6	8	7	6	5	4	3	3	3	**3**	2
t	0	1	1	1	1	1	3	3	3	5	**8**	7	7	6	5	4	3	2	2	2	**5**

a**x**c_g**x**t_axc_t	Alignment Score: 8 LCS:7
a_c**y**g_t**y**a_c**a**t	Edit Distance: 6 Length:10

a**xx**g**xx**ac**x**t	Alignment Score: 5 LCS:5
a**cy**g**ty**ac**a**t	Edit Distance: 5 Length:10

Figure A.1: Mosaic effect of the Smith Waterman algorithm.
Scoring: 2 for match, -1 for each edit operation.

t / s		a	c	a	a	t	t	c	g	t	g
	0	0	0	0	0	0	0	0	0	0	0
a	0	**2**	1	2	2	1	0	0	0	0	0
g	0	**1**	1	1	1	1	0	0	2	1	2
c	0	0	**3**	2	1	0	0	**2**	1	1	1
g	0	0	2	**2**	1	0	0	1	**4**	3	3
a	0	2	1	4	**4**	3	2	1	**3**	3	2
g	0	1	1	3	3	**3**	2	1	**2**	2	5
t	0	0	0	2	2	5	**5**	**4**	3	**4**	4
g	0	0	0	1	1	4	4	4	**6**	5	**6**
t	0	0	0	0	0	3	6	5	5	**8**	7
g	0	0	0	0	0	2	5	5	7	7	**10**
c	0	0	2	1	0	1	4	7	6	6	9

Figure A.2: Shadow effect of the Smith Waterman algorithm.
Scoring: 2 for match, -1 for each edit operation.

Appendix B

Optimized Baeza-Yates and Gonnet's Algorithm.

B.1 Error-bounded Edit Distance in Linear Time

Problem 5 Edit Distance. *Given two strings s and t, find the minimum number of edit operations (substitutions, insertions and deletions), needed to convert s into t.*

The problem can be solved by dynamic programming, in time quadratic in lengths of strings. The score σ for an edit distance calculation is defined as following:

$$\sigma(a,b) = \begin{cases} 0 & \text{if a and b match} \\ 1 & \text{if a and b do not match or one of them is a space} \end{cases}$$

Base conditions: **for each** i, j $V(i,0) = i$, $V(0,j) = j$.

Recurrence relation:

$$V(i,j) = MIN \begin{cases} V(i-1,j-1) + \sigma(s[i],t[j]) \\ V(i-1,j) + \sigma(s[i],-) \\ V(i,j-1) + \sigma(-,t[j]) \end{cases}$$

Ukkonen in (16) suggested an idea, that if we are interested to know if the edit distance between s and t is lower than some threshold K, we can calculate values of the dynamic programming table only in a $2K+1$ strip around the main

s \ t		a	c	a	a	t	t	c	g	t	g
	0	1	2	3	4						
a	1	0	1	2	3	∞					
g	2	1	2	2	3	4	∞				
c	3	2	1	2	3	4	5	∞			
a	4	3	2	1	2	3	4	5	∞		
a		∞	3	2	1	2	3	4	5	∞	
g			∞	3	2	2	3	4	4	5	∞
t				∞	3	2	2	3	4	4	5
c					∞	3	3	2	3	4	5
g						∞	4	3	2	3	4
t							∞	4	3	2	3
g								∞	4	3	2

Figure B.1: Error-bounded calculation of an edit distance between two strings.

diagonal of DP table. An example of an error-bounded edit distance calculation is shown in Figure B.1

B.2 Optimized Algorithm by Baeza-Yates and Gonnet

We used this idea when implementing algorithm $BYG + U$ for (threshold) all-against-all approximate substring matching over suffix tree. The pseudocode of our variant of the $BYG + U$ algorithm is given in Figures B.2 and B.3.

An example of processing by algorithm $BYG + U$, based on suffix trees in Figure B.4, is shown in Figure B.5.

