On the String Matching with *k* Differences in DNA Databases

Yangjun Chen and Hoang Hai Nguyen

Dept. Applied Computer Science, University of Winnipeg, Canada

Introduction

In this paper, we discuss an efficient and effective index mechanism for the string matching with *k* differences, by which we will find all the substrings of a target string *y* of length *n* that align with a pattern string *x* of length *m* with not more than *k* insertions, deletions, and mismatches. A typical application is the searching of a DNA database, where the size of a genome sequence in the database is much larger than that of a pattern.

- **String Matching with *k* Differences**

To find all the occurrences of a pattern string $x = x_1x_2 \dots x_m$ in a target string $y = y_1y_2 \dots y_n$ with at most *k* differences, where x_i, $y_j \in \Sigma$, a given alphabet. In general, we distinguish among three kinds of differences:

1. **A character of the pattern corresponds to a different character of the target. In this case we say that there is a mismatch between the two characters;**
2. **A character of the target corresponds to "no character" in the pattern (an insertion into the pattern); and**
3. **A character of the pattern corresponds to "no character" in the target (a deletion from the pattern).**

The following example shows an occurrence of a pattern in a target with *k* = 3.

```
b p d q e g h      <----- pattern
| | | | | | |
c b c d   e f g h i <----- target
```

- **The basic method based on Dynamic Programming paradigm**

Prefix of pattern: $X_i = x_1x_2 \dots x_i$; prefix of target: $Y_j = y_1y_2 \dots y_j$

$D(0, j) = j$, $0 \le j \le n$; $D(i, 0) = i$, $0 \le i \le m$;

$$D(i, j) = min \begin{cases} D(i-1, j) + w(x_i, \phi) \\ D(i-1, j-1) + \delta(x_i, y_j) \\ D(i, j-1) + w(\phi, y_j) \end{cases}$$

D(i-1, j-1) D(i-1, j) D(i, j-1) D(i, j)

where $D(i, j)$ represents the distance between x_i and y_j, $w(x_i, y_j)$ is the cost to change x_i to y_j, $\delta(x_i, y_j)$ is 1 if $x_i = y_j$; therwise $\delta(x_i, y_j) = w(x_i, y_j)$.

	j	0	1	2	3	4	5	6	7
i			*a*	*c*	*a*	*t*	*a*	*T*	*g*
0		0	0	0	0	0	0	0	0
1	*g*	1	1	1	1	1	1	1	0
2	*c*	2	2	1	2	2	2	2	1
3	*a*	3	2	2	1	2	2	3	2
4	*c*	4	2	2	2	2	3	3	3
5	*a*	5	4	3	**2**	3	**2**	4	4

Methods

Combine Dynamic Programming Paradigm with BWT Array Searching

- **BWT array *L* of *y*, denoted as *BWT*(*Y*), can be established by using the suffix array *SA* of *y*:**

$$L[i] = \begin{cases} \$, & \text{if } SA[i] = 0; \\ y[SA[i] - 1, & \text{otherwise.} \end{cases}$$

The generation of *BWT*(*y*) can also be described in a different way, which is a little tedious, but enables us to observe why it can be used to speed up the string matching

To produce *BWT*(*y*), we first rotate *y* consecutively to create |*y*| different strings, sort them lexicographically. Then, write them stacked vertically as shown in the 6th column of the following table, where each character is subscripted to represent its position in the original *y*. (That is, we rewrite y as $g_1t_1a_1t_2a_2c_1a_3$.) For example, a_2 represents the second appearance of a in y; and t_1 the first appearance of t in y. In the same way, we can check all the other appearances of different characters.

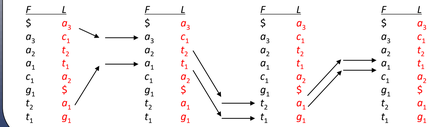
Suffix	Sorted suffix	SA_y	r_F	*F*	Sorted rotations	*L*	r_L
gtataca$	$	7	-	$	$$g_1t_1a_1t_2a_2c_1a_3$$	*a*	1
tataca$	a$	6	1	*a*	$a_3$$g_1t_1a_1t_2a_2c_1$	*c*	1
ataca$	aca$	4	2	*a*	$a_2c_1a_3$$g_1t_1a_1t_2$	*t*	1
taca$	*ataca*$	2	3	*a*	$a_1t_2a_2c_1a_3$$g_1t_1$	*a*	2
aca$	*ca*$	5	1	*c*	$c_1a_3$$g_1t_1a_1t_2a_2$	*a*	2
ca$	*gtataca*$	0	1	*g*	$g_1t_1a_1t_2a_2c_1a_3$$	$	-
a$	*taca*$	3	1	*t*	$t_2a_2c_1a_3$$g_1t_1a_1$	*a*	3
$	*tataca*$	1	2	*t*	$t_1a_1t_2a_2c_1a_3$$g_1$	*g*	1

