King Abdullah University of Science and Technology

CS260 Design and Analysis of Algorithms

Final Project Report

APPROXIMATE STRING MATCHING

BY

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

Computer, Electrical and Mathematical Sciences & Engineering Division

Thuwal, Saudi Arabia

November 25, 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

Project Planning

All group members approved this planning.

1.1 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~~.

1.2 October

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

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

Gang and Fatima presented the midterm report.

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

1.3 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 presented the final report.

Chapter 2

Introduction

Gang Liao

2.1 Definition of String Matching Problem

In string matching problems, there are a text $T = t_1t_2...t_n$ and a pattern $P = p_1p_2...p_m$ where both t_i's and p_j's are characters. Throughout this report, 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.

2.1.1 Exact String Matching Problem

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

2.1.2 Example of the Exact String Matching

Example 2.1.1. 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].

2.2 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.

2.2.1 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	

2.2.2 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

2.2.3 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:

2.2.4 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: $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$. 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:

When $i = 0$, $ed(0,j) = j$. When $j = 0$, $ed(i,0) = i$.

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.

2.3 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) $\leq k$.

2.4 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 3

Bit-Parallel Approximate String Matching

Gang Liao, Wentao Hu

3.1 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.

3.1.1 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 3.1.1. $T = deaabeg$, $P = aabac$ 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 $ED(A, P) = ED(aab, aabac) = 2$.

3.1.2 Fundamental Observation

Our approach is based upon the two following observation:

For case 1, as shown in Fig. 3.1, 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 $\mathrm{ED}(S_2, P_2) = 0$, and $\mathrm{ED}(S_1, P_1) \leq k$, we have $\mathrm{ED}(S, P) \leq k$.

Example 3.1.2. $T = addcd$, $P = abcd$ and $k = 2$.

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

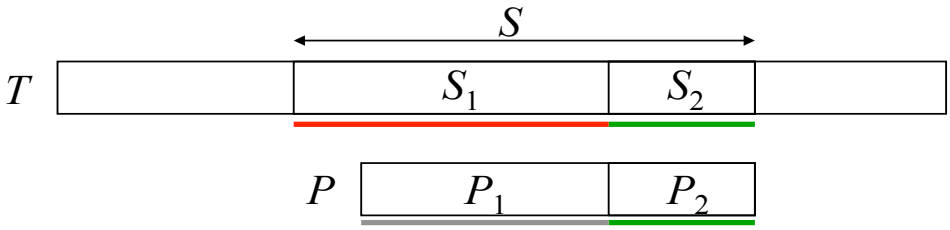

Figure 3.1: Obsevation of case 1.

For case 2, as shown in Fig. 3.2, 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) $\leq k$.

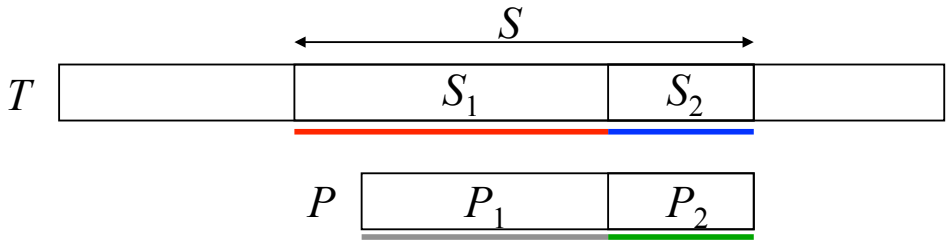

Figure 3.2: Obsevation of case 2.

Example 3.1.3. $T = addb$, $P = dc$ 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.

3.2 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.

3.3 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.

3.4 Shift-And Algorithm

3.4.1 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 $\vee$.

3.4.2 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 3.4.1. If a pattern P = *aabac*, then

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

3.4.3 Definition of Shift-And Algorithm

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

Example 3.4.2. 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, $R^0_9 = (0,0,1,0,0)$ and $\sum(t_{10}) = (1,1,0,1,0)$, then we have

$$\begin{aligned} R^0_{10} &= ((R^0_9 \gg 1) \vee 10^4) \& \sum(t_{10}) \\ &= ((0,0,0,1,0) \vee (1,0,0,0,0)) \& (1,1,0,1,0) \\ &= (1,0,0,1,0) \end{aligned}$$

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

3.5 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.

