King Abdullah University of Science and Technology

CS260 Design and Analysis of Algorithms

Midterm Project Report

# APPROXIMATE STRING MATCHING

BY


| | |
|---|---|
| Group Member | Gang Liao |
| Group Member | Fatima Zohra Smaili |
| Group Member | Wentao Hu |
| Group Member | Guangming Zang |

Department of Computer Science

Computer, Electrical and Mathematical Sciences & Engineering Division

Thuwal, Saudi Arabia


October 2014

# Acknowledgements

We are profoundly grateful to Prof. Mikhail Moshkov for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion.

We would like to express deepest appreciation towards our group members[1] whose invaluable guidance supported ourselves in completing this project. The KAUST report template was developed by Gang Liao, if you're interested in the LaTeX source code, you can download it from GitHub **[ Template ]**. In addition, the entire project can also be found in the GitHub **[ Project ]**.

At last we must express our sincere heartfelt gratitude to all the staff members of Computer, Electrical and Mathematical Sciences & Engineering Division who helped us directly or indirectly during this course of work.

| | |
|---|---|
| Group Member | Gang Liao |
| Group Member | Fatima Zohra Smaili |
| Group Member | Wentao Hu |
| Group Member | Guangming Zang |

[1] {liao.gang, fatimazohra.smaili, wentao.hu, guangming.zang}@kaust.edu.sa

# ABSTRACT

*Fatima Zohra Smaili*

String matching algorithms are an important class of algorithms in computer science used to solve some famous problems mainly DNA strings matching, text processing, spell checking, spam filtering. The idea behind them is to quickly find the first or all occurrences of a string in a text. In other words, given a text string T and a pattern P, we need to find a quick way to find whether there is any occurrence in P and where it appears in case it exists.

A slightly different but more interesting problem is approximate string matching problem, which is the problem we chose to work on for our project. For the approximate string matching problem we look for a substring that is similar to pattern P in text T. The word similar here refers to a string that needs a minimum number of operations (insertion, deletion and substitution) to be converted to P. This minimum number of operations is what we refer to as the edit distance.

In our report, we will try to give a brief overview on the two approximate string matching algorithms we chose to work with. The comparative performance evaluation between them is then carried out.
**Keywords:** Approximate, Bit Parallel, Dynamic Programming, String Matching

# Contents

# List of Figures

# Chapter 1

# Introduction

*Gang Liao*

## 1.1 Definition of String Matching Problem

In string matching problems, there are a text $T = t_1t_2...t_n$ and a pattern $P = p_1p_2...p_n$ where both $t_i$'s and $t_j$'s are characters. Throughout this book, we assume that $m \leq n$. We denote $t_it_{i+1}...t_j$ by $T(i,j)$. We shall consider two different kinds of string matching problems: exact string matching problem and approximate string matching problem.

### 1.1.1 Exact String Matching Problem

For the exact string matching problem, we are given a text $T = t_1t_2...t_n$ and a pattern $P = p_1p_2...p_n$. Our job is to find whether $P$ appears in $T$ and if it does, where it appears.

### 1.1.2 Example of the Exact String Matching

**Example 1.1.1.** If $T = aaccgtcaccggt$ and $P = acc$. We can find $P$ does appear in $T$ as shown below:

T = a<u>acc</u>gtc<u>acc</u>ggt

In the literatures, there are many algorithms developed to solve this exact string matching problem [1] [2] [3].

## 1.2 Approximate String Matching Problem

For the approximate string matching problem, we first define a distance function called edit distance which measures the similarity between two strings. Given a string $S_1$ and a string $S_2$, we can transform $S_1$ to $S_2$ by three operations: deletions, insertions and substitutions.

### 1.2.1 Insertion Operation

Let $S_1 = aactgt$ and $S_2 = actt$ . We may insert $a$ and $g$ at locations 2 and 5 respectively into $S_2$ to transform $S_2$ to $S_1$ as shown below:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $S_1$ | = | a | a | c | t | g | t |
| $S_2$ | = | a | – | c | t | – | t |
| | | | a | | | g | |

### 1.2.2 Deletion Operation

Let $S_1 = aacgt$ and $S_2 = aaccggt$. We may delete $c$ in location 4 and $g$ in location 5 from $S_2$. Then after these two deletion operations as illustrated below, we can transform $S_2$ to $S_1$ as shown below:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | = | a | a | c | – | – | g | t |
| $S_2$ | = | a | a | c | $\boxed{c}$ | $\boxed{g}$ | g | t |

### 1.2.3 Substitution Operation

Let $S_1 = aactgtt$ and $S_2 = abctatt$. Then we may transform $S_2$ to $S_1$ by the following substitution operations at locations 2 and 5. Note that in location 2, $b$ is substituted by $a$ and in location 5, $a$ is substituted by $g$.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $S_1$ | = | a | a | c | t | g | t | t |
| $S_2$ | = | a | $\boxed{b}$ | c | t | $\boxed{a}$ | t | t |
| | | | a | | | g | | |

Having defined these operations, we may now define edit distance as follows:

### 1.2.4 Edit Distance

The edit distance between two strings $S_1$ and $S_2$ is the minimum number of deletions, insertions and substitutions needed to transform $S_2$ to $S_1$. The edit distance between

$S_1$ and $S_2$ is denoted as ED($S_1$, $S_2$). The problem of finding the edit distance is an optimization problem and can be solved by the dynamic programming approach.

