Through Inspiration, Discovery
King Abdullah University of Science and Technology

Approximate String Matching: Bit-Parallel Approach

Midterm Presentation

Gang Liao, liao.gang@kaust.edu.sa
Fatima Zohra Smailiy, fatimazohra.smaili@kaust.edu.sa
Wentao Hu, wentao.hu@kaust.edu.sa
Guangming Zang, guangming.zang@kaust.edu.sa

Background

Approximate string matching is one of the main problems in classical algorithms, with applications to text searching, computational biology, pattern recognition, text processing, spell checking, spam filtering, etc.

Background

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.

Example:

Insertion: cat → c**a**s**t**

Deletion: **c**at → at

Substitution: ca**t** → ca**r**

Background

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 $1 \leqq i \leqq n$ such that there exists a suffix A of $T_{1,i}$ and $\mathrm{d}(A,P) \leqq k$ where $\mathrm{d}(x,y)$ is the edit distance (or **difference**) between x and y.

Background

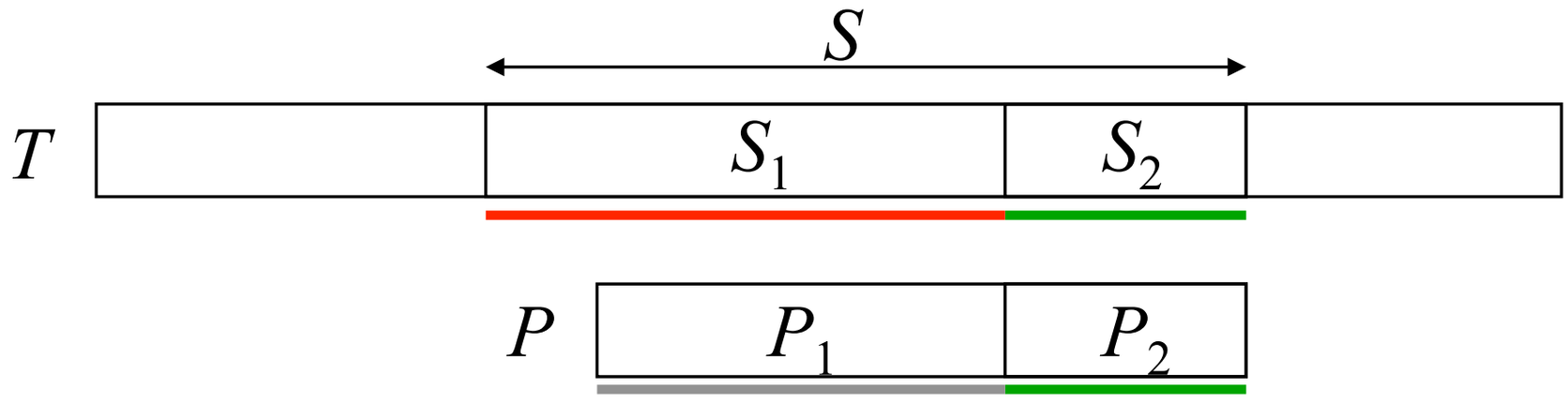
Example: T=deaabeg, P=aabac and k=2.

For i=5. $T_{1,5}$= deaab.

We note that there exists a suffix A=aab of $T_{1,5}$ such that d(A,P)=d(aab,aabac)=2.

Our approach is based upon the following observation:

Case 1 :



