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

Yangjun Chen and Hoang Hai Nguyen

Dept. of Applied Computer Science
University of Winnipeg, Canada

Outline

- Motivation
  - Statement of Problem
  - Related work
- Basic Techniques
  - Dynamic programming
  - BWT Arrays – A space-economic Index for String Matching
- Main algorithm for string Matching with *k* Differences
  - Search trees
  - Suffix trees over patterns, Similar paths, Pattern partition
- Experiments
- Conclusion and Future Work

Statement of Problem

- **String *matching with k differences:*** ***to find all the occurrences of a pattern string*** $x = x_1x_2 \ldots x_m$ **in a target string** $y = y_1y_2 \ldots y_n$ **with at most** k **differences, where** $x_i, y_j \in \sum$, **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).**

Statement of Problem

- **String *matching with k differences:***
- to find all the occurrences of a pattern string $x = x_1x_2 \ldots x_m$ in a target string $y = y_1y_2 \ldots y_n$ with at most k differences.

Example: $k = 3$

Related Work

- **Exact string matching**
  - **On-line algorithms:**

    Knuth-Morris-Pratt*, *Boyer-Moore*, *Aho-Corasick
  - **Index based:**

    suffix trees (*Weiner*; *McCreight*; *Ukkonen*), suffix arrays (*Manber*, *Myers*), BWT-transformation (*Burrow-Wheeler*), Hash (*Karp*, *Rabin*)
- **Inexact string matching**
  - **String matching with *k* mismatches - Hamming distance (*Landau, Vishkin; Amir at al.; Cole*; Chen, Wu)**
  - **String matching with *k* differences - Levelshtein distance (*Chang*, *Lampe*)**
  - **String matching with wild-cards (*Manber, Baeza-Yates*)**

Basic Techniques

- Dynamic Programming
  - to calculate distance between pattern and targets
- BWT transformation
  - to ‘fold’ the target strings

Dynamic Programming

- $X_i = x_1 x_2 \dots x_i$
- $Y_i = y_1 y_2 \dots y_j$

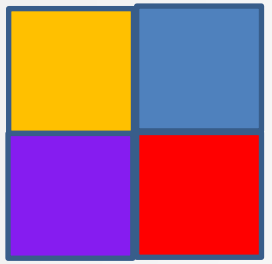
Time complexity: O(*mn*)

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

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

$D(i\text{-}1, j\text{-}1)$ $D(i\text{-}1, j)$

$D(i, j\text{-}1)$ $D(i, j)$



where $w(x_i, y_j)$ is the cost to change x_i to y_j, and $\delta(x_i, y_j)$ is 1 if $x_i = y_j$. Otherwise $\delta(x_i, y_j) = w(x_i, y_j)$.

Dynamic Programming

- **Example: *X* = *gcaca*, *Y* = *acatatg*, *k* = 2. For each y_j, the distance between $y_1 \ldots y_j$ and $x_1 \ldots x_i$ for all x_i will be calculated.**

	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

BWT Transformation

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

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

- **BWT array was proposed by M. Burrows and D.J. Wheeler in 1994.** (M. Burrows, D.J. Wheeler, (1994), *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation.)

BWT Transformation

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

L = BWT(Y)

BWT Transformation

- **Burrows-Wheeler Transform (*BWT*)**
- $y = g_1 t_1 a_2 t_2 a_3 c_1 a_3 \$$

$g_1\ t_1\ a_1\ t_2\ a_2\ c_1\ a_3\ \$$
$t_1\ a_1\ t_2\ a_2\ c_1\ a_3\ \$\ g_1$
$a_1\ t_2\ a_2\ c_1\ a_3\ \$\ g_1\ t_1$
$t_2\ a_2\ c_1\ a_3\ \$\ g_1\ t_1\ a_1$
$a_2\ c_1\ a_3\ \$\ g_1\ t_1\ a_1\ t_2$
$c_1\ a_3\ \$\ g_1\ t_1\ a_1\ t_2\ a_2$
$a_3\ \$\ g_1\ t_1\ a_1\ t_2\ a_2\ c_1$
$\$\ g_1\ t_1\ a_1\ t_2\ a_2\ t_2\ a_3$

Sort these sequences lexicographically.

Rank correspondence:

rk_F	F	L	rk_L	Suffix Array
-	$\$\ g_1\ t_1\ a_1\ t_2\ a_2\ t_2$	a_3	1	7
1	$a_3\ \$\ g_1\ t_1\ a_1\ t_2\ a_2$	c_1	1	6
2	$a_2\ c_1\ a_3\ \$\ g_1\ t_1\ a_1$	t_2	1	4
3	$a_1\ t_2\ a_2\ c_1\ a_3\ \$\ g_1$	t_1	2	2
1	$c_1\ a_3\ \$\ g_1\ t_1\ a_1\ t_2$	a_2	2	5
1	$g_1\ t_1\ a_1\ t_2\ a_2\ c_1\ a_3$	$\$$	-	0
1	$t_2\ a_2\ c_1\ a_3\ \$\ g_1\ t_1$	a_1	3	1
2	$t_1\ a_1\ t_2\ a_2\ c_1\ a_3\ \$$	g_1	1	2

rank: 3

rank: 3

$rk_F(e) = rk_L(e)$

BWT construction:

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

***SA*[...] – suffix array**

A suffix array can be established in O(*n*).