We are given two strings: $A = a_1a_2...a_i$ and $B = b_1b_2...b_j$. Our job is to find ED($A,B$). Let us first define a new term, denoted as $ed(i,j) = ED(A(1,i),B(1,j))$. Then we have the following statement:

If $a_i = b_j$,

$$ed(i,j) = ed(i-1,j-1)$$

If $a_i \neq b_j$,

$$ed(i,j) = min\begin{cases} ed(i-1,j)+1 & insertion \\ ed(i,j-1)+1 & deletion \\ ed(i-1,j-1)+1 & substitution \end{cases}$$

Edit distance can be viewed as a measurement of the similarity between two strings. If the edit distance is small, it means that these two strings are similar to each other and quite different if otherwise. If edit distance is zero, these two strings are identical.

## 1.3 Two Kinds of Approximate String Matching

**Approximate String Matching Problem 1:** Given a text $T$, a pattern $P$ and an error $k$, find every location $i$ in $T$ such that there is a substring $S$ of $T$ which ends at location $i$ such that ED($S,P$) $\leq k$.
**Approximate String Matching Problem 2:** Given a text $T$, a pattern $P$ and an error $k$, find all substrings $S$'s in $T$ such that ED($S,P$) $\leq k$.

## 1.4 Summary

Approximate string matching is one of the main problems in classical algorithms, with applications to text searching, computational biology, pattern recognition, etc. Many algorithms have been presented that improve approximate string matching, for instance [4, 5, 6, 7, 8, 9, 11]. We decide to implement two of them and compare them via the time and space complexity.

# Chapter 2

# Bit-Parallel Approximate String Matching

*Gang Liao, Wentao Hu*

## 2.1 Approximate String Matching

Using bit-parallelism [9][10] has resulted in fast and practical algorithms for approximate string matching under Levenshtein edit distance, which permits a single edit operation to insert, delete or substitute a character.

### 2.1.1 Definition

Our approximate string matching problem is defined as follows: We are given a text $T_{1,n}$, a pattern $P_{1,m}$ and an error bound $k$. We want to find all locations $T_i$ when $1 \leq i \leq n$ such that there exists a suffix $A$ of $T_{1,i}$ and $\mathrm{ED}(A,P) \leq k$.

**Example 2.1.1.** $T$ = *deaabeg*, $P$ = *aabac* and $k = 2$.

For $i = 5$, if $T_{1,5}$ = *deaab*, we note that there exists a suffix $A$ = *aab* of $T_{1,5}$ such that $ED(A,P) = ED(aab,aabac) = 2$.

### 2.1.2 Fundamental Observation

Our approach is based upon the two following observation:

For case 1, as shown in Fig. 2.1, let $S$ be a substring of $T$. If there exists a suffix $S_2$ of $S$ and a suffix $P_2$ of $P$ such that ED($S_2$, $P_2$) = 0, and ED($S_1$, $P_1$) $\leq k$, we have ED($S$, $P$) $\leq k$.

**Example 2.1.2.** $T$ = *addcd*, $P$ = *abcd* and $k = 2$.

We may decompose $T$ and $P$ as follows: When $T$= *add* + *cd*, $P$ = *ab* + *cd*, then ED(*add*, *ab*) = 2 and ED(*cd*,*cd*) = 0, thus ED($T$,$P$) = 2.

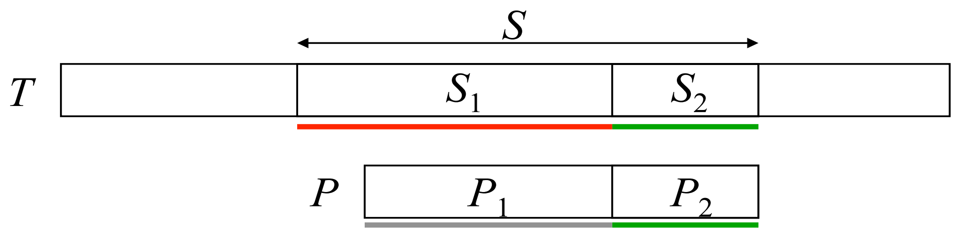


**Figure 2.1:** Obsevation of case 1.

For case 2, as shown in Fig. 2.2, let $S$ be a substring of $T$. If there exists a suffix $S_2$ of $S$ and a suffix $P_2$ of $P$ such that ED($S_2$, $P_2$) = 1, and ED($S_1$, $P_1$) $\leq k-1$, we have ED($S$, $P$) $\leq k$.

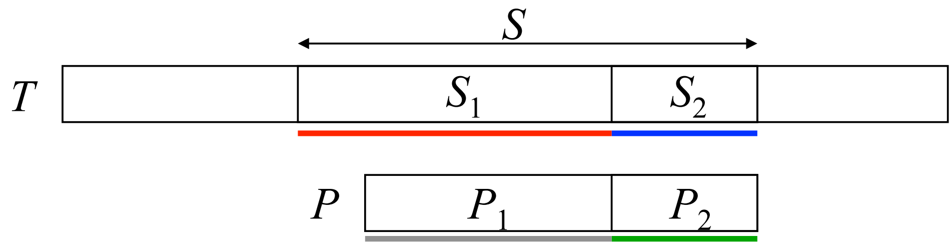


**Figure 2.2:** Obsevation of case 2.

**Example 2.1.3.** $T = addb$, $P = dc$ and $k = 3$.

We may decompose $T$ and $P$ as follows: When $T = add + b$, $P = d + c$, then ED($b$, $c$) = 1 and ED($add$, $d$) = 2, thus ED($T$,$P$) = 3.