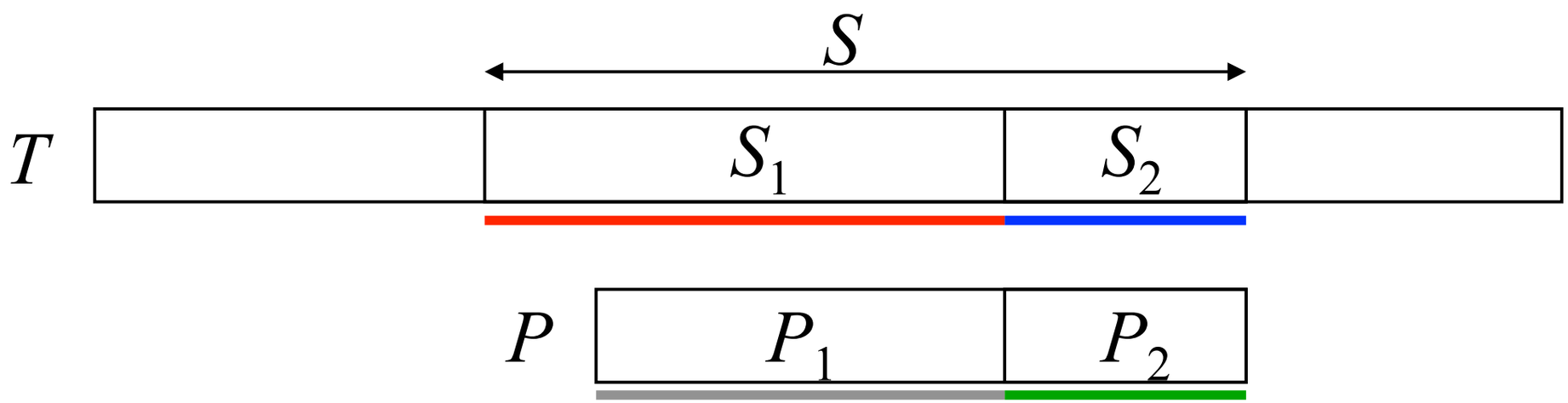
Let S be a substring of T.

If there exists a suffix S_2 of S and a suffix P_2 of P such that

$d(S_2, P_2) = 0$, and $d(S_1, P_1) \leqq k$,

we have $d(S, P) \leqq k$.

Case 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

$d(S_2, P_2) = 1$, and $d(S_1, P_1) \leqq k\text{-}1$,

we have $d(S, P) \leqq k$.

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

$$R^k(i,j) \begin{cases} = 1 \text{ if there exists a suffix } A \text{ of } T_{1,i} \text{ such that } d(A,P_{1,j}) \leqq k. \\ = 0 \text{ otherwise.} \end{cases}$$

where $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$.

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 [S80](String Matching with Errors).

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

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

$R^0(i,0) = 1$ for all i. $\quad R^0(0,j) = 0$ for all j.

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

Shift-And

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

We use & and v to represent AND and OR operation. We further use $a^m b^n$ to denote $(\underbrace{a,..a}_{m},\underbrace{b,..b}_{n})$.

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

Given an alphabet α and a pattern *P*, we need to know where α appears in *P*.

This information is contained in a vector Σ(α) which is defined as follows:
Σ(α)[*i*]=1 if p_i=α .
Σ(α)[*i*]=0 if otherwise.

Example: *P* : aabac
Σ(a) =(1,1,0,1,0)
Σ(b) =(0,0,1,0,0)
Σ(c) =(0,0,0,0,1)

Example:

T : aabaacaabacab,

P : aabac and k=0.

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

Σ(a)	11010
Σ(b)	00100
Σ(c)	00001
*	00000

$$R_{10}^0 = ((R_9^0 >> 1) \vee 10^4) \,\&\, \Sigma[t_{10}]$$

$$=((0,0,0,1,0)\text{v}(1,0,0,0,0))\&(1,1,0,1,0)$$

$$=(1,0,0,1,0)$$

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

The Shift-And formula $R^0 = ((R^0_{i-1} >> 1) \vee 10^{m-1}) \,\&\, \Sigma(t_i)$