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

Let $R^k_I(i,j)$, $R^k_D(i,j)$ and $R^k_S(i,j)$ denote the $R^k(i,j)$ related to insertion, deletion and substitution respectively.

$R^k_I(i,j) = 1$, $R^k_D(i,j) = 1$ and $R^k_S(i,j) = 1$ 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 3.5.1. 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, $R^1(6,5) = 1$.

3.5.1 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 3.5.2. In Fig.3.3, given a substring of a text $T = aabacb$, a substring of a pattern $P = aabac$ and $k = 1$.

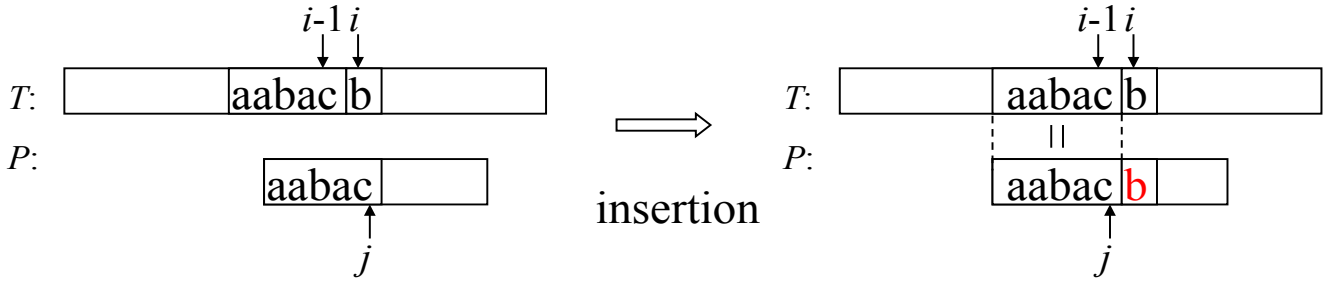

Figure 3.3: Insertion operations.

Example 3.5.3. Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.3.4, 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	1	1	0	0	1	
4	a	0	0	0	1	1	**0**	0	0	1	**1**	0	0	
5	c	0	0	0	0	0	1	0	0	0	1	1	0	

Figure 3.4: The example for insertion.

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

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

3.5.2 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 3.5.4. In Fig.3.5, given a substring of a text $T = aabac$, a substring of a pattern $P = aabacb$ and $k = 1$.

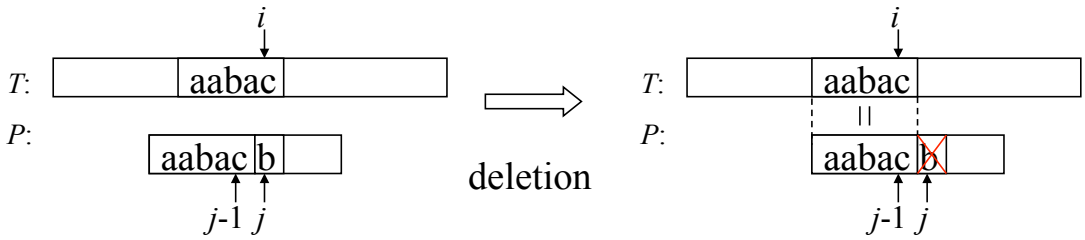

Figure 3.5: Deletion operations.

Example 3.5.5. Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.3.6, 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	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

$R^1_D[13,5]$

		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	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 3.6: The example for deletion.

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

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

3.5.3 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 3.5.6. In Fig.3.7, given a substring of a text $T = aabacb$, a substring of a pattern $P = aabaca$ and $k = 1$.