## 2.2 $R^0$ Table

To solve our approximate string matching problem, we use a table, called $R^k[n,m]$.

$$R^k(i,j) = \begin{cases} 0 & Otherwise \\ 1 & if\ there\ exists\ a\ suffix\ A\ of\ T_{1,i}\ such\ that\ ED(A,P_{1,j}) \leq k \end{cases}$$

Where $1 \leq i \leq n$ and $1 \leq j \leq m$.

It will be shown later that $R^k$ is obtained from $R^{k-1}$. Thus, it is important for us to know how to find $R^0$. $R^0$ will be obtained by Shift-And algorithm later under bit-parallel. But now, we will explain it in dynamic programming.

## 2.3 Dynamic Programming

$R^0(i,j)$ can be found by dynamic programming.

$$\begin{cases} R^0(0,j) = 0 & for\ all\ j \\ R^0(i,0) = 1 & for\ all\ i \end{cases}$$

$$R^0(i,j) = \begin{cases} 0 & Otherwise \\ 1 & if\ R^0(i-1,j-1) = 1\ and\ t_i = p_j \end{cases}$$

From the above, we can see that the $i^{th}$ column of $R^0$ can be obtained from the $(i-1)^{th}$ column of $R^0$, $t_i$ and $p_j$.

## 2.4 Shift-And Algorithm

### 2.4.1 Bit manipulation

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization.

For most other tasks, modern programming languages allow the programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.

Bit manipulation, in some cases, can obviate or reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are processed in parallel, but the code can become rather more difficult to write and maintain.

#### Shift Operator

The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left:

$$[variable] \ll [number\ of\ places]$$

It shouldn't surprise you that there's a corresponding right-shift operator: $\gg$.

Generally, due to the implementation of hardware, using the left and right shift operators will result in significantly faster code than calculating and then multiplying by a power of two.

**Bitwise AND and OR**

The bitwise AND operator is a single ampersand: &. In essence, a binary AND simply takes the logical AND of the bits in each position of a number in binary form.

Bitwise OR works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1. (If both bits are a 1, the result will also have a 1 in that position.) The symbol is a pipe: $|$ or $\vee$.

### 2.4.2 Predefined Operations

We now define a right shift operation $\gg i$ on a vector A = $(a_1, a_2, ..., a_m)$ as follows:

$$A \gg i = (\overbrace{0, ..., 0}^{i}, a_1, a_2, ..., a_{m-i})$$

That is, we delete the last $i$ elements and add $i$ 0's at the front. For instance, A = (1,0,0,1,1,1), then A $\gg$ 1 = (0,1,0,0,1,1). We use & and $\vee$ to represent AND and OR operation. We further use $a^m b^n$ to denote (a,...,a,b,...,b) as below.

$$a^m b^n = (\underbrace{a, ..., a}_{m}, \underbrace{b, ..., b}_{n})$$

For example, $10^3$ = (1,0,0,0).

Given an alphabet $a$ and a pattern $P$, we need to know where $a$ appears in $P$. This information is contained in a vector $\sum(a)$ which is defined as follows:

$$\sum(a)[i] = \begin{cases} 0 & if\ P_i \neq a \\ 1 & if\ P_i = a \end{cases}$$

**Example 2.4.1.** If a pattern $P$ = $aabac$, then

$$\begin{cases} \sum(a) = (1,1,0,1,0) \\ \sum(b) = (0,0,1,0,0) \\ \sum(c) = (0,0,0,0,1) \end{cases}$$

### 2.4.3 Definition of Shift-And Algorithm

In general, we will update the $i^{th}$ column of $R^0[n,m]$, which is $R_i^0$, by using the formula $R_i^0 = ((R_{i-1}^0 \gg 1) \vee 10^{m-1}) \& \sum(t_i)$ for each new character of $i$. If $R^0(i,m) = 1$, report a match at position $i$.

**Example 2.4.2.** Given a text $T$ = *aabaacaabacab*, a pattern $P$ = *aabac* and an error bound $k = 0$.

Then we can get the $R^0[13,5]$ as follows:

| | a | a | b | a | a | c | a | a | b | a | c | a | b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| a | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| a | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| c | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

From the table above, $R_9^0 = (0,0,1,0,0)$ and $\sum(t_{10}) = (1,1,0,1,0)$, then we have

$$\begin{aligned} R_{10}^0 &= ((R_9^0 \gg 1) \vee 10^4) \& \sum(t_{10}) \\ &= ((0,0,0,1,0) \vee (1,0,0,0,0)) \& (1,1,0,1,0) \\ &= (1,0,0,1,0) \end{aligned}$$

Similarly, we can conclude that $R^0(10,1) = 1$.

## 2.5 $R^k$ Table

The approximate string matching problem is exact string matching problem with errors. We can use like Shift-And to achieve the $R^0[n,m]$ table. Here, we will introduce three operation which are insertion, deletion and substitution in approximate string matching.

$$R^k(i,j) = \begin{cases} 0 & Otherwise \\ 1 & if\ there\ exists\ a\ suffix\ A\ of\ T_{1,i}\ such\ that\ ED(A,P_{1,j}) \le k \end{cases}$$

Let $R_I^k(i,j)$, $R_D^k(i,j)$ and $R_S^k(i,j)$ denote the $R^k(i,j)$ related to insertion, deletion and substitution respectively.

$R_I^k(i,j) = 1$, $R_D^k(i,j) = 1$ and $R_S^k(i,j) = 1$ indicate that we can perform an insertion, deletion and substitution respectively without violating the error bound which is $k$.