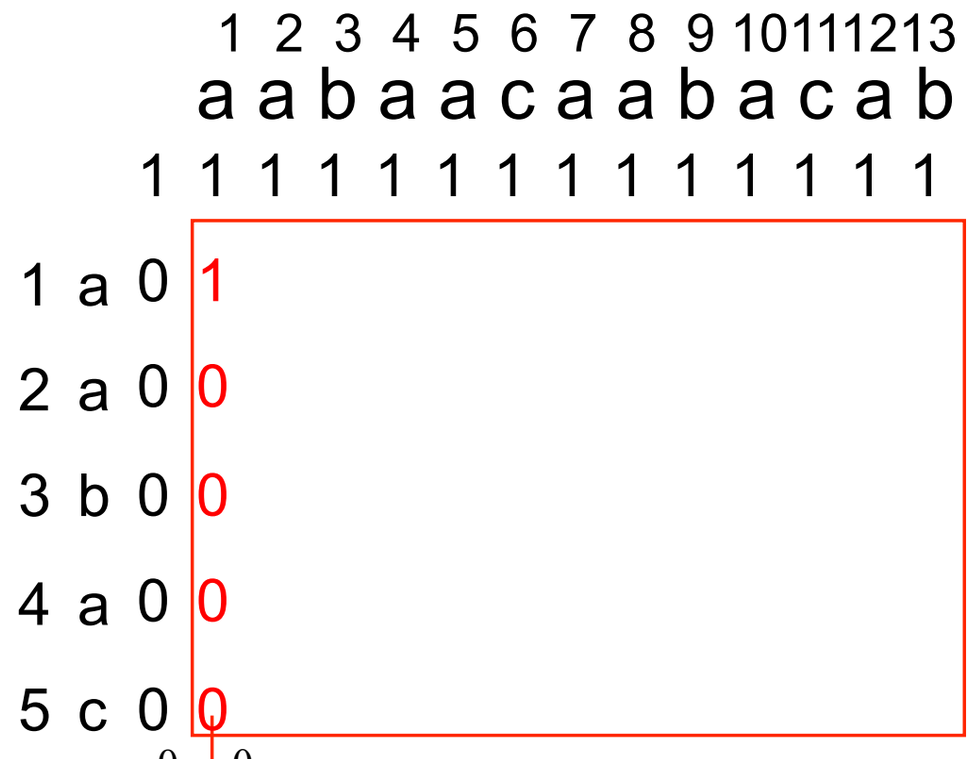
Example:

Text = aabaacaabacab

pattern = aabac

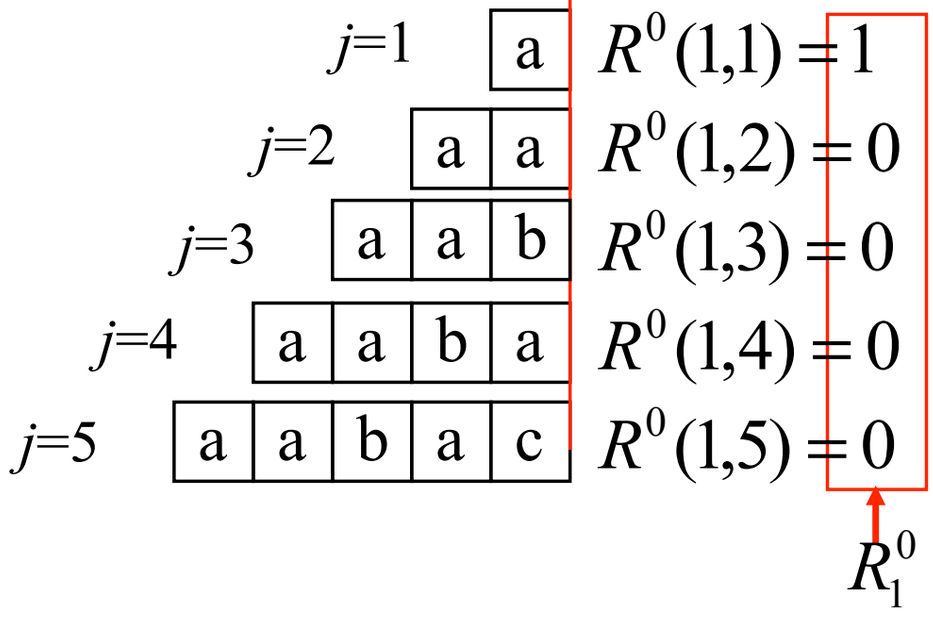
when i=1

Σ(a)	11010
Σ(b)	00100
Σ(c)	00001
*	00000



	10000	$(R^0_0 >> 1) \vee 10^{m-1}$
&	11010	Σ(a)
R^0_1	10000	

1	2	3	4	5	6	7	8	9	• • •
a	a	b	a	a	c	a	a	b	a c a b



The operation "&" will examine whether the $\Sigma(t_i)$ equals to the p_j or not currently.