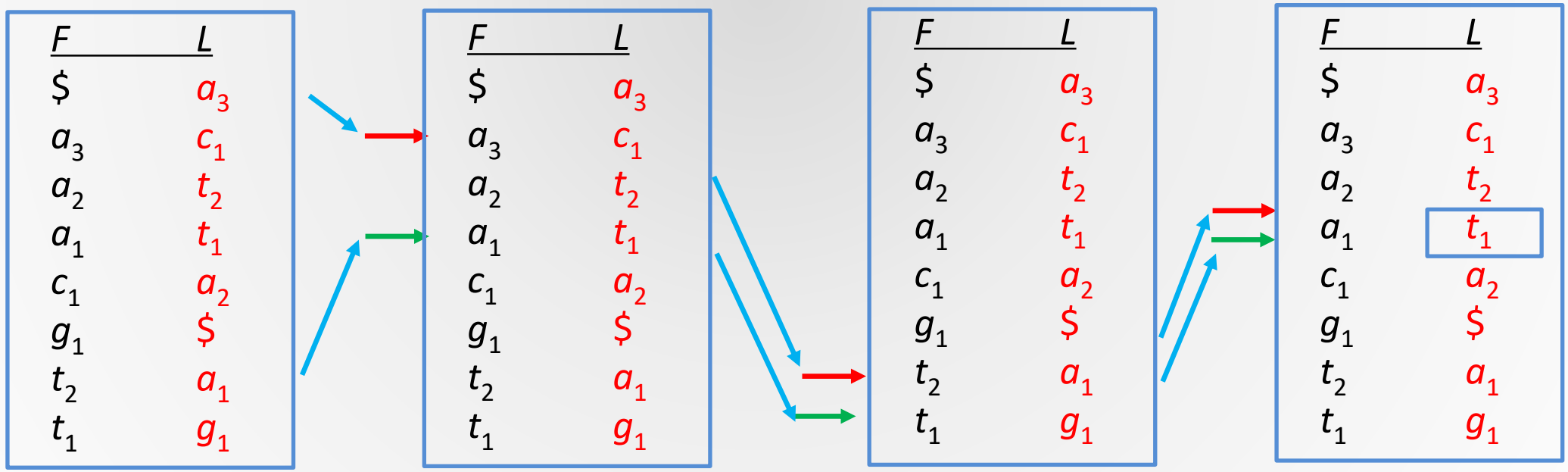
Backward Search of BWT-Index

- $y = g_1t_1a_1t_2a_2c_1a_3\$$
- $x = tata$

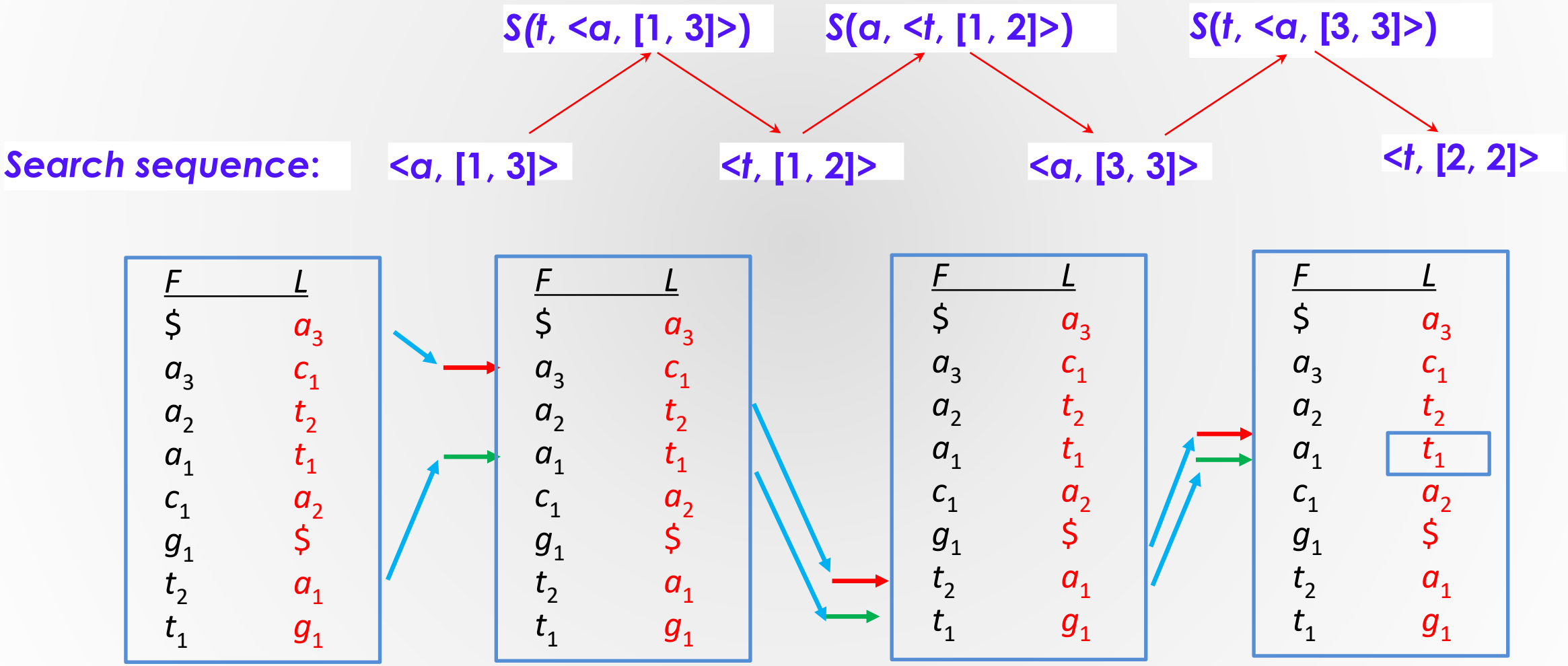
$$search(z, \pi) = \begin{cases} \langle z, [\alpha, \beta] \rangle, & \text{if } z \text{ appears in } L_\pi; \\ \phi, & \text{otherwise.} \end{cases}$$

<---- *Backward Search*

Z: a character π: a range in *F*
L_π: a range in *L*, corresponding to π



Backward Search of BWT-Index



rankAll

- Arrange $|\Sigma|$ arrays each for a character $x \in \Sigma$ such that $A_x[i]$ (the ith entry in the array for x) is the number of appearances of x within $L[1 .. i]$.
- Instead of scanning a certain segment $L[\alpha .. \beta]$ ($\alpha \leq \beta$) to find a subrange for a certain $x \in \Sigma$, we can simply look up A_x to see whether $A_x[\alpha - 1] = A_\alpha[\beta]$. If it is the case, then α does not occur in $\alpha .. \beta]$. Otherwise, $[A_x[\alpha - 1] + 1, A_x[\beta]]$ should be the found range.

Example

To find the first and the last appearance of t in $L[1 .. 3]$, we only need to find $A_t[1 - 1] = A_t[0] = 0$ and $A_t[3] = 2$. So the corresponding range is $[A_t[1 - 1] + 1, A_t[3]] = [1, 2]$.

F	L
\$	a_3
a_3	c_1
a_2	t_2
a_1	t_1
c_1	a_2
g_1	\$
t_2	a_1
t_1	g_1

$A_\$$	A_a	A_c	A_g	A_t
0	1	0	0	0
0	1	1	0	0
0	1	1	0	1
0	1	1	0	2
0	2	1	0	2
1	2	1	0	2
1	3	1	0	2
1	3	1	1	2

Reduce *rankAll*-Index Size

- *F*-ranks: $F_\alpha = \langle a; x_a, y_a \rangle$
- BWT array: *L*
- Reduced appearance array: A_α with bucket size β.
- Reduced suffix array: *SA** with bucket size γ.

Find a range:

$top' \leftarrow F(x_\alpha) + A_\alpha[\lfloor (top - 1)/\beta \rfloor] + r + 1$

$bot' \leftarrow F(x_\alpha) + A_\alpha[\lfloor bot/\beta \rfloor] + r'$

r is the number of α's appearances within $L[\lfloor (top - 1)/\beta \rfloor \beta \,..\, top - 1]$

r' is the number of α's appearances within $L[\lfloor bot/\beta \rfloor \beta \,..\, bot\,]$

i	*F*	*L*	rk_L	*SA*
1	\$	a_3	1	7
2	a_3	c_1	1	6
3	a_2	t_2	1	4
4	a_1	t_1	2	2
5	c_1	a_2	2	5
6	g_1	\$	-	0
7	t_2	a_1	3	1
8	t_2	g_1	1	2

$F_\alpha = \langle \alpha; x_\alpha, y_\alpha \rangle$

$F_\$ = \langle \$; 1, 1 \rangle$
$F_a = \langle a; 2, 4 \rangle$
$F_c = \langle c; 5, 5 \rangle$
$F_g = \langle g; 6, 6 \rangle$
$F_t = \langle t; 7, 8 \rangle$

\+

L
a_3
c_1
t_2
t_1
a_2
\$
a_1
g_1

\+

$A_\$$	A_a	A_c	A_g	A_t
0	1	0	0	0
0	1	1	0	0
0	1	1	0	1
0	1	1	0	2
0	2	1	0	2
1	2	1	0	2
1	3	1	0	2
1	3	1	1	2

\+

*SA**
7
6
4
2
5
0
1
2

String Matching with *k* Differences

- Different from the evaluation of an exact string matching, to find all the occurrences of $\overline{x} = z_1 z_2 \ldots z_m = x_m x_{m-1} \ldots x_1$ in BWT(y) for a target string y with k differences, a tree, instead of a single sequence, will be dynamically created. In such a tree, each path

$$v_0 \rightarrow v_2 \rightarrow \ldots \rightarrow v_l$$

corresponds to a search sequence. Each v_j is labeled with $\langle e_j, [\alpha_j, \beta_j] \rangle$. The D-vector of v_0 is $\langle 0, 1, \ldots, m \rangle^T$.