BY_U(Suffix tree $\mathcal{S}$ for s,Suffix tree $\mathcal{T}$ for t,s,t,S,K)
  $root_s$= root of $\mathcal{S}$
  $root_t$= root of $\mathcal{T}$
  $root_children_s$= children of $root_s$
  $root_children_t$= children of $root_t$
  for i **from** 1 **to size of** $root_children_s$
    $suffix_s$= $root_children_s[i]$
    for j **from** 1 **to size of** $root_children_t$
      $suffix_t$=$root_children_t[j]$
      initialize **int array** *table* of size MAX_LENGTH
      fill first row and first column of *table* - base condition
      for each cell $table[m, n]$ in a $2K + 1$ strip around the main diagonal
        do recurrence relations for a global alignment
      if the calculated cell value $\leq K$ **and**
          $m - suffix_s.start \geq S$ **and** $n - suffix_t.start \geq S$
        add substring pair $s[suffix_s.start, m]$, $t[suffix_t.start, n]$ to solution set
      if not all calculated cells in last row and column $> K$
        call Continue_compare($suffix_s$,$suffix_t$,$table$)

Figure B.2: Algorithm *BYG_U*.

continue_compare(Suffix tree node $parent_s$, Suffix tree node $parent_t$, *table*)
  $children_s$= children of $parent_s$
  $children_t$= children of $parent_t$
  for i **from** 1 **to size of** $children_s$
   $suffix_s$=$children_s[i]$
   for j **from** 1 **to size of** $children_t$
    $suffix_t$=$children_t[j]$
    fill missing initial values in the first row and first column of *table*
    for each cell $table[m, n]$ in a $2K + 1$ strip around the main diagonal
     do recurrence relations for a global alignment
    if the calculated cell value $\leq K$ **and**
      $m - suffix_s.start \geq S$ **and** $n - suffix_t.start \geq S$
     add substring pair $s[suffix_s.start, m]$, $t[suffix_t.start, n]$ to solution set
    if not all calculated cells in last row and column $> K$
     call Continue_compare($suffix_s$,$suffix_t$,*table*)

Figure B.3: Algorithm *continue_compare*.

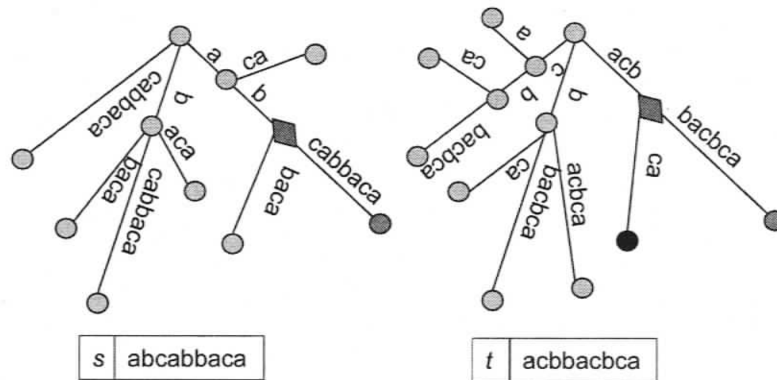


Figure B.4: Suffix trees for strings *abcabbaca* and *acbbacbca*.

	s	a	b ◆	c	a	b	b	a	c	a ●
t	0	1	2	3	4					
a	1	0	1	2	3	4				
c	2	1	1	1	2	3	4			
b	3	2	1	2	2	2	3	4		
b ◆	4	3	2	2	3	2	2	3	4	
a		4	3	3	2	3	3	2	3	4
c			4	3	3	3	4	3	2	3
b				4	4	3	3	4	3	3
c					5	4	4	4	4	4
a ●						5	5	4	5	4

Figure B.5: An example of algorithm $BYG + U$. After we have calculated an error-bounded edit distance between parent nodes (grey diamonds), we in turn compare a child node of the first parent with the child node of the second one (grey ovals). Note, that the values in the light-grey corner cells are reused for all child nodes, and the values in a dark-grey row are reused when comparing the first child node with all child nodes of the second tree (e.g. black oval).