The Shift-And formula $R^0 = ((R^0_{i-1} >> 1) \vee 10^{m-1}) \,\&\, \Sigma(t_i)$

Example:

Text = aabaacaabacab

pattern = aabac

when *i*=2

Σ(a)	11010
Σ(b)	00100
Σ(c)	00001
*	00000

			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	1	1	1	1	1	1	1	1	1	1	1	1
1	a	0	1	1											
2	a	0	0	1											
3	b	0	0	0											
4	a	0	0	0											
5	c	0	0	0											

$R^0_1 R^0_2$

1	2	3	4	5	6	7	8	9	. . .
a	a	b	a	a	c	a	a	b	a	c	a	b

j=1 a $R^0(2,1) = 1$

j=2 a a $R^0(2,2) = 1$

j=3 a a b $R^0(2,3) = 0$

j=4 a a b a $R^0(2,4) = 0$

j=5 a a b a c $R^0(2,5) = 0$

R^0_2

$$\begin{array}{rl} 11000 & (R^0_1 >> 1) \vee 10^{m-1} \\ \underline{\&\ \ 11010} & \Sigma(\mathrm{a}) \\ R^0_2\ 11000 & \end{array}$$

Example:

Text = aabaacaabacab

pattern = aabac

Σ(a)	11010
Σ(b)	00100
Σ(c)	00001
*	00000

			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	1	1	1	1	1	1	1	1	1	1	1	1	1
1	a	0	1	1	0	1	1	0	1	1	0	1	0	1	0
2	a	0	0	1	0	0	1	0	0	1	0	0	0	0	0
3	b	0	0	0	1	0	0	0	0	0	1	0	0	0	0
4	a	0	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	0	**1**	0	0

aabaacaabacab
aabac

Approximate String Matching

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

$$R^k(n,m)\begin{cases} = 1 \text{ if there exists a suffix } A \text{ of } T_{1,i} \text{ such that } \mathrm{d}(A,P_{1,j}) \leqq k. \\ = 0 \text{ otherwise.} \end{cases}$$

where $1 \leqq i \leqq n$ and $1 \leqq j \leqq m$.

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.

If $t_i = p_j$, and $R^k(i-1, j-1) = 1$, $R^k(i, j) = 1$.

Otherwise, $R^k(i, j) = R_I^k(i, j) \vee R_D^k(i, j) \vee R_S^k(i, j)$.