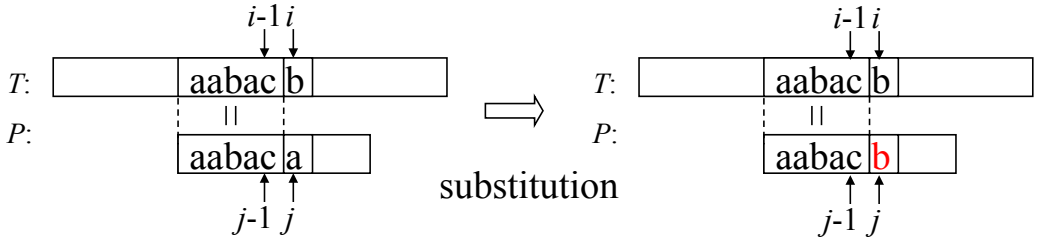

Figure 3.7: Substitution operations.

Example 3.5.7. Given a text $T = aabaacaabacab$, a pattern $P = aabac$ and $k = 1$.

As shown in Fig.3.8, 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

$R^1_S[13,5]$

		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 3.8: The example for substitution.

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

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

3.5.4 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

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

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

3.6 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.

In this algorithm, we need to maintain R^k table, it contains $\lceil \frac{m}{w} \rceil$ rows and n columns. The algorithm traverses the table column by column, so it takes $O(\lceil \frac{m}{w} \rceil n)$ in the specific table R^k (where $k = 0, 1, ..., k$). Since the error value is k, previous steps have to loop k times, then the time complexity will be $O(k\lceil \frac{m}{w} \rceil n)$. In the real implementation, we only need one table to implement this algorithm, thus the space complexity will be $O(\lceil \frac{m}{w} \rceil n)$.

3.7 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 4

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.

4.1 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.4.1. 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.

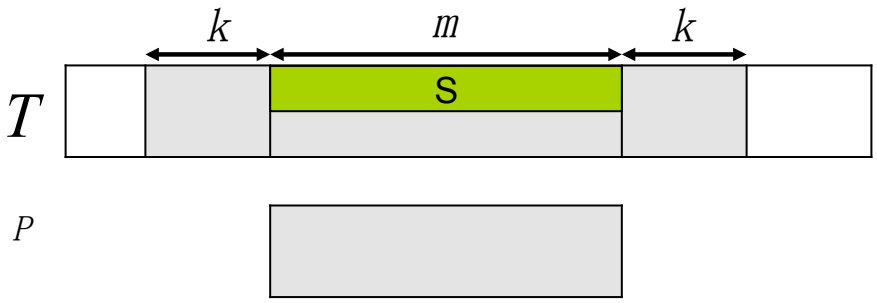

Figure 4.1: The size should not exceed $m+2k$.

4.2 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.4.2.

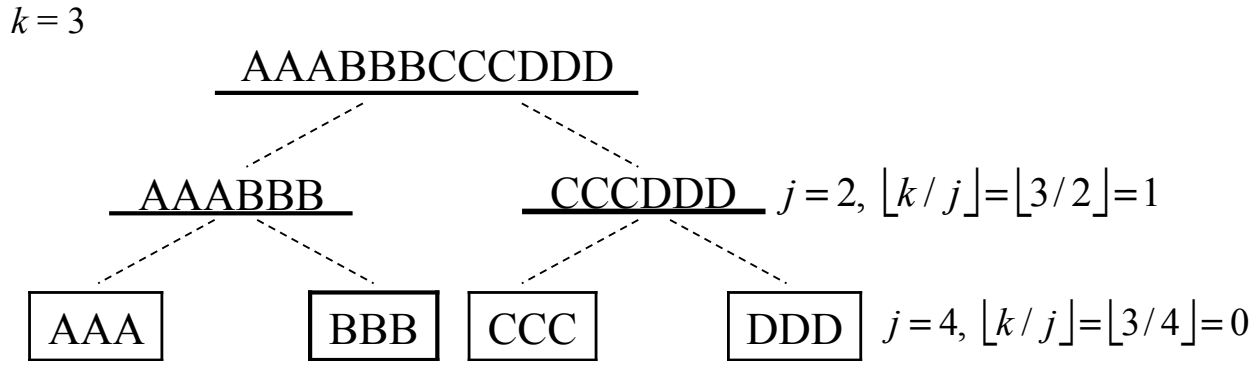

Figure 4.2: An example of this algorithm.