Nomenclature

BIOLOGY

DNA (Deoxyribonucleic acid) - the molecule that encodes genetic information.

RNA (Ribonucleic acid) - responsible for translating the genetic code of DNA into $proteins$.

Protein (Polypeptide) - a molecule composed of a long chain of amino acids. Proteins make up much of the body's tissue in addition to being the main part of enzymes, hormones, and immunologic substances.

Gene - a sequence of DNA that represents a fundamental unit of heredity.

bp (Base Pairs) - formed when complementary nucleotides pair by hydrogen bonding. In DNA, the A nucleotide bonds with T, and G bonds with C. Base pairs form the "rungs" of the DNA ladder and the number of base pairs in a strand can be used to describe the length of DNA.

Mutation - a heritable change in the nucleotide sequence of DNA.

Exon - a segment of eukaryotic gene that is transcribed as part of the primary transcript and is retained, after processing, with other exons to form a functional $mRNA$ molecule.

Intron - a segment which is cut off from the message before it is translated into a protein.

Virus - an infective agent with a specific structure and able to cause its own multiplication after infection of specific cell.

BIOINFORMATICS

Sequence - the order of nucleotides in a DNA or RNA molecule, or the order of amino acids in a *protein* molecule.

FASTA format - a text-based format for representing both nucleic and protein sequences, in which base pairs or proteins are represented using a single-letter code. The format also allows for sequence names to precede the sequences (preceded by $>$) and for comments (preceded by a semicolon).

Alignment of two sequences s and t is obtained by first inserting chosen spaces, either into, at the ends of, or before s and t, and then placing the two resulting sequences one above the other so that every character or space in either sequence opposite a unique character or a unique space in the other sequence.

Scoring Matrix - a matrix containing pairwise scores for each pair of characters (including spaces) in the alphabet Σ.

Similarity . Given a pairwise scoring matrix, the similarity of two sequences s and t is defined as the maximum possible value of the alignment between s and t.

Global Alignment - an alignment with the maximum possible similarity value between the entire strings s and t.

Local Alignment - an alignment between substrings of s and t with the maximum similarity value.

COMPUTER SCIENCE

Data Mining - the process of analyzing data to identify patterns or relationships.

String - a sequence of elements of the same nature, such as characters, bits etc.

Substring - a contiguous sequence of characters taken from a string.

Pattern - a recognizable shape or arrangement of things, for example common pattern for two strings is a substring which can be distinguished in both strings from the rest of their substrings.

Match - the pair of identical substrings.

Approximate Match - the pair of similar substrings.

Natural Language - a human language whose rules have evolved from current usage, as opposed to an artificial language whose rules are prescribed prior to its construction and use.

Edit Distance - the minimum number of edit operations (insertions, deletions and substitutions) between two strings, needed to transform the one string into the other.

Edit Transcript - a string over alphabet $\Sigma = \{M(match), D(deletion), I(insertion), F(mismatch)\}$ that describes a transformation of one string into another.

Dynamic Programming - a class of solution methods for solving sequential decision problems with a compositional cost structure

Pattern-against-all - the computation of an edit distance between a short pattern p and each substring of a long string t.

(Threshold) All-against-all - the problem of finding every pair of substrings in two strings s and t, such that an edit distance between substrings in each pair is less than the predefined parameter K.

Graph - a construct that consists of many nodes connected with edges. The edges usually represent a relationship between the objects represented by the nodes.

Triangle Inequality - the sum of distances from object A to object B, and then to object C is not shorter than going directly from A to C.

NP-complete Problem - a problem that can be solved quickly by a nondeterministic computer, which only needs to verify its lucky guess; but the problem is exponentially difficult to solve on a deterministic machine, which has to try every possibility in search of an answer (Unless NP is not in P).

ACRONIMS

$APBT$ - the All Paths below Threshold Algorithm (new).

$BYG + U$ - the Bayeza-Yates-Gonnet algorithm (2) for the all-against-all problem, optimized with Ukkonen's algorithm for error-bounded edit distance (16).