Thus, there is the general formula for $R^k$ Table as follows:

$$\begin{cases} R^k(i,j) = 1 & if\ t_i = p_j\ and\ R^k(i-1,j-1) = 1 \\ R^k(i,j) = R^k_I(i,j) \vee R^k_D(i,j) \vee R^k_S(i,j) & Otherwise \end{cases}$$

**Example 2.5.1.** Given a text $T$ = *aabaacaabacab*, a pattern $P$ = *aabac* and an error bound $k = 1$.

Consider $i$ = 6 and $j$ = 5, $S = T_{1,6} = aabaac$ and $P_{1,5} = aabac$. Since $t_6 = p_5 = c$ and $R^1(5,4) = 1$, there exists a suffix $A$ of $S$ such that ED($A,P$) $\leq 1$. Finally, $R^1(6,5) = 1$.

### 2.5.1 Insertion

We can define the following formula for insertion operations.

$$\begin{cases} R^k_I(i,j) = 0 & Otherwise \\ R^k_I(i,j) = 1 & if\ t_i \neq p_j\ and\ R^{k-1}(i-1,j) = 1 \end{cases}$$

**Example 2.5.2.** In Fig.2.3, given a substring of a text $T = aabacb$, a substring of a pattern $P = aabac$ and $k = 1$.

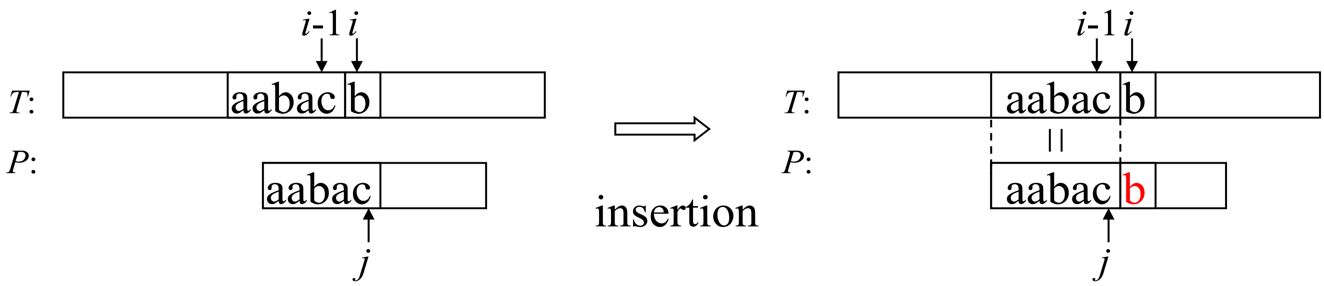


**Figure 2.3:** Insertion operations.

**Example 2.5.3.** Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.2.4, when $i = 6$ and $j = 4$, $t_6$ = $c \neq p_4 = a$, $R^0(5,4) = 0$, so $R^1_I(6,4) = 0$. when $i = 11$ and $j = 4$, $t_{11}$ = $c \neq p_4 = a$, $R^0(10,4) = 1$, so $R^1_I(11,4) = 1$.

| $R^0[13,5]$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | a | b | a | a | c | a | a | b | a | c | a | b |
| 1 | a | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | a | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | b | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | a | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

| $R_I[13,5]$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | a | b | a | a | c | a | a | b | a | c | a | b |
| 1 | a | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | a | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 3 | b | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 4 | a | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

**Figure 2.4:** The example for insertion.

According to the definition of insertion, the insertion formula can be expressed based on bit-parallel as below:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee R_{i-1}^{k-1}$$

### 2.5.2 Deletion

We can define the following formula for deletion operations.

$$\begin{cases} R_D^k(i,j) = 0 & Otherwise \\ R_D^k(i,j) = 1 & if\ t_i \neq p_j\ and\ R^{k-1}(i,j-1) = 1 \end{cases}$$

**Example 2.5.4.** In Fig.2.5, given a substring of a text $T = aabac$, a substring of a pattern $P = aabacb$ and $k = 1$.

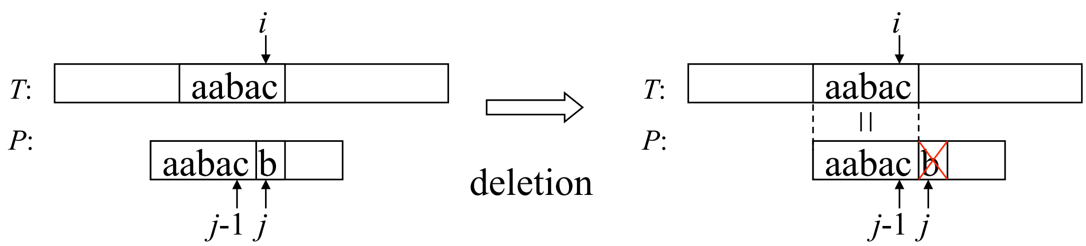


**Figure 2.5:** Deletion operations.

**Example 2.5.5.** Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.2.6, when $i = 6$ and $j = 4$, $t_6 = c \neq p_4 = a$, $R^0(5,4) = 0$, so $R_I^1(6,4) = 0$. when $i = 11$ and $j = 4$, $t_{11} = c \neq p_4 = a$, $R^0(10,4) = 1$, so $R_I^1(11,4) = 1$.