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.4.3, let us assume that we search for AT:

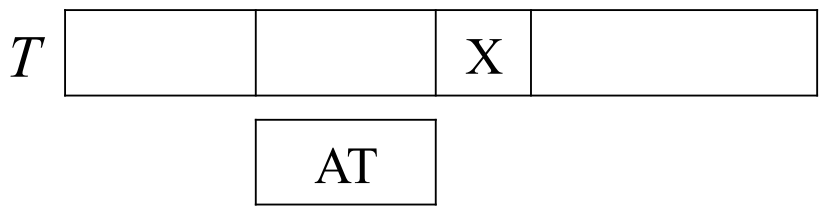

Figure 4.3: Searching for AT.

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:

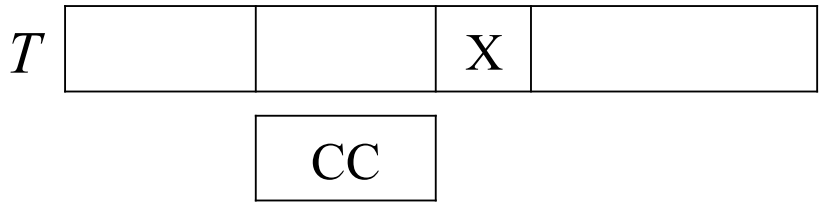

Figure 4.4: Searching for CC.

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:

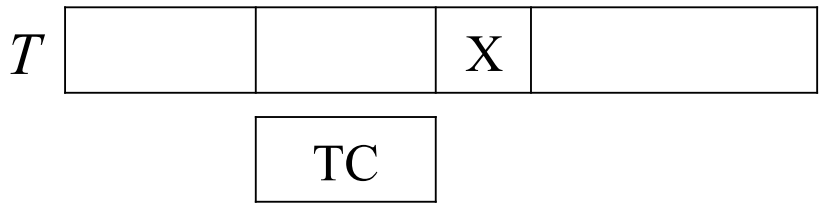

Figure 4.5: Searching for TC.

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.4.6, 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 4.6: Shift Table.

4.3 Example

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

Example 4.3.1. 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:

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 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:

T C C A A 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

4.4 Time Complexity

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

Chapter 5

System Testing

Gang Liao

Tests were conducted on a system composed of an Intel Core i7 quad core processor running at 2.5 GHz, with 1600 MHz and 16 GB DRAM.

We use the standard datasets in PBBS benchmark from CMU [13].

1. chr22.dna: A DNA sequence, about 34 million characters.
2. etext99: 105 Million characters, text from Guttenberg.

5.1 Test Cases and Test Results

As shown in Fig. 5.1 and Fig. 5.2, both data sets tell us the truth, that is, when m (the size of pattern string) is fixed to 32, Bit-Parallel algorithm is more efficient than the simple and fast algorithm in the real applications.

The result is quite reasonable. From the source code in the Appendix, we can see that the most part of the implementation of Bit-Parallel algorithm is bit manipulates. However, the simple and fast algorithm is implemented by recursive procedure, which need amounts of time and space.

5.2 Summary

In the real application, besides time and space complexity of algorithms, we also need to consider how big impact comes from computer architecture. When theoretical complexities of two algorithms are similar, at the most of time, the one which is more suitable in the architecture is better than the others, even the others have the better theoretical complexity.