For $j > o$, we have

$$\begin{cases} D_j[0] = D_{j-1}[0] + 1 \\ D_j[i] = min\{D_j[i-1] + w(z_i, \phi), D_{j-1}[i] + w(\phi, e_j), D_{j-1}[i-1] + \delta(z_i, e_j), \}, \text{ for } i > 0. \end{cases}$$

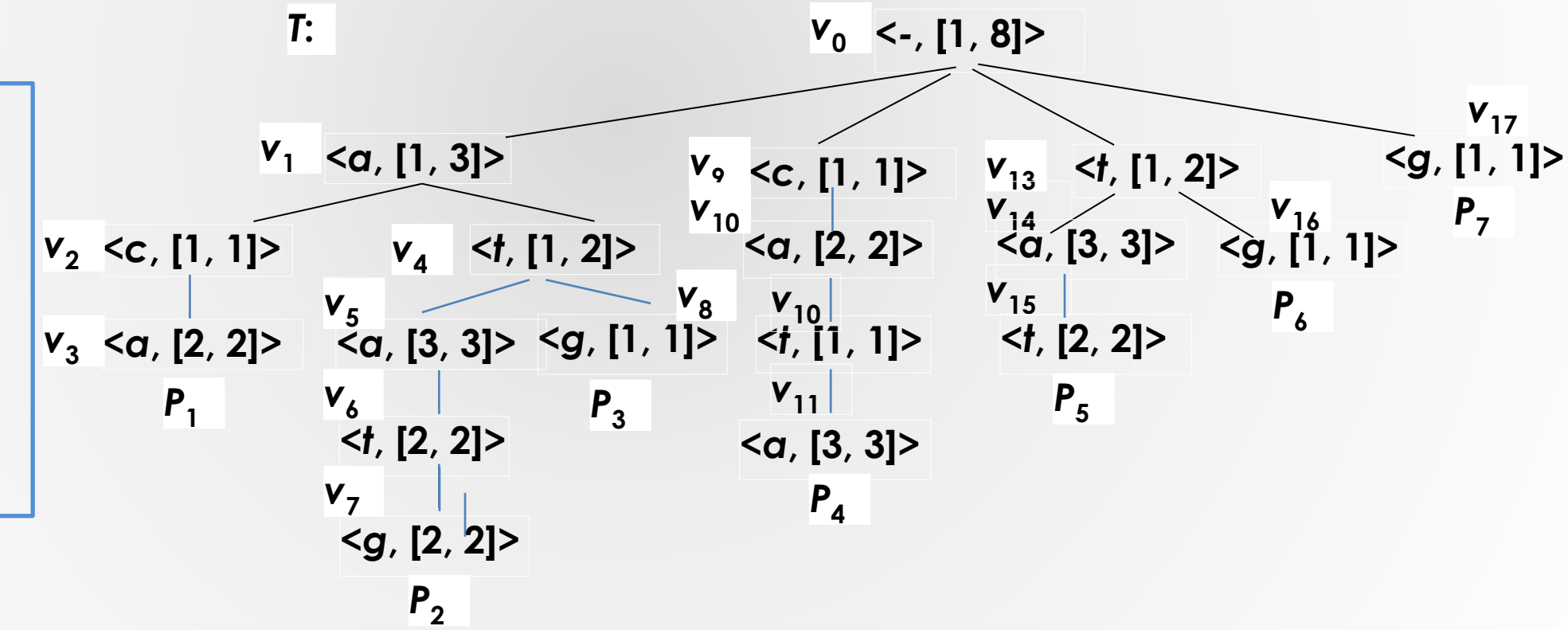
Search Trees

Definition (*search tree*) A search tree (*S*-tree for short) *T* with respect to *x* and *y* is a tree structure to represent the search of $\overline{x}$ against BWT(*y*). In *T*, each node is labeled with a pair <e, [α, β]> and there is an edge from *v* (= <e, [α, β]>) to *u* (= <e′, [α', β']>) if *S*(e′, *v*) = *u*. In addition, a special node is designated as the *root*, labeled with <-, [1, |*L*|]>, representing the whole BWT-array *L* = *BWT*(*y*).

Search Trees

- pattern: x = *acacg* ($\bar{x}$ = *gcaca*);
- target: y = *gtataca* ($\bar{y}$ = *acatatg*);
- k = 2.

F	L
\$	a_3
a_3	c_1
a_2	t_2
a_1	t_1
c_1	a_2
g_1	\$
t_2	a_1
t_1	g_1



String Matching with *k* Differences

D-vectors:

D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}
-	*a*	*c*	*a*	*t*	*a*	*t*	*g*	*g*	*c*	*a*	*t*	*a*	*t*	*a*	*t*	*g*	*g*
0	1	2	3	2	3	4	5	3	1	0	3	4	1	2	3	2	1
1	1	2	3	2	3	4	4	2	1	1	3	4	1	2	3	1	0
2	2	1	2	2	3	4	5	3	1	2	3	4	2	2	3	2	1
3	2	2	1	3	2	3	4	3	2	3	2	3	3	2	3	3	3
4	3	2	2	3	3	3	4	3	3	4	2	3	4	3	3	4	4
5	4	3	2	4	3	4	4	4	5	4	3	2	5	4	4	5	5

Computational Complexities

- Time complexity

  Worst case: $O(k \cdot |T|)$

  Average time complexity: $O(k \cdot |\Sigma|^{2k})$

- Space complexity

  Worst case: $O(km + n)$

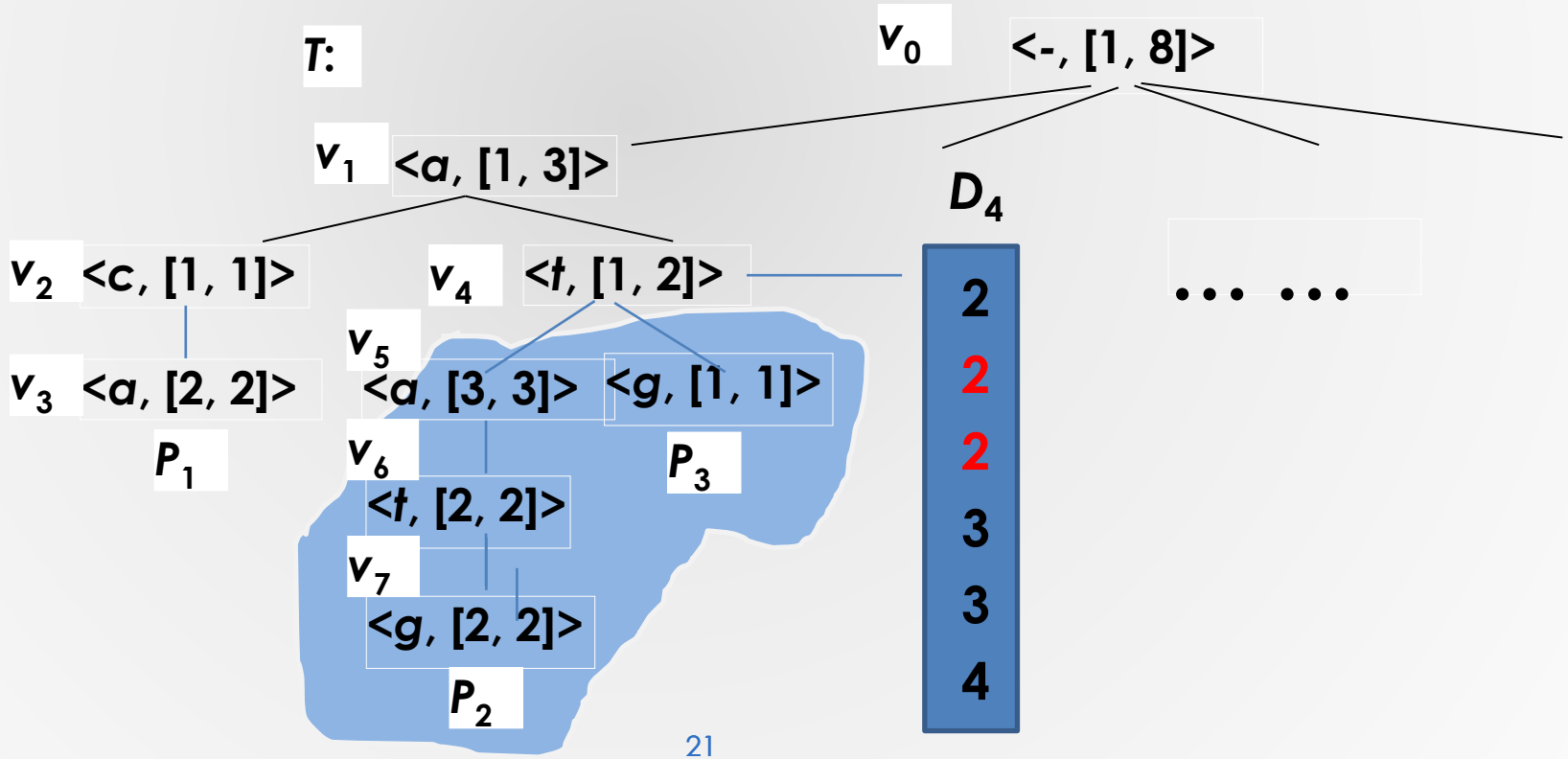
- Existing methods:

time complexity – $O(k \cdot n)$; space complexity – $O(m + n)$

Improvement-1

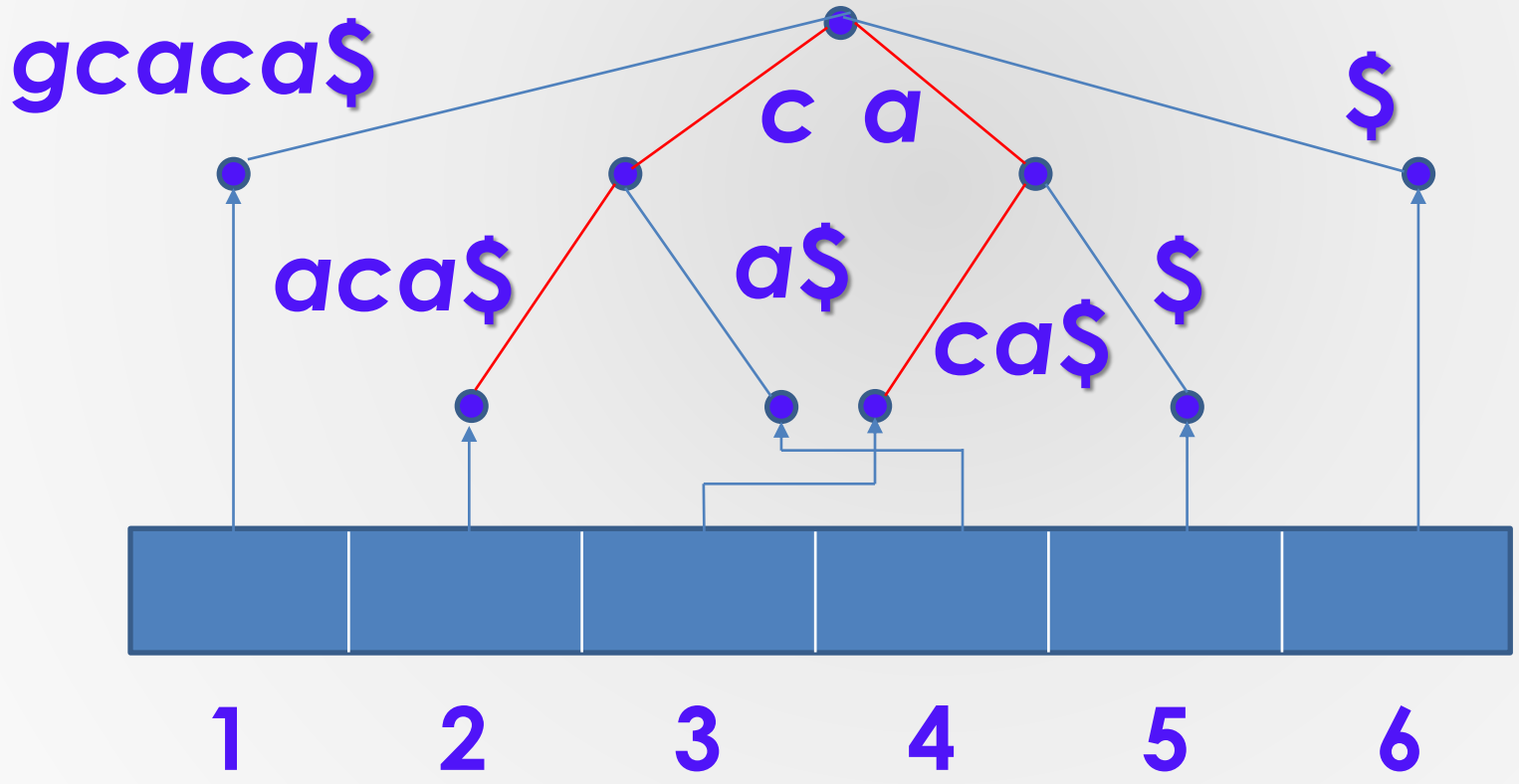
- Searching suffix trees over patterns to replace searching part of *T*
  - pattern: *x* = *acacg* ($\bar{x}$ = *gcaca*); target: *y* = *gtataca* ($\bar{y}$ = *acatatg*); *k* = 2.

F	*L*
$	a_3
a_3	c_1
a_2	t_2
a_1	t_1
c_1	a_2
g_1	$
t_2	a_1
t_1	g_1



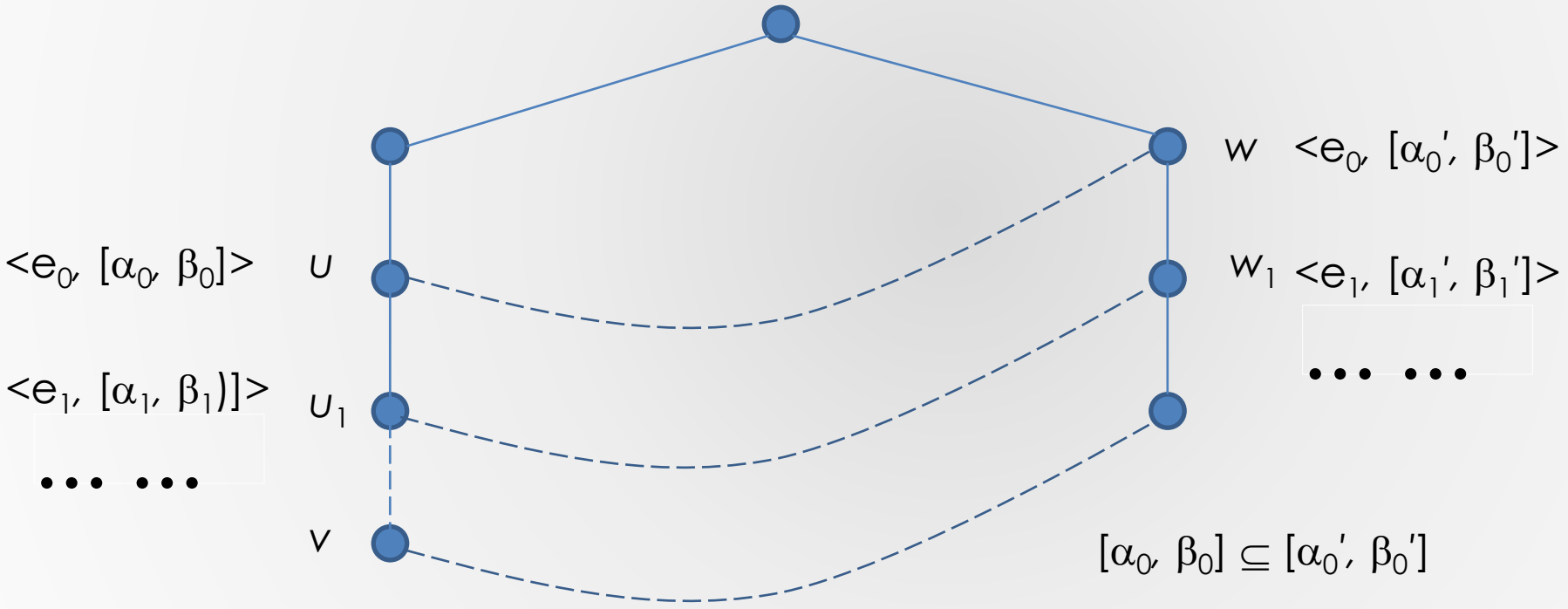
Improvement-1

- Suffix tree over $\bar{x}$ = *gcaca*



Improvement-2

- **Recognizing similar paths**



Pattern Partition

- As *k* increases, the performance of our algorithm degrades.
- Partition a pattern to get subpatterns with smaller *k'* values.
- Quickly find all those substrings in a target, which match a subpattern with smaller *k'* differences.
- For each surviving substring, recheck it to see whether it is an occurrence of the pattern, but with *k* differences.