| $R^0[13,5]$ | | 1 a | 2 a | 3 b | 4 a | 5 a | 6 c | 7 a | 8 a | 9 b | 10 a | 11 c | 12 a | 13 b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | a | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | a | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | b | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | a | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

| $R_D^1[13,5]$ | | 1 a | 2 a | 3 b | 4 a | 5 a | 6 c | 7 a | 8 a | 9 b | 10 a | 11 c | 12 a | 13 b |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | a | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | a | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 3 | b | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 4 | a | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

**Figure 2.6:** The example for deletion.

According to the definition of deletion, the deletion formula can be expressed based on bit-parallel as below:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee ((R_i^{k-1} \gg 1) \vee 10^{m-1})$$

### 2.5.3 Substitution

We can define the following formula for substitution operations.

$$\begin{cases} R_S^k(i,j) = 0 & Otherwise \\ R_S^k(i,j) = 1 & if\ t_i \neq p_j\ and\ R^{k-1}(i-1,j-1) = 1 \end{cases}$$

**Example 2.5.6.** In Fig.2.7, given a substring of a text $T = aabacb$, a substring of a pattern $P = aabaca$ and $k = 1$.

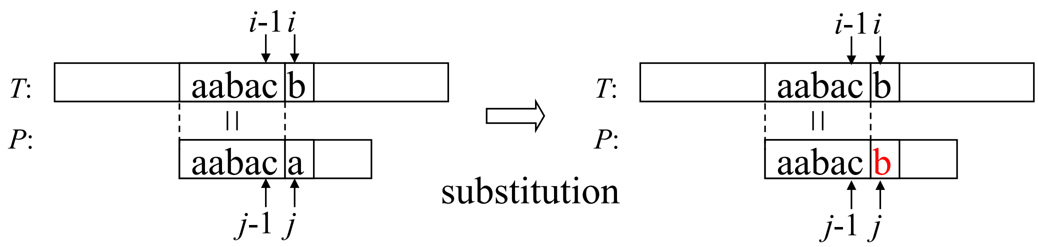


**Figure 2.7:** Substitution operations.

**Example 2.5.7.** Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.2.8, when $i = 6$ and $j = 4$, $t_6 = c \neq p_4 = a$, $R^0(5,4) = 0$, so $R_I^1(6,4) = 0$. when $i = 11$ and $j = 4$, $t_{11} = c \neq p_4 = a$, $R^0(10,4) = 1$, so $R_I^1(11,4) = 1$.

| $R^0[13,5]$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | a | b | a | a | c | a | a | b | a | c | a | b |
| 1 | a | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 2 | a | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | b | 0 | 0 | 1 | 0 | **0** | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | a | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

| $R_S^1[13,5]$ | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | a | a | b | a | a | c | a | a | b | a | c | a | b |
| 1 | a | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | a | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | b | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 4 | a | 0 | 0 | 0 | 1 | 0 | **0** | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | c | 0 | 0 | 0 | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

**Figure 2.8:** The example for substitution.

According to the definition of substitution, the substitution formula can be expressed based on bit-parallel as below:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee ((R_{i-1}^{k-1} \gg 1) \vee 10^{m-1})$$

### 2.5.4 Formulaization

The algorithm which contains insertion, deletion and substitution allows k error. In the physical meaning, we just combine the formulas of insertion, deletion and substitution. So now, we combine the three formula using the OR operation as follows:

The insertion:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee R_{i-1}^{k-1}$$

The deletion:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee ((R_i^{k-1} \gg 1) \vee 10^{m-1})$$

The substitution:

$$R_i^k = (((R_{i-1}^k \gg 1) \vee 10^{m-1}) \& \sum(t_i)) \vee ((R_{i-1}^{k-1} \gg 1) \vee 10^{m-1})$$

Finally, for $R^k$ Table, the formula will be

$$\text{Initially, } R_0^k = 1^k 0^{m-k}$$

$$R_i^k = ((R_{i-1}^k \gg 1) \& \sum(t_i)) \vee (R_{i-1}^{k-1}) \vee (R_i^{k-1} \gg 1) \vee (R_{i-1}^{k-1} \gg 1) \vee 10^{m-1}$$

## 2.6 Time and Space Complexity

If $k$ denotes error value, $m$ is the length of pattern, $n$ is the length of text and $w$ denotes the size of the word of computer. Then the time complexity will be $O(k\lceil \frac{m}{w} \rceil n)$ and the space complexity will be $O(\lceil \frac{m}{w} \rceil n)$.

## 2.7 Summary

Approximate string matching is an important topic in computational molecular biology. In this chapter, we can see that the basic dynamic programming algorithm can be greatly improved by using bit parallelism.

# Chapter 3

# Very Fast and Simple Approximate String Matching

*Fatima Zohra Smaili, Guangming Zang*

The second algorithm considered is based on the paper ”very fast and simple approximate string matching” by Ricardo Baeza-Yates and Gonzalo Navarro published in the Information Processing Letters journal in 1999 [5].

Before getting into the details of this algorithm, lets first formulate the problem we are trying to solve in this project one more time. The approximate string matching problem we are trying to solve is defined as follows: given a pattern $P$ of length $m$ and a text string $T$ of length $n$ and a maximal number $k$ of errors, we need to find all text positions in $T$ that match the pattern $P$ with up to edit distance equal to $k$.

## 3.1 Lemma

This algorithm is based on a lemma presented by the same authors in a previous paper [12]. In our project, we will not go through the proof of this lemma as it is not our main concern and as the proof is available on the paper. The lemma says the following:

Let $T$ and $P$ be two strings. Let $P$ be divided into $j$ pieces $p_1, p_2, p_3, ..., p_j$. If the editing distance between $T$ and $P$ is $ED(T,P) \leq k$, then there exists at least one $p_i$ and a substring $S$ in $T$ such that $ED(S, p_i) \leq \lfloor \frac{k}{j} \rfloor$. If we choose $j = k+1$, then $\lfloor \frac{k}{j} \rfloor = 0$. In this case, if $ED(T,P) \leq k$ then at least one of the pieces $p_i$ should occur as it is in $T$.