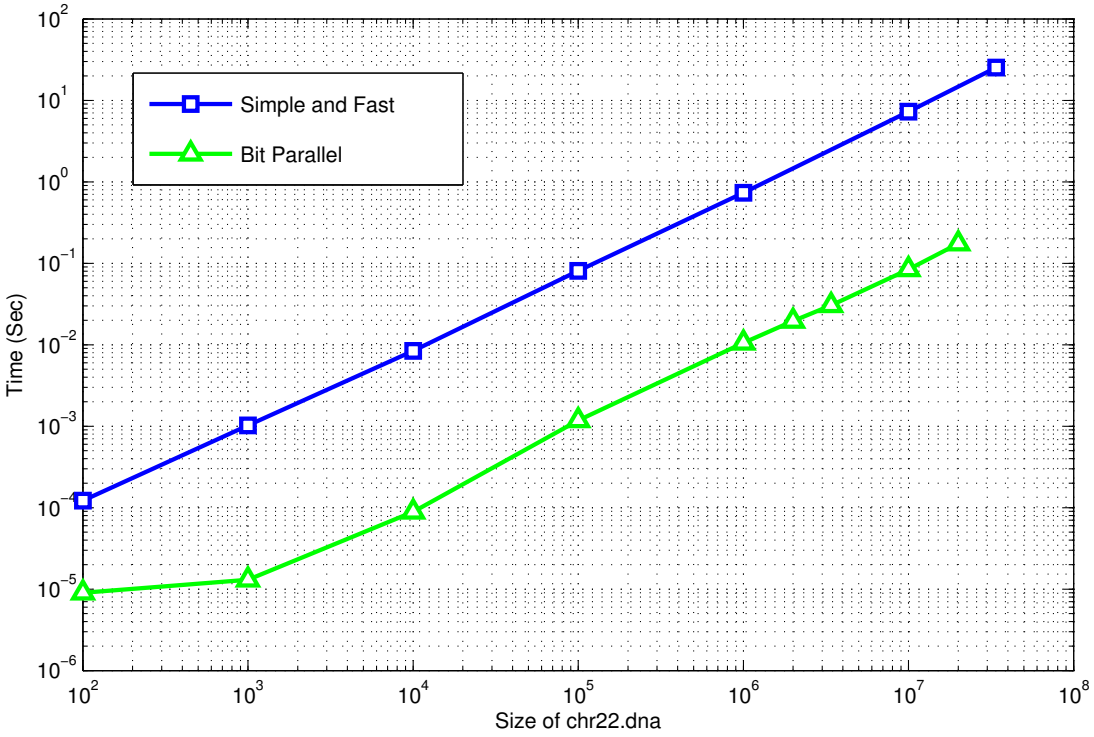

Figure 5.1: chr22.dna: A DNA sequence, about 34 million characters.

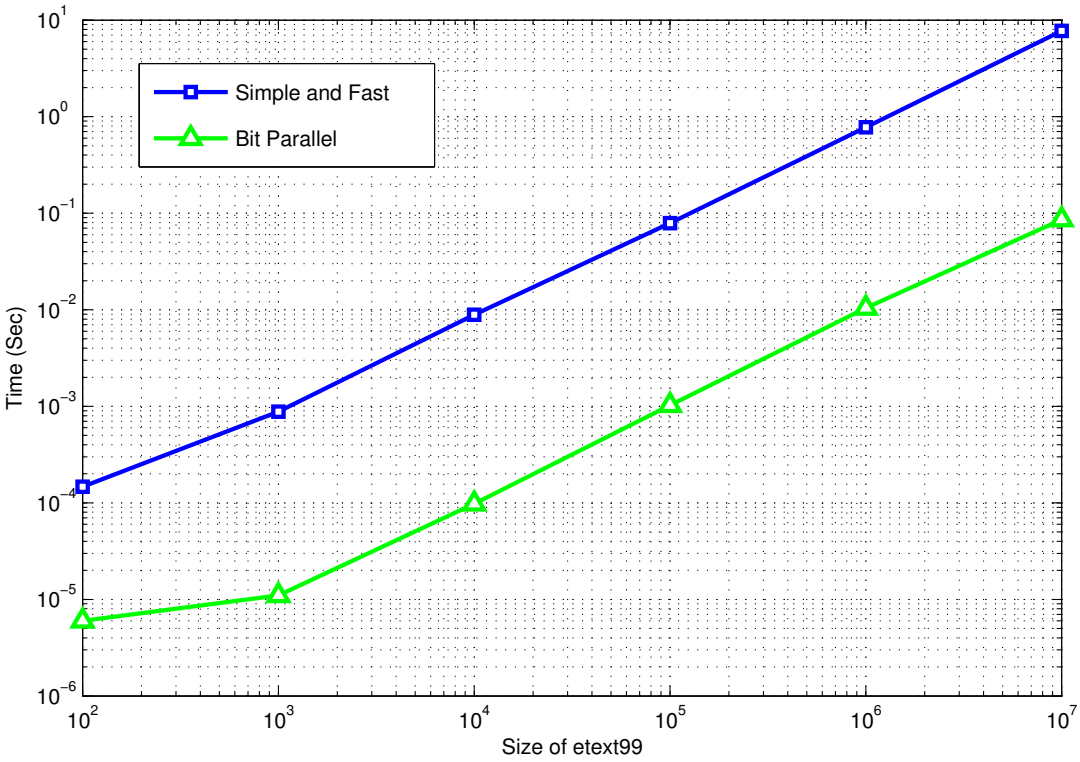

Figure 5.2: etext99: 105 Million characters, text from Guttenberg.