Pattern Partition

- **Two-step method: Filtering and Exact matching**

  Filtering

  In the first step, we partition the pattern $x = x_1 \ldots x_m$ evenly into l segments, denoted as $x = P_1 P_2 \ldots P_l$ with each $P_i = x_{(i-1)r+1} \ldots X_{ir}$ for $1 \leq l \leq l - 1$, and $P_l = x_{(l-1)r+1} \ldots x_m$, where $r = \lceil m/l \rceil$. Then, we check each P_i against $BWT(_y)$ with $k' = \lfloor k/l \rfloor$ differences in turn to find all the occurrences of P_i $(1 \leq i \leq l)$ in y. Each occurrence is represented by (i, j), indicating that P_i matches a segment ending at position j in y with k' differences.

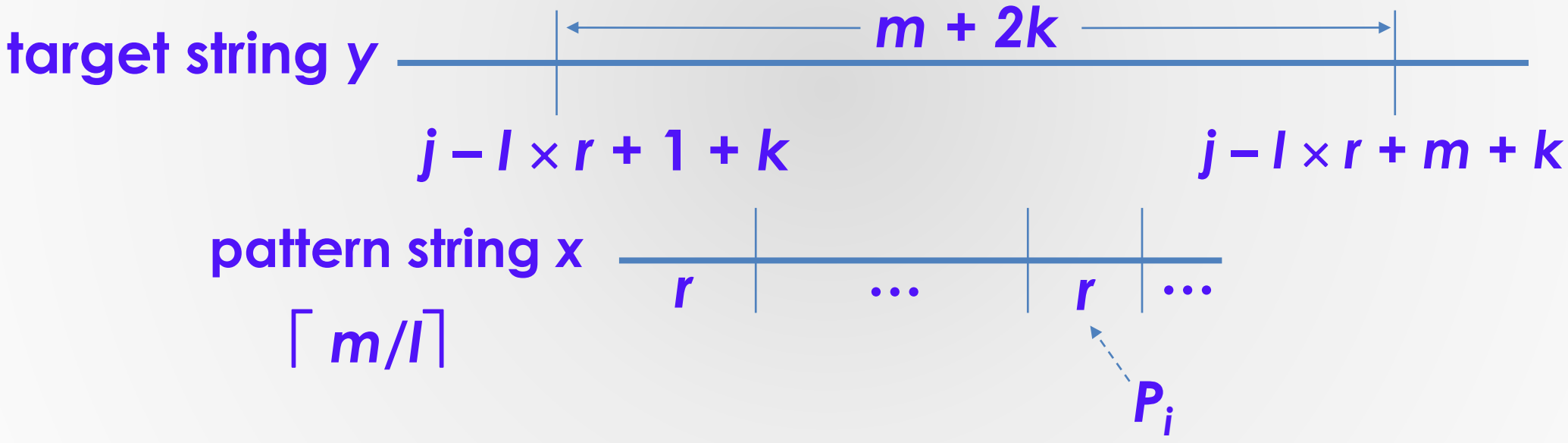
Pattern Partition

- **Two-step method**
- **Exact checking**

  In the second step, for each occurrence (i, j) found in the first step, the substring of the target: $s_{i,j} = y_{j-ir+1-k} \cdots y_{j-ir+1+m+k}$ will be again closely checked against x with k differences by using a classical algorithm. The length of $s_{i,j}$ is $m + 2k$.

Pattern Partition

- Illustration for pattern partition



Experiments

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

1. Ukkonen's onlline method (u-o for short, [57]),
2. Chang-Lawer's first method (ch-1 for short, [14]),
3. Chang-Lawer's second method (ch-2 for short, [14]),
4. Ukkonen's index-based method (u-i for short, [58]),
5. Myers's index-based method (m-i for short, [44]),
6. Peri-Culpepper's index-based method (pc-i for short, [49]), and
7. ours, discussed in this paper.

Experiments

- **Test bed**

1. All codes are written in C++ and compiled by GNU g++ compiler version 5.4.0 with compiler option `-O2'.
2. All tests run on a 64-bit Ubuntu OS with a single core of Intel Xeon E5-2637 @3.50Ghz. The system memory is of 64 GB.

Experiments

- **Test bed**

1. For time measurements, we used the Unix time commands. In addition, the suffix trees for patterns (in the Chang-Lawler's method and ours), as well as for reference sequences (in the Ukkonen's index-based method) are constructed by using the algorithm described in [59].
2. To construct the suffix arrays and the BWT-arrays, we used a code found in the libdivsufsort library (https://github.com/y256/libdivsufsort)

Experiments

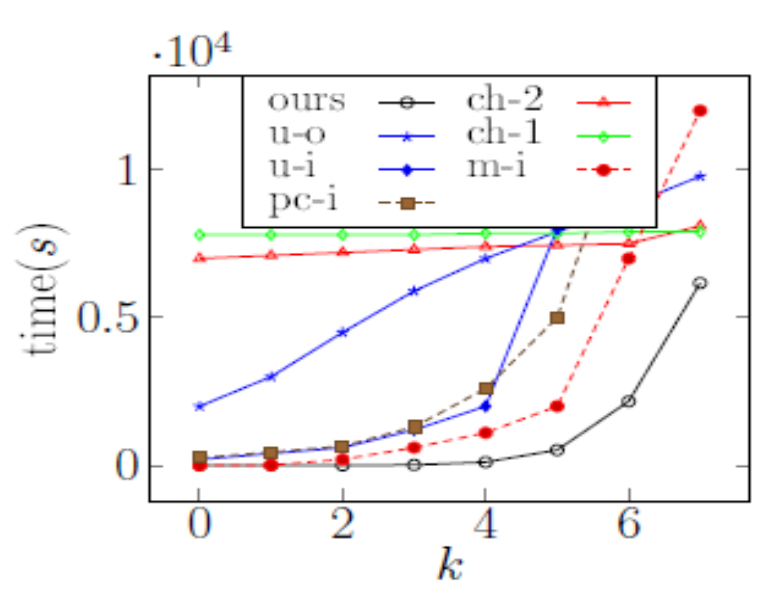
- **Data**

Reference sequences*	Num. characters	Time (s) for building BWT(y)
Gorilla	3063,403,506	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

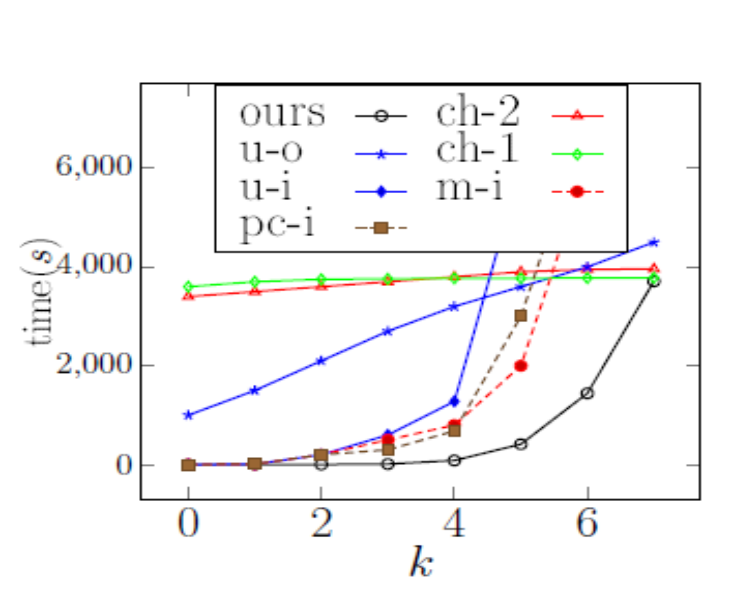
*The first three are genome sequences while the last two are protein sequences.