We will use this simple lemma to detect if there exist an approximate matching of $P$ in the text $T$ simply by first dividing the text $T$ into a set of windows and dividing $P$ into $k$+1 pieces $p_i$ and systematically looking for each one of these pieces in the text $T$. If we cannot find any matching of any of these pieces in any of the windows of $T$, then we can stop and safely say that there is no approximate matching of $P$ in $T$ with a maximum number of errors less or equal to $k$. That is to say we have to possible cases:

(1) If in a some window of $T$ ,we can find an exact matching of at least one of the $p_i$ substrings, we look at a bigger substring of $P$ and use dynamic programming to determine whether there exists an approximate matching of $P$ allowing only $k$ or less errors.

(2) If in a window we cannot find any exact matching of any of the $p_i$ substrings of $P$, we ignore this window because there is no need to check any further for an approximate matching of $P$ in that window.

The clear advantage of this method is that it allows us to quickly reduce the number of windows that need to be checked which can be very helpful with long texts.

A legitimate question to ask here is how large should the window size be to make the algorithm work efficiently. This detail is very flexible and can be determined according to the preference of the designer, however we should keep in mind that the size should not exceed $m+2k$, as shown in Fig.3.1. To understand why, lets assume that a substring S of T matches perfectly the pattern $P$. Any extension of $S$ to the right or to the left with $k$ characters will result in obtaining an approximate matching to $P$ with edit distance less or equal to $k$.

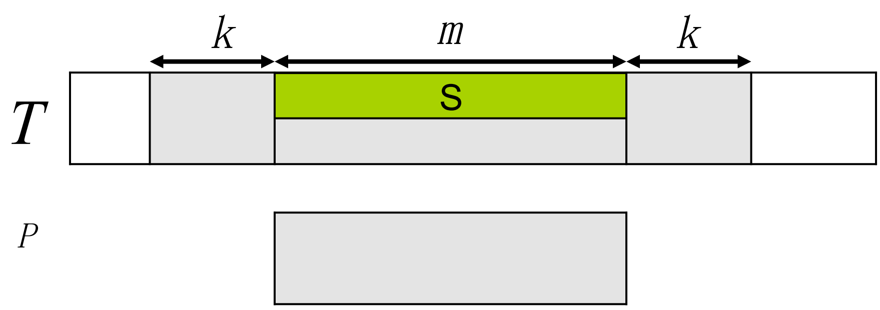


**Figure 3.1:** The size should not exceed $m+2k$.

## 3.2 Details of Algorithm

To recapitulate, in this algorithm we first divide the pattern into $k+1$ pieces, and we divide the text $T$ into a set of windows that we choose as long as we do not exceed the maximum allowed size as discussed above. After determining the occurrences of exact matching of small pieces, we start to determine the occurrences of larger piece of $P$ in $T$, for instance, as shown in Fig.3.2.

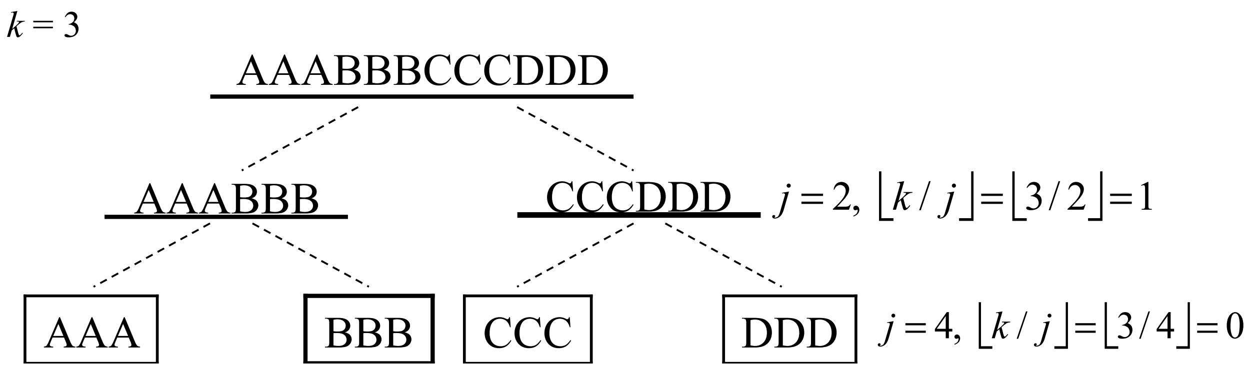


**Figure 3.2:** An example of this algorithm.

Now, inside each window we need to know how to systematically check for each piece and how to shift the window accordingly. To do so, for each piece of $P$, we will construct a table and fill it as follows:

Let $x$ be a character in the alphabet. We record the position of the last $x$, if it exists in piece of $P$, we record the position of $x$ from the right end. If $x$ does not exist in piece of $P$, we record it as $m+1$.

Suppose we have $P = ATCCTC$ with $k = 2$. We divide $P$ into three pieces: $p_1 = AT$, $p_2 = CC$ and $p_3 = TC$. To search for exact matching, we actually perform an exhaustive search.