Chapter 6

Conclusion and Future Work

Gang Liao

6.1 Conclusion

In computer science, approximate string matching (often colloquially referred to as fuzzy string searching) is the technique of finding strings that match a pattern approximately (rather than exactly). The problem of approximate string matching is typically divided into two sub-problems: finding approximate substring matches inside a given string and finding dictionary strings that match the pattern approximately. It is one of the most important problem in bioinformatics.

In this project, we choose two algorithms, Bit-Parallel and Fast & Simple one, to solve the approximate string matching problem. Both algorithms work perfectly. Finally, the comparative evaluation of these algorithms is carried out. We use the standard data sets from CMU group to evaluate the performance and scalability of our algorithms. The result shows that Bit-Parallel algorithm is better than the other one in the vast majority of cases.

6.2 Future Work

Since, in the era of post petascale computing, how to parallel those algorithms more efficiency is one of the most important things both academics and industries need to figure out. We plan to parallel these algorithms on both Multi-core (CPUs) and Manycore (GPUs) architectures.

Using dynamic parallel model to optimize the recursive procedure of Fast and Simple one. To support extreme scale data, transform Bit-Parallel one to Vector Parallel algorithm.

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.

[13] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri and Kanat Tangwongsan. http://www.cs.cmu.edu/ pbbs/

Appendix A

Bit-Parallel Approximate String Matching

Gang Liao

We implemented Bit-Parallel algorithm using C/C++ as follows:

```
The MIT License (MIT)

Copyright (c) 2014 Gang Liao

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
    all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

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

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
```

```
#include<time.h>
#include<iostream>

using namespace std;

#define INDEX(c)     (c)
#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));
    }

}

void read_data(char *filename, char *buffer, int num){
    FILE *fh;
    fh = fopen(filename, "r");
    fread(buffer, 1, num, fh);
    buffer[num] = '\0';
    fclose(fh);
}

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

    clock_t start, end;
    char* filename = "/the local address of data sets";
    int n;                      //input size
    char *text;                   //data set pointer
    double runTime;

    printf("Please input the size of dataset you want to evaluate: \t");
    scanf("%d", &n);

    text = (char *) malloc((n+1)*sizeof(char));
    read_data(filename, text, n);

    //pattern length = 32
    string pattern = "Project Gutenberg Etexts are usually created from
   multiple editions";

    //asuume the numbers of error k is 10
    int k = 1; // allowed errors

    unsigned int set[256];
    // all the postions of each character c in pattern p

    memset(set, 0, sizeof(int)*256); // set zeroes into this array

```

```
65      start = clock();
66
67      initSet(pattern, set);
68
69      int pre = 0, cur = 1;
70
71      //int R[2][n+1];
72
73      char* R[2];
74      for(int i=0; i<2; i++)
75          R[i] = new char[n+1];
76
77      R[pre][0] = 0; //if the length of the text == 0 and error k == 0
78
79      //firstly , we need to get R0 table using shift-and algorithm
80      //under bit-parallel
81      for(int i = 1 ; i <= n; i++){
82
83          R[pre][i] = ((R[pre][i - 1] >> 1) | BIT) & set[INDEX(text[i - 1])];
84      }
85
86      //R0 -> R1 -> R2 ... -> Rk
87      for(int i = 1; i <= k; i++)
88      {
89          R[cur][0] = 0;
90          for(int j = 0; j < i; j++) {
91              R[cur][0] |= (1 << (31 - j));
92              //if the length of the text == 0 and error k == i
93          }
94
95          //O(kn*(m/w)) = O(kn)
96          //m = 32 = 4 bytes = one integer  (n = 256, k = 10)
97          for(int j = 1; j <= n; j++){
98
99              R[cur][j] = ((R[cur][j - 1] >> 1) & set[INDEX(text[j - 1])]) |
100                         (R[pre][j - 1]) | /*insertion*/
101                         (R[pre][j] >> 1) | /*deletion*/
102                         (R[pre][j - 1] >> 1) /*substitution*/ | BIT;
103
104         }
105
106         cur = !cur;  //exchange the index, 1 to 0 or 0 to 1
107         pre = !pre;  //exchange the index, 1 to 0 or 0 to 1
108
109     }
110
111     end = clock();                  //record the end time
112     runTime = (end - start) / (double) CLOCKS_PER_SEC ;   //run time
113
114     cout << "NUM: "<< n <<"\t Time: " << runTime << " Sec"<<endl;
115
116     return 0;
117 }
```