- **To find all occurrences of *x* in *y*, search *x* backwards against *L* = BWT(*y*):**

$y = g_1t_1a_1t_2a_2c_1a_3$$

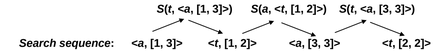
$x = tata$

$$search(z, \pi) = \begin{cases} \langle z, [\alpha, \beta] \rangle, & \text{if } z \text{ appears in } L_\pi; \\ \phi, & \text{otherwise.} \end{cases}$$



Methods (continued)

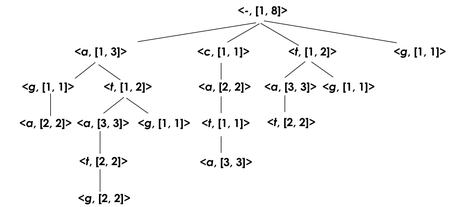
- **String matching by using the BWT array can be represented as a sequence, called a search sequence:**



- **By the dynamic programming computation, replace the target string by the corresponding search sequence.**

pattern: *x* = *acacg* ($\bar{x}$ = *gcaca*); *k* = 2.

target: *y* = *gtataca* (*BWT(y)= actta$ag*);



In the above 'search tree', each path represents a search sequence.

- **Three possible improvements:**

1. **Replacing part of a search tree by searching part of the suffix tree built for a pattern with no dynamic programming matrix established**
2. **Recognizing similar paths**
3. **Pattern partition to get subpatterns each with a smaller *k* value**

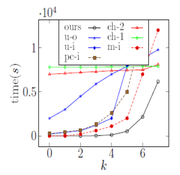
Experiments

- **In our experiments, we have tested altogether 7 strategies:**

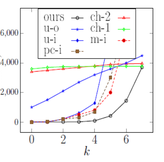
1. **Ukkonen's onlline method (u-o for short),**
2. **Chang-Lawer's first method (ch-1 for short),**
3. **Chang-Lawer's second method (ch-2 for short),**
4. **Ukkonen's index-based method (u-i for short),**
5. **Myers's index-based method (m-i for short),**
6. **Peri-Culpepper's index-based method (pc-i for short), and**
7. **ours, discussed in this paper.**

Data

Reference sequences*	Num. characters	Time (s) for building BWT(y)
Gorilla	3,063,403,506	406.817
Danio Rerio (ZebraFish)	1,373,472,378	173.142
Gorilla Chr1	228,908,641	25.03
Protein-1	144,000,000	15.92
Protein-2	30,000,000	3.04



Gorilla genome



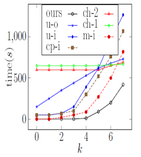
ZebraFish genome

Conclusions

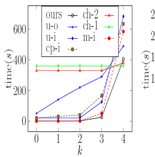
In this paper, a new index-based method is discussed for solving the string matching with *k* differences. This is highly important to searching genome and protein sequences in modern DNA databases, as well as genetics research, especially, for very long sequences. The average time complexity of our method is bounded by $O(k \cdot |\Sigma|^{2k})$. Together with pattern partition, our method can achieve more than 1000-fold improvement than the existing methods.

Bibliography

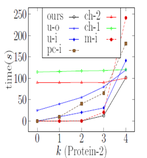
1. **W.I. Chang and E.L. Lawler. 1994. Sublinear Approximate String-Matching and Biological Applications. *Algorithmica* 12, 4 (1994), 327–344.**
2. **E. Ukkonen. 1993. Approximate String-Matching over Suffix Trees. In *Proc of the .4th Annual Symposium on Combinatorial Pattern Matching*. Springer-Verlag, 228–242.**
3. **M. Burrows and D. J. Wheeler, A Block-sorting Lossless Data Compression Algorithm, Systems Research Center, May 1994.**
4. **B. Langmead, Introduction to the Burrows-Wheeler Transform and FM Index , http://www.cs.jhu.edu/langmea/resources/bwt_fm.pdf, 2013.**



Gorilla Chr1



Protein 1



Protein 2