First, in Fig.3.3, let us assume that we search for $AT$:

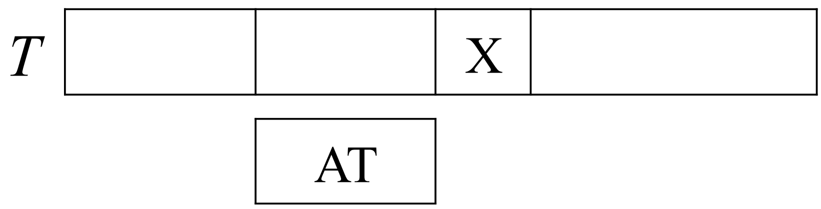


**Figure 3.3:** Searching for $AT$.

We have three possible cases:

(1) X = $A$. We move $AT$ 2 steps.

(2) X = $T$. We move $AT$ 1 steps.

(3) X $\neq A$ and X $\neq T$, we move $AT$ 3 steps.

Now, let us assume that we search for $CC$:

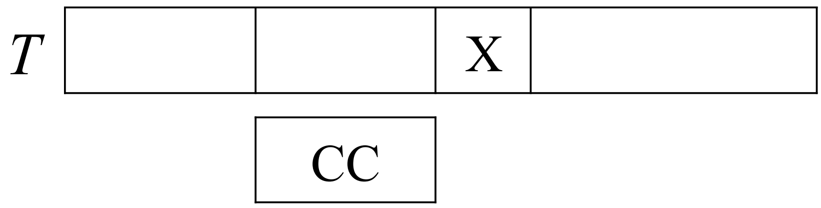


**Figure 3.4:** Searching for $CC$.

We have two possible cases:

(1) X = $C$. We move $CC$ 1 step.

(2) X $\neq C$. We move $CC$ 3 steps.

Finally, lets assume that we search for $TC$:

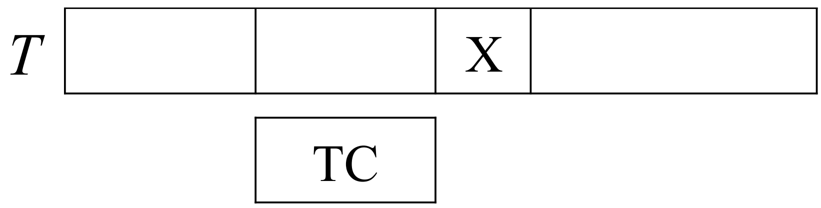


**Figure 3.5:** Searching for $TC$.

We have three possible cases:

(1) X = $T$. We move $TC$ 2 steps.

(2) X = $C$. We move $TC$ 1 step.

(3) X $\neq T$ and X $\neq C$, we move $TC$ 3 steps.

At last, we will end up with three tables for each one of the pieces we have:

$p_1 = \text{AT}$

| A | T | * |
|---|---|---|
| 2 | 1 | 3 |

$p_2 = \text{CC}$

| C | * |
|---|---|
| 1 | 3 |

$p_3 = \text{TC}$

| T | C | * |
|---|---|---|
| 2 | 1 | 3 |

In Fig.3.6, all three tables can now be combined into one shit table that we can use to know by how many steps we need to shift the window according to the current letter we are considering:

| A | T | C | * |
|---|---|---|---|
| 2 | 1 | 1 | 3 |

**Figure 3.6:** Shift Table.

## 3.3 Example

Lets know apply this shift table on a text $T = TCCAAGTTATAGCTC$.

**Example 3.3.1.** So what we have is the following: $T = TCCAAGTTATAGCTC$, $P = ATCCTC$, $k = 2$ and $p_1 = AT, p_2 = CC, p_3 = TC$.

(1) First Step:

$\boxed{\text{T C}}$ C A A G T T A T A G C T C

We open a window with length two to compare with $AT, CCandTC$. We can see that the window contains an exact matching with $p_3$. Then shift the window according to shift table value of next position.

(2) Second Step:

T $\boxed{\text{C C}}$ A A G T T A T A G C T C

Here again there is an exact matching with $p_2$. Then we shift the window by two positions.

(3) Third Step:

$$\text{T C C } \boxed{\text{A A}} \text{ G T T A T A G C T C}$$

Here there is no matching with any of $p_1, p_2$ and $p_3$. Therefore, we shift the window by three positions and so on and so forth.

Using these results, we will find out that $AT$ occurs in $T$ at position 9, $CC$ occurs at position 2 and $TC$ at positions 1 and 14. These results can be summarize in the following table:

| | |
|---|---|
| AT | 9 |
| CC | 2 |
| TC | 1, 14 |

## 3.4 Time Complexity

Theoretically the time complexity of this algorithm is $O(\frac{kn}{m})$ where our error level is $a = \frac{k}{m}$.

# Chapter 4

# System Testing

To be Continued.

## 4.1 Test Cases and Test Results

# Chapter 5

# Project Planning

*All group members approved this planning.*

## 5.1 September

(1) ~~Review and research on the topic and its applications~~.

(2) ~~Review specific papers on the two algorithms chosen~~.

(3) ~~Task dispatching among team members~~.

## 5.2 October

(1) ~~Implementation of the algorithms~~.

(2) ~~Preparation of Midterm Presentation~~.

**Gang and Fatima did the midterm presentation.**

(3) ~~Preparation of Midterm Report~~.

## 5.3 November

(1) Evaluation of the implemented algorithms on different case studies.

(2) Analysis of time and space complexity of the algorithms.

(3) Synthesis and comparison of the algorithms

(4) Work on improving the existing algorithms as future work.

(5) Preparation of Final Presentation and Report.