Experiments

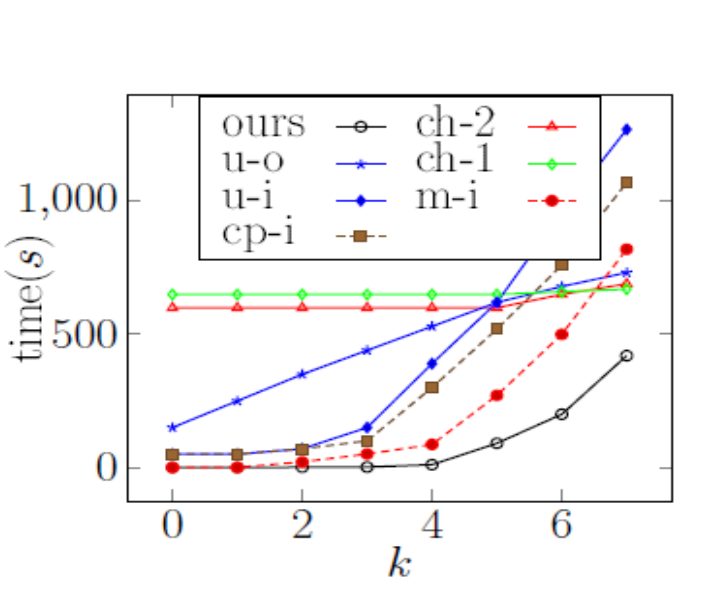
- Experiments on the string matching with small number of differences
  - Pattern length = 100 characters



Gorilla genome



ZebraFish genome



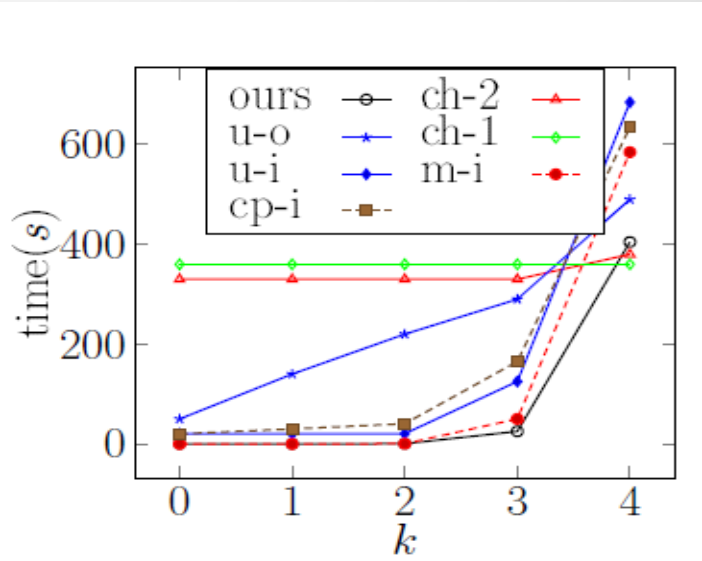
Gorilla Chr1

Experiments

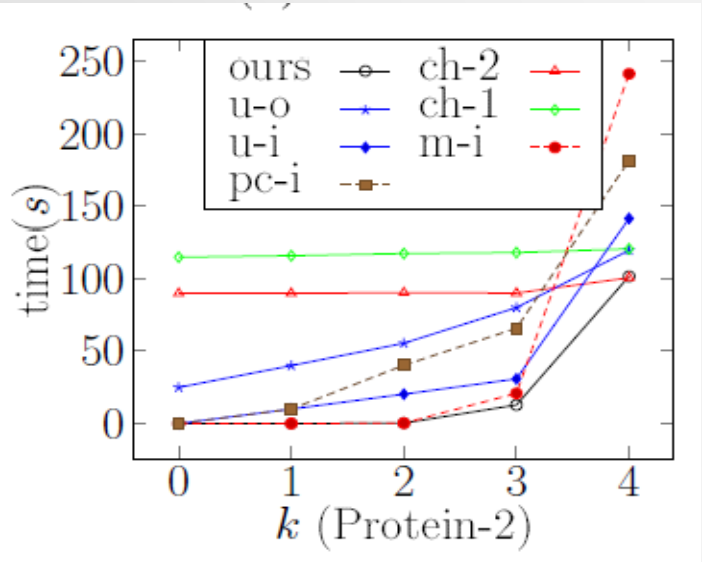
Number of nodes in *T* (Gorilla)

k	1	2	3	4	5	6	7
|*T*|	1.4k	25k	278.5k	2M	10M	39.72M	92M

Experiments



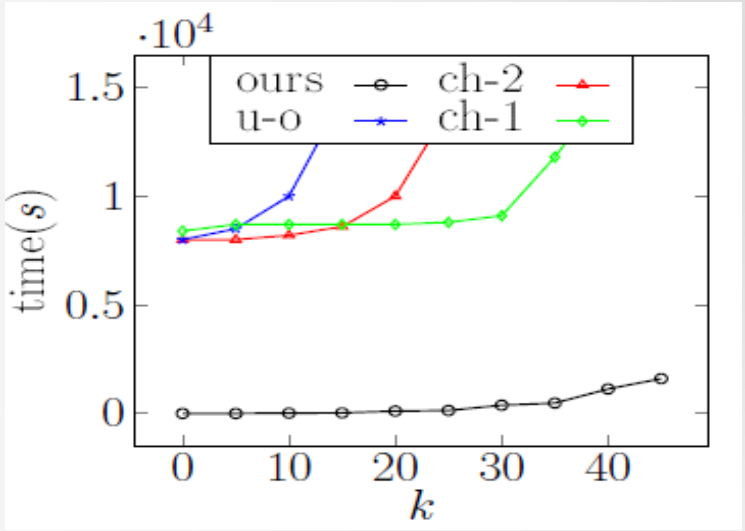
Protein 1



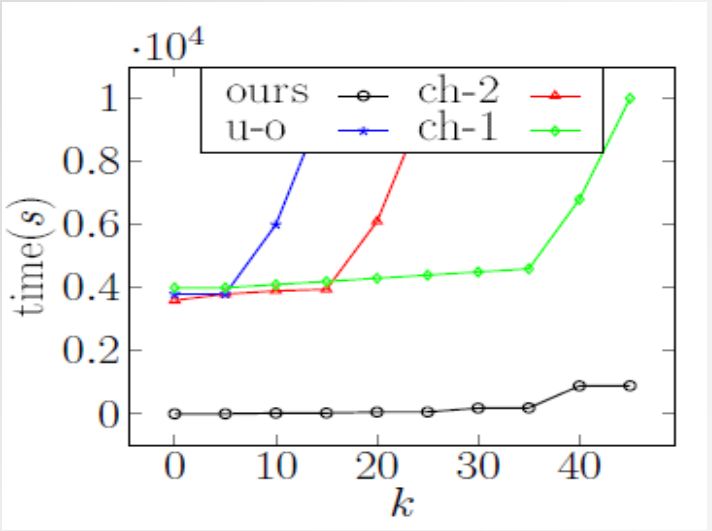
Protein 2

Experiments

- Experiments on the string matching with large number of differences (for which the two-step method is used.)
  - Pattern length = 300 characters