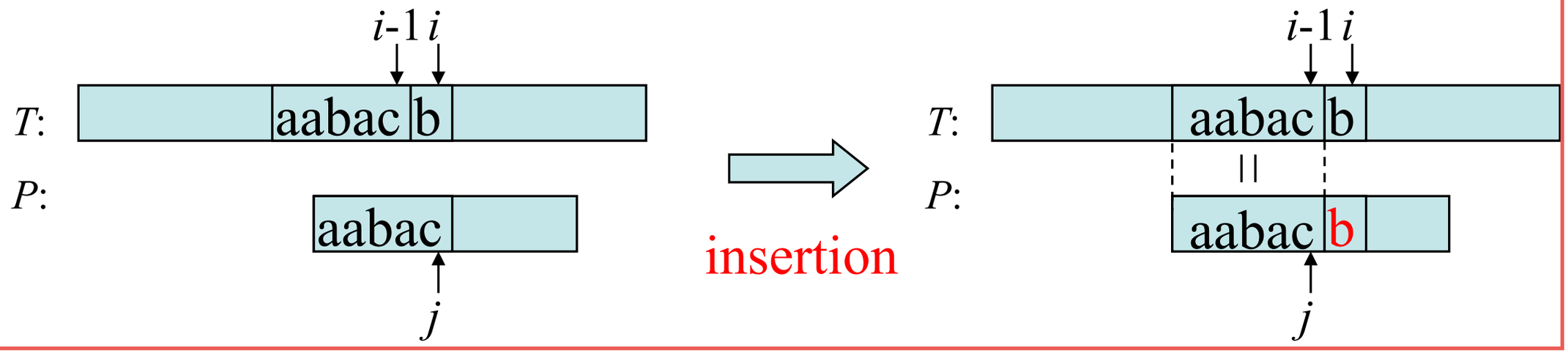
Insertion

$R_I^k(i, j) = 1$ if $t_i \neq p_j$ and $R^{k-1}(i-1, j) = 1$.

$R_I^k(i, j) = 0$ otherwise.

The insertion formula: $R_i^k = (((R_{i-1}^k >> 1) \vee 10^{m-1}) \,\&\, \sum(t_i)) \vee R_{i-1}^{k-1}$

Example: when k=1.



If t_i=p_j, and $R^k(i-1, j-1)=1$, $R^k(i, j)=1$.

Otherwise, $R^k(i, j)=R_I^k(i, j)\vee R_D^k(i, j)\vee R_S^k(i, j)$.

Deletion

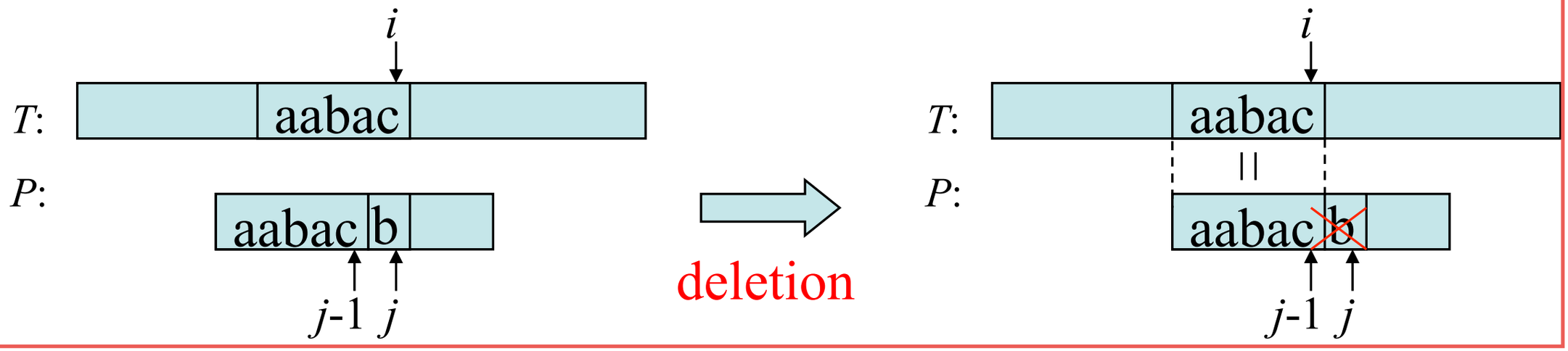
$R_D^k(i, j)=1$ if t_i≠p_j and $R^{k-1}(i, j-1)=1$.

$R_D^k(i, j)=0$ otherwise.

The deletion formula:

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

Example: when k=1.



If $t_i = p_j$, and $R^k(i-1, j-1) = 1$, $R^k(i, j) = 1$.

Otherwise, $R^k(i, j) = R_I^k(i, j) \vee R_D^k(i, j) \vee R_S^k(i, j)$.