Appendix B

Very Fast and Simple Approximate String Matching

Wentao Hu

We implemented Very Fast and Simple algorithm using C/C++ as follows:

```
The MIT License (MIT)

Copyright (c) 2014 Wentao Hu

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
    all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

```
//
//
//  FastMatch.cpp
//
//  Created by Wentao Hu on 11/21/14.
//  Copyright (c) 2014 wentao. All rights reserved.
//

```

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <math.h>
using namespace std;

void read_data(char *filename, char *buffer, int num){
    FILE *fh;
    fh = fopen(filename, "r");
    fread(buffer, 1, num, fh);
    buffer[num] = '\0';
    fclose(fh);
}

//the edit distance of string1 and string2
int edit_dist(string string1, string string2)
{

    int m = (int)string1.length();
    int n = (int)string2.length();
    int i,j;
    int edit[m+1][n+1];
    for (i=0; i<=m;i++)
        edit[i][0]=i;
    for (j=0; j<=n; j++)
        edit[0][j]=j;
    for (i=1;i<m+1;i++)
        for (j=1;j<n+1;j++)
            edit[i][j]=min(min(edit[i-1][j]+1,edit[i][j-1]+1),edit[i-1][j
   -1]+(int)(!(string1[i-1]==string2[j-1])));
    return edit[m][n];
}

// whether the edit distance between pattern and substrings of textWindow can
    be less than or equal to k
int edit_window(string textWindow,string pattern,int k)
{
    int length =(int) (textWindow.length()-pattern.length());
    for (int i=0; i<=length;i++)
        if (edit_dist(textWindow.substr(i,i+pattern.length()),pattern) <= k)
            return 1;
    return 0;
}



/* bcTable. Because we use lower case letters only, there are 26 different
   lower case letters.
 string p : the pattern
 unsigned int* bc : bcTable
 int seg_length : the smallest length of pattern segments
 */
void bcTable(string p, unsigned int* bc, int seg_length)
```

```
{
    int i;
    for (i = 0; i < p.length(); i++)
        bc[p[i]] = bc[p[i]] | (1 << (p.length()-i-1));
    for(i = 0;i<256;i++)
    {
        int j;
        int k;
        int value=seg_length+1;
        for(j=0;j<p.length();j=j+seg_length)
        {
            for(k=1;k<=seg_length+1;k++)
                if (((bc[i]>>j)&(1<<(k-1))) != 0) break;
            if(k<value) value=k;
        }

        bc[i]=value;
    }
}


// to calculate the edit distance limits of k during each step
vector<int> segLength(int k)
{
    vector<int> kl;
    for(;k!=0;k=k/2)
        kl.push_back(k);
    return kl;
}


/*main method: recursive function
 parameters:
 text, pattern : the original text and pattern
 textWindow, patternWindow : changes at each step
 textWindowStart, patternWindowStart : the start position of each window in
    original text and pattern
 kl: the segLength(k)
 n: used to find which the step the function is in

 returns the start position of textWindow in text, otherwise returns -1.
 */
int fastmach(string text,string pattern,string textWindow,string
    patternWindow,int textWindowStart,int patternWindowStart,vector<int> kl,
    int n)
{
    if (n==0 && edit_window(textWindow,pattern,kl[0])==1)
        return textWindowStart;
    else
        if (edit_window(textWindow,patternWindow,kl[n])==1)
        {
            string newPatternWindow;
            if (patternWindowStart%2==0)
```

```
                newPatternWindow = pattern.substr(patternWindowStart*
    patternWindow.length(),2*patternWindow.length());
            else
                newPatternWindow = pattern.substr((patternWindowStart-1)*
    patternWindow.length(),2*patternWindow.length());
            int end =(int)(textWindowStart+newPatternWindow.length()+kl[n-1])
    ;
            if((textWindowStart = textWindowStart-kl[n-1])<0)
                textWindowStart =0;
            if(end > text.length())
                end = (int) text.length();
            textWindow = text.substr(textWindowStart,end-textWindowStart);
            return fastmach(text,pattern,textWindow,newPatternWindow,
    textWindowStart,patternWindowStart/2,kl,n-1);
        }
    return -1;
}


int main(int argc, const char * argv[]) {
    while (1) {

    clock_t start, end;
    char* filename = "/Users/gang/gangliao/code/Approximate-String-Matching/
    bit-parallel approximate string/etext99";
    int len;                    //input size
    char *data;                 //data set pointer
    double runTime;

    printf("Please input the size of dataset you want to evaluate: \t");
    scanf("%d", &len);

    data = (char *) malloc((len+1)*sizeof(char));
    read_data(filename, data, len);

    string text = data;

    //pattern length must be power(2,integer), such as 4, 8, 16, 32, ...
    //pattern length = 32
    string pattern = "abababababababababababababababababababbababababab";
        pattern =pattern.substr(0,32);

    int k=1;    // the error limit: k

    unsigned int bc[256];   //bcTable
    for (int i=0;i<256;i++)
        bc[i] = 0;     // Initialize bcTable = 0;

    vector<int> kl = segLength(k);
    vector<int> value;

    start = clock();