**Wen-tao and Guang-ming expect to do the final presentation.**

# Chapter 6

# Implementation

*Gang Liao*

The 1st algorithm.

```
//  main.cpp
//  bit-parallel approximate string
//
//  Created by gangliao on 10/9/14.
//  Copyright (c) 2014 gangliao. All rights reserved.

#include <iostream>
#include <string>

using namespace std;

#define INDEX(c)    ((c) - 'a')
#define BIT 1 << 31

//according to the pattern p, initializing
//all the positions into the set container
void initSet(string p /*pattern*/, unsigned int* set)
{
    for (int i = 0; i < 31; i++) {
        set[INDEX(p[i])] = set[INDEX(p[i])] | (1 << (31-i));
    }
}

int main(int argc, const char * argv[])
{

    //pattern length = 32
    string pattern = "gttggcagcagtcgatcaaattgccgatccga";

    //text length = 256
    string text =  "gttggcagcagtcgatcaaattgccgatccgagtt
    ggcagcagtcgatcaaattgccgatccaatgataaattcggttggcagctt
    agtcgatcaaaatgcccatcccacggttggcagcagtcgatcaaatcgacc
    accgatgcagatcggttggcagcagtcgatttgccgatccgagtgcagtcg
    atcaaattgccgatccgagttggcagcagtcgatcaaattgccgatccgaa
    gtctcaaattgccgatc";
```

```
    //asuume the numbers of error k is 10
    int k = 10; // allowed errors

    unsigned int set[26]; // all the postions of each character c in pattern p

    memset(set, 0, sizeof(int)*26); // set zeroes into this array

    initSet(pattern, set);

    int pre = 0, cur = 1;

    int R[2][257];

    R[pre][0] = 0; //if the length of the text == 0 and error k == 0

    // firstly , we need to get R0 table using shift-and algorithm under bit-
        parallel
    for(int i = 1 ; i <= 256; i++){
        R[pre][i] = ((R[pre][i - 1] >> 1) | BIT) & set[INDEX(text[i - 1])];
    }

    //R0 -> R1 -> R2 ... -> Rk
    for(int i = 1; i <= k; i++)
    {
        R[cur][0] = 0;
        for(int j = 0; j < i; j++) {
            R[cur][0] |= (1 << (31 - i)); //if the length of the text == 0 and
                error k == i
        }

        //O(kn*(m/w)) = O(kn)
        //m = 32 = 4 bytes = one integer  (n = 256, k = 10)
        for(int j = 1; j <= 256; j++){
            R[cur][j] = ((R[cur][j - 1] >> 1) & set[INDEX(text[j - 1])]) |
                        (R[pre][j - 1]) | /*insertion*/
                        (R[pre][j] >> 1) | /*deletion*/
                        (R[pre][j - 1] >> 1) /*substitution*/ | BIT;
        }

        cur = !cur;  //exchange the index, 1 to 0 or 0 to 1
        pre = !pre;  //exchange the index, 1 to 0 or 0 to 1

    }

    int sum = 0;
    for(int i = 1; i <= 256; i++){

         if(R[pre][i] & 1)
         {
             sum++;
             cout << "#" << sum << endl;
             for (int j = 0; j < i; j++) {
                 cout << text[j];
```

```
            }
            cout << endl;
        }
    }


    cout << "# approximate string with k error is " << sum << endl;
    return 0;
}
```

The 2nd Algorithm. (to be continued)

# Chapter 7

# Conclusion and Future Scope

## 7.1 Conclusion

WRITE HERE.

## 7.2 Future Scope

WRITE HERE.

- ITEM 1
- ITEM 2
- ITEM 3

# References

[1] Gusfield. Dan, *Algorithms on strings, trees and sequences: computer science and computational biology*, Cambridge University Press, 1997.

[2] Crochemore, Maxime and Rytter, Wojciech and Crochemore, Maxime, *Text algorithms*, World Scientific, 1994.

[3] Crochemore, Maxime and Hancart, Christophe and Lecroq, Thierry, *Algorithms on strings*, Cambridge University Press, 2007.

[4] E. Ukkonen, *Finding approximate patterns in strings*, Journal of algorithms, vol. 6, no. 1, pp. 132–137, 1985.

[5] G. Navarro and R. Baeza Yates, *Very fast and simple approximate string matching*, Information Processing Letters, vol. 72, no. 1, pp. 65–70, 1999.

[6] G. M. Landau and U. Vishkin, *Fast parallel and serial approximate string matching*, Journal of algorithms, vol. 10, no. 2, pp. 157–169, 1989.

[7] R. A. Baeza Yates and C. H. Perleberg, *Fast and practical approximate string matching*, in Combinatorial Pattern Matching. Springer, 1992, pp. 185–192.

[8] R. Baeza-Yates and G. Navarro, *Faster approximate string matching*, Algorithmica, vol. 23,no. 2, pp. 127-158, 1999.

[9] H. Hyyro, *Bit-parallel approximate string matching algorithms with transposition*, in String Processing and Information Retrieval. Springer, 2003, pp. 95-107.

[10] Wu, Sun, and Udi Manber. *Fast text searching: allowing errors*, Communications of the ACM 35.10 (1992): 83-91.

[11] S. Wu and U. Manber, *Fast text searching with errors*, University of Arizona, Department of Computer Science, 1991.

[12] Navarro, Gonzalo, and Ricardo Baeza-Yates. *A hybrid indexing method for approximate string matching.*, Journal of Discrete Algorithms 1, no. 1 (2000): 205-239.