Gorilla genome



ZebraFish genome

Experiments

Two-step execution details on Gorilla genome

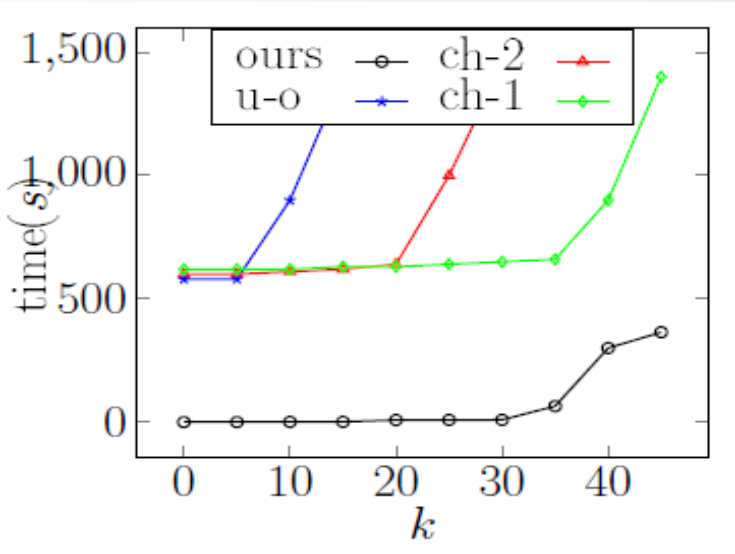
k	20	25	30	35	40	45
Step-1	23.1	23.1	173.2	172.9	1187.0	1187.5
Step-2	97.41	122.1	263.4	311.7	492.32	565.58
num. of surviving segments	30.5k	30.5k	52.9k	52.7k	69.5k	69.3k
Size of a segment	353	364	388	399	428	439

Two-step execution details on ZebraFish genome

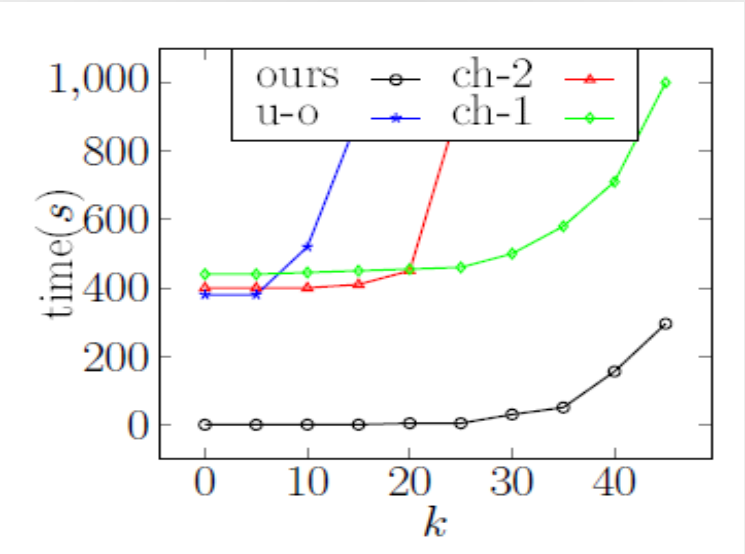
k	20	25	30	35	40	45
Step-1	58.73	58.57	187.5	187.6	953.0	952.4
Step-2	18.41	21.48	32.24	38.85	60.33	60.70
num. of surviving segments	3638	3633	4709	4702	6376	6365
Size of a segment	423	423	444	455	463	474

Experiments

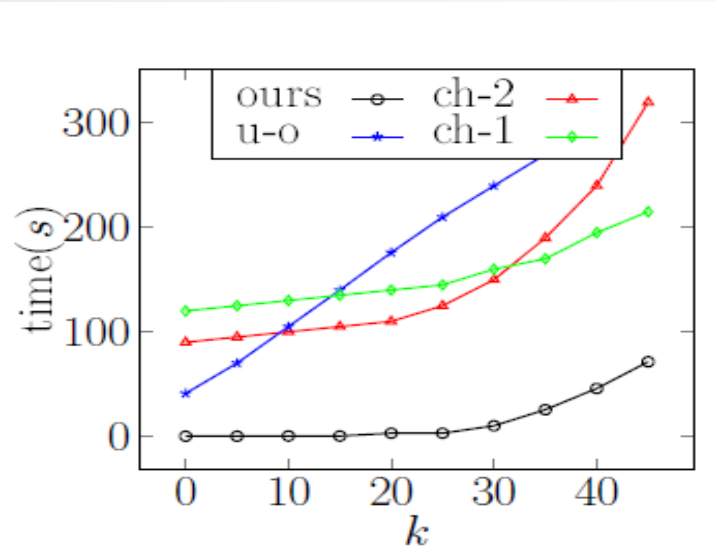
- Experiments on the string matching with large number of differences (for which the two-step method is used.)
  - Pattern length = 300 characters



Gorilla Chr1



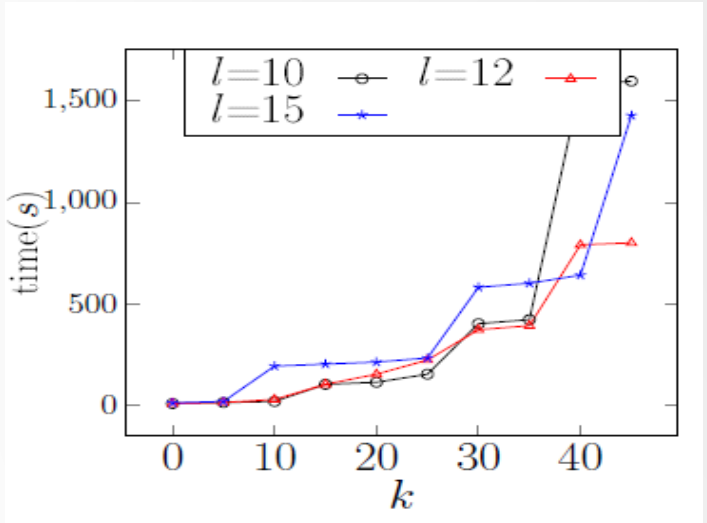
Protein 1



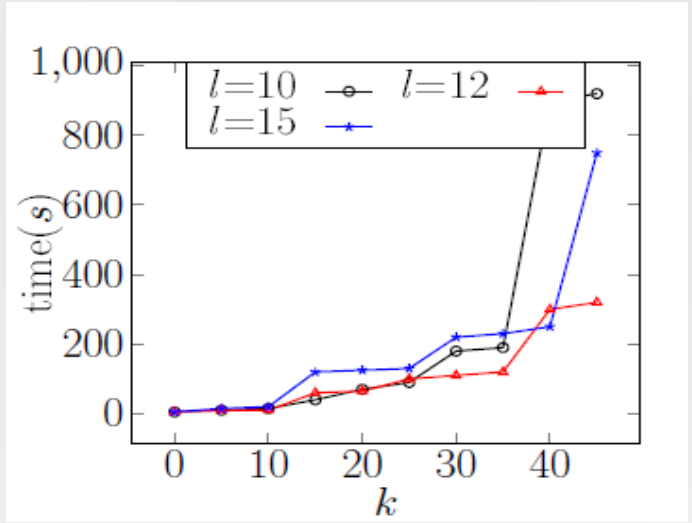
Protein 2

Experiments

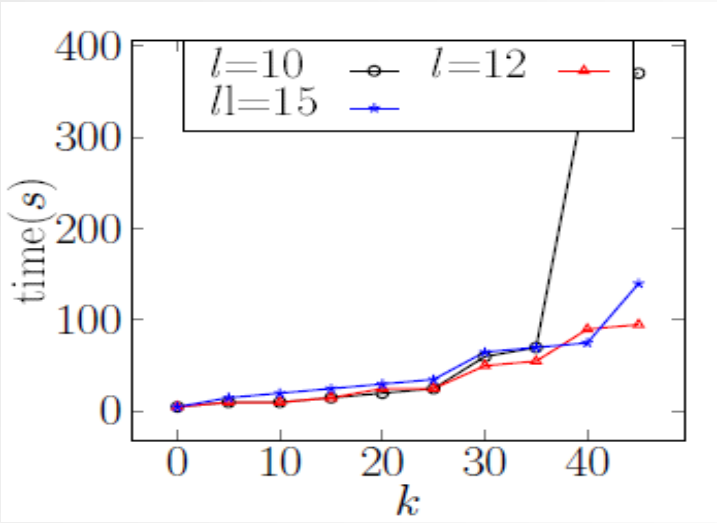
- Experiments on number of subpatterns
  - Pattern length = 300 characters