```

```
158     int n=(int)kl.size();    // the steps we use in recursive function
    fastmach
159     int segmentLength=(int)pattern.length()/pow(2, n); // the smallest length
     of pattern segments
160
161     bcTable(pattern,bc,segmentLength); // calculate bcTable
162
163     string textWindow;
164     string patternWindow;
165     int textWindowStart;
166
167
168
169     for (int j=0; j<pow(2, n);j++)
170     {
171         int i = 0;
172         patternWindow = pattern.substr(j*segmentLength,segmentLength);
173         while (true)
174         {
175             textWindow = text.substr(i,patternWindow.length());
176             if(edit_window(textWindow, patternWindow,0)==1)
177             {
178                 if (j%2==0)
179                     patternWindow = pattern.substr(j*segmentLength,2*
    segmentLength);
180                 else
181                     patternWindow = pattern.substr((j-1)*segmentLength,2*
    segmentLength);
182                 if((textWindowStart = i-kl[n-1]) < 0)
183                     textWindowStart =0;
184                 int end =i+(int)patternWindow.length()+kl[n-1];
185                 if(end > text.length())
186                     end = (int)text.length();
187                 textWindow = text.substr(textWindowStart,end-textWindowStart)
    ;
188                 int start = fastmach(text,pattern,textWindow,patternWindow,
    textWindowStart,j/2,kl,n-1);
189                 if(start!=-1)
190                 {   int m=0;
191                     for (;m<int(value.size());m++)
192                         if(start == value[m]) break;
193                     if (m==value.size())
194                     {
195                         value.push_back(start);
196                         //printf("The textWindow's start position in text is
    %d \n", start+1);
197                         int textend = start+(int)pattern.length()+2*k;
198                         if (textend > text.length())
199                             textend = (int) text.length();
200                         //cout<< "the textWindow is " + text.substr(start,
    textend-start) <<endl<<endl;
201                     }
202                 }
203
```

```
204                 }
205
206                 if (i+patternWindow.length()>text.length()-1)
207                     break;
208                 i=i+bc[text[i+patternWindow.length()]];
209                 if ((i+patternWindow.length())>text.length()-1)
210                     break;
211             }
212         }
213
214         end = clock();                  //record the end time
215         runTime = (end - start) / (double) CLOCKS_PER_SEC ;   //run time
216         cout << "NUM: "<< len <<"\t Time: " << runTime << " Sec"<<endl;
217         }
218         return 0;
219 }
```