Substitution

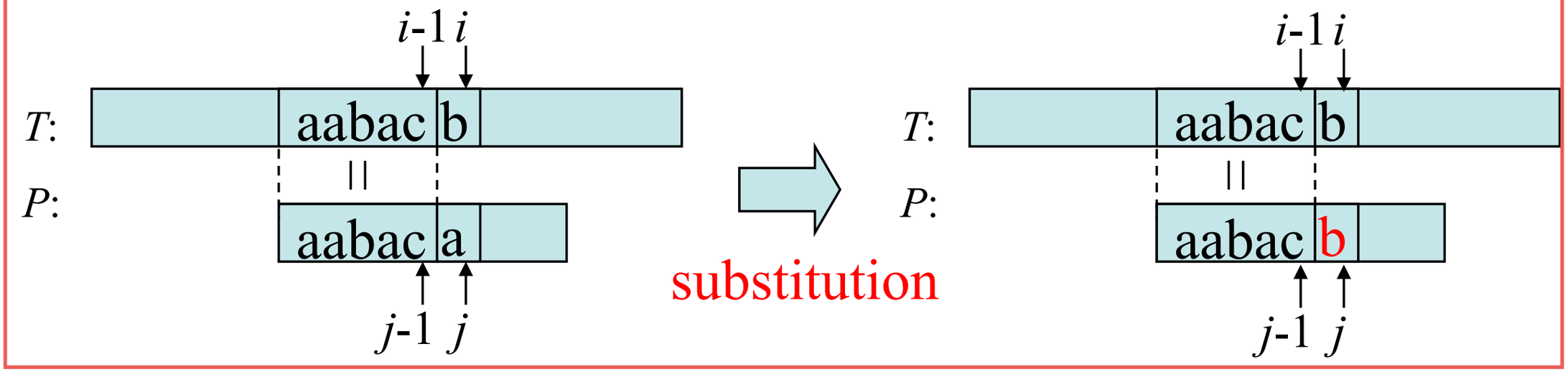
$R_S^k(i, j) = 1$ if $t_i \neq p_j$ and $R^{k-1}(i-1, j-1) = 1$.

$R_S^k(i, j) = 0$ otherwise.

The substitution formula:

$$R_i^k = (((R_{i-1}^k >> 1) \vee 10^{m-1}) \,\&\, \Sigma(t_i)) \vee ((R_{i-1}^{k-1} >> 1) \vee (10^{m-1}))$$

Example: when k=1.



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

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

The substitution : $R_i^k = (((R_{i-1}^k >> 1) \vee 10^{m-1}) \& \sum(T_i)) \vee ((R_{i-1}^{k-1} >> 1) \vee (10^{m-1}))$

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

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

$t_i = p_j$ insertion deletion substitution

and $R^k(i-1, j-1) = 1$

Time complexity

$$O(k\left\lceil \frac{m}{w} \right\rceil n)$$

k: error value

m: the length of pattern

n : the length of text

w : the size of word of computer

The Second Algorithm

Very fast and simple approximate string matching

- divide and Conquer
- edit distance

We still work on….

Very Fast and Simple Approximate String Matching

Gonzalo Navarro Ricardo Baeza-Yates

Department of Computer Science
University of Chile
Blanco Encalada 2120 - Santiago - Chile
{gnavarro,rbaeza}@dcc.uchile.cl

Abstract

We improve the fastest known algorithm for approximate string matching. This algorithm can only be used for low error levels. By using a new algorithm to verify potential matches and a new optimization technique for biased texts (such as English), the algorithm becomes the fastest one for medium error levels too. This includes most of the interesting cases in this area.

Key words: Approximate Matching, Text Searching, Information Retrieval.

1 Introduction

Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc.

The problem can be formally stated as follows: given a (long) text of length n, a (short) pattern of length m, and a maximal number of errors allowed k, find all text positions that match the pattern with up to k errors. The allowed errors are character insertions, deletions and substitutions. Text and pattern are sequences of characters from an alphabet of size σ. We call $\alpha = k/m$ the *error ratio* or *error level*.

In this work we focus on on-line algorithms, where the classical solution, involving dynamic programming, is $O(mn)$ time [8]. In the last years many

* This work has been supported in part by FONDECYT grant 1950622.

Thank You!

https://github.com/GangLiao/Approximate-String-Matching