Gorilla genome



ZebraFish genome



Gorilla Chr1

l: the number of subpatterns

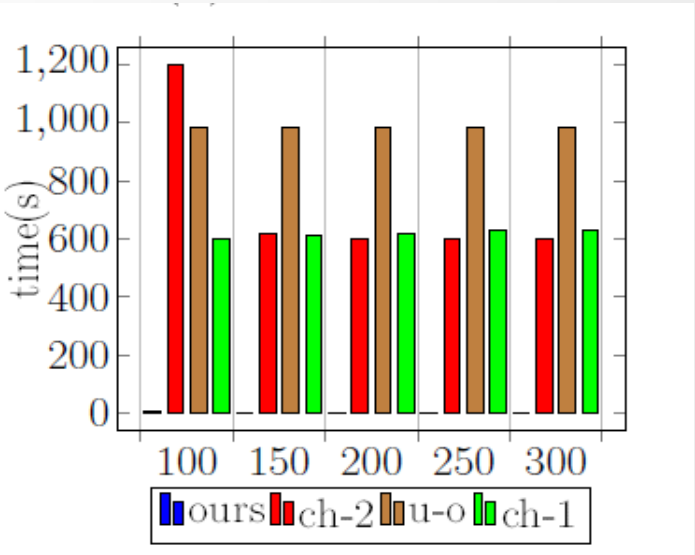
Experiments

Number of segments checked in Step-2

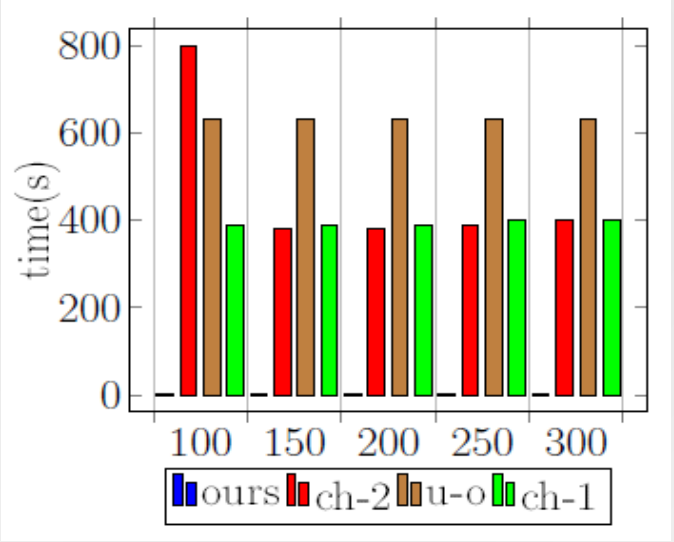
K	20	25	30	35	40	45
l =10	30.5k	30.5k	52.9k	52.7k	69.5k	69k
l = 12	28.7k	56.2k	56.0k	55.8k	60.5k	61.4k
l = 15	30.5k	30.5k	52.9k	52.7k	69.5k	69.3k

Experiments

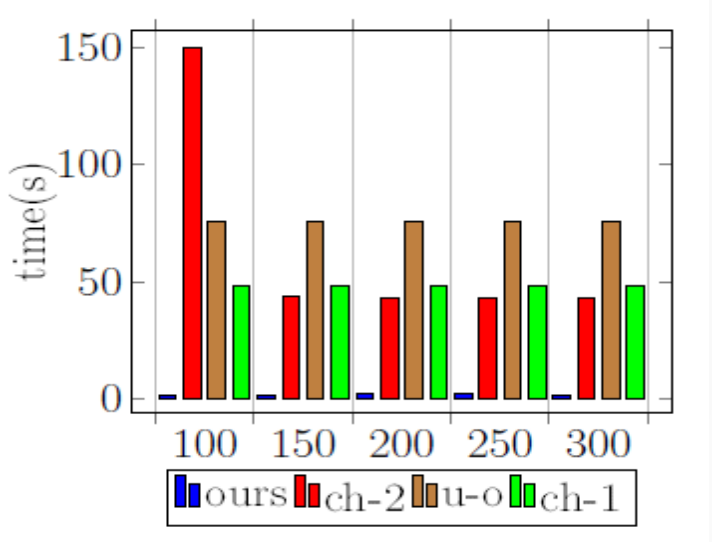
- Experiments on length of patterns



Gorilla genome



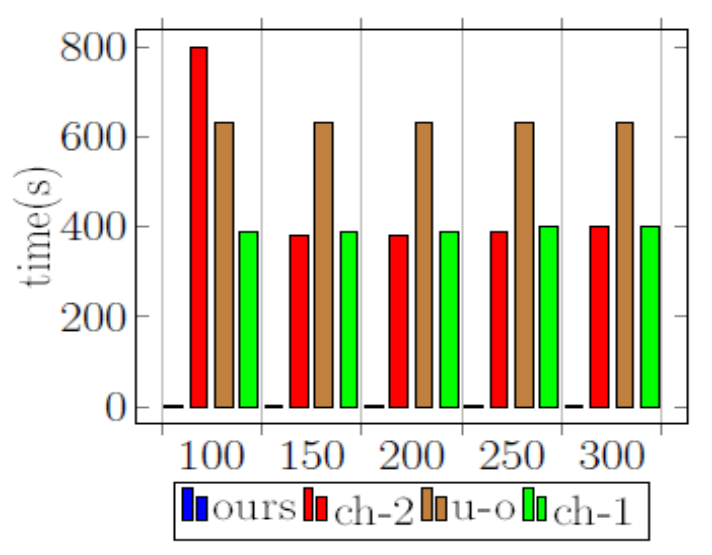
ZebraFish genome



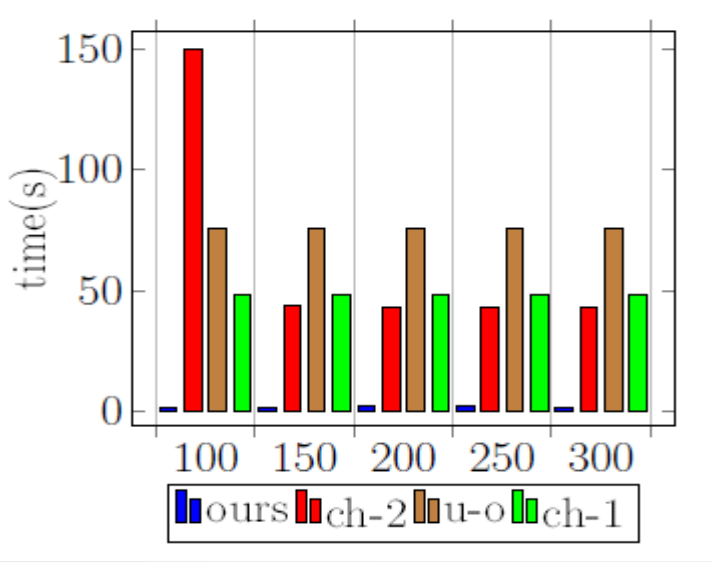
Gorilla Chr1

Experiments

- **Experiments on length of patterns**



Protein 1



Protein 2

Conclusion

- Main contribution
  - An algorithm for the string matching with *k* differences
    - Combination of dynamic programming and BWT indexes for the problem of string matching with *k* difference
    - Concept of search trees and two branch cutting methods
  - Extensive tests
- Future work
  - String matching with *don't care* symbols (using BWT transformation as indexes)

Thank you!