BWT Arrays and Mismatching Trees: A New Way for String Matching with *k* Mismatches

Yangjun Chen, Yujia Wu
Department of Applied Computer Science
University of Winnipeg

Outline

- Motivation
  - Statement of Problem
  - Related work
- BWT Arrays – A space-economic Index for String Matching
- String Matching with *k* Mismatches
  - Search trees
  - Mismatching information
  - Mismatching trees
- Experiments
- Conclusion and Future Work

Statement of Problem

- **String matching with *k* mismatches:** find all the occurrences of a pattern string *r* in a target string *s* with each occurrence having up to *k* positions different between *r* and *s*.
  - In DNA databases, due to polymorphisms or mutations among individuals or even sequencing errors, a read (a short sample DNA sequence) may disagree in some positions at any of its occurrences in a genome.

Example: $k = 4$

```
      a a a a a c a a a c     <---------- pattern
    a c a c a c a g a a g c c   <---------- target
        |   |   |     |
```

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, U. Vishkin; Amir at al.; Cole*)
  - String matching with *k differences* - Levelshtein distance (*Chang, Lampe*)
  - String matching with *wild-cards* (*Manber, Baeza-Yates*)

BWT-Index

- **Burrows-Wheeler Transform (*BWT*)**
- $s = a_1c_1a_2g_1a_3c_2a_4\$$

a_1	c_1	a_2	g_1	a_3	c_2	a_4	$\$$
c_1	a_2	g_1	a_3	c_2	a_4	$\$$	a_1
a_2	g_1	a_3	c_2	a_4	$\$$	a_1	c_1
g_1	a_3	c_2	a_4	$\$$	a_1	c_1	a_2
a_3	c_2	a_4	$\$$	a_1	c_1	a_2	g_1
c_2	a_4	$\$$	a_1	c_1	a_2	g_1	a_3
a_4	$\$$	a_1	c_1	a_2	g_1	a_3	c_2
$\$$	a_1	c_1	a_2	g_1	a_3	c_2	a_4

Rank correspondence:

rank: 3

rank: 3

rk_F	F							L	rk_L
-	$\$$	a_1	c_1	a_2	g_1	a_3	c_2	a_4	1
1	a_4	$\$$	a_1	c_1	a_2	g_1	a_3	c_2	1
2	a_3	c_2	a_4	$\$$	a_1	c_1	a_2	g_1	1
3	a_1	c_1	a_2	g_1	a_3	c_2	a_4	$\$$	-
4	a_2	g_1	a_3	c_2	a_4	$\$$	a_1	c_1	2
1	c_2	a_4	$\$$	a_1	c_1	a_2	g_1	a_3	2
2	c_1	a_2	g_1	a_3	c_2	a_4	$\$$	a_1	3
1	g_1	a_3	c_2	a_4	$\$$	a_1	c_1	a_2	4

$rk_F(e) = rk_L(e)$

BWT construction:

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

***SA*[...] – suffix array**

Backward Search of BWT-Index

- $s = a_1c_1a_2g_1a_3c_2a_4\$$
- Search $p = aca$

$\leftarrow$- - - *Backward Search*

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

Z: a character **π: a range in *F***

L_π: a range in *L*, corresponding to π

F	L
$\$$	a_4
a_4	c_2
a_3	g_1
a_1	$\$$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	L
$\$$	a_4
a_4	c_2
a_3	g_1
a_1	$\$$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	L
$\$$	a_4
a_4	c_2
a_3	g_1
a_1	$\$$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	L	Suffix Array
$\$$	a_4	8
a_4	c_2	7
a_3	g_1	5
a_1	$\$$	1
a_2	c_1	3
c_2	a_3	6
c_1	a_1	2
g_1	a_2	4

Backward Search of BWT-Index

search(*c*, <*a*, [2, 5]>) *search*(*a*, <*c*, [1, 2]>)

Search sequence: <*a*, [2, 5]> <*c*, [1, 2]> <*a*, [3, 4]>

F	*L*
\$	a_4
a_4	c_2
a_3	g_1
a_1	\$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	*L*
\$	a_4
a_4	c_2
a_3	g_1
a_1	\$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	*L*
\$	a_4
a_4	c_2
a_3	g_1
a_1	\$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

F	*L*	*Suffix Array*
\$	a_4	8
a_4	c_2	7
a_3	g_1	5
a_1	\$	1
a_2	c_1	3
c_2	a_3	6
c_1	a_1	2
g_1	a_2	4

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 $L[\alpha .. \beta]$ ($\alpha \le \beta$) 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 c in $L[2 .. 5]$, we only need to find $A_c[2 - 1] = A_c[1] = 0$ and $A_c[5] = 2$. So the corresponding range is $[A_c[2 - 1] + 1, A_c[5]] = [1, 2]$.

F	L
\$	a_4
a_4	c_2
a_3	g_1
a_1	\$
a_2	c_1
c_2	a_3
c_1	a_1
g_1	a_2

$A_\$$	A_a	A_c	A_g	A_t
0	1	0	0	0
0	1	1	0	0
0	1	1	1	0
1	1	1	1	0
1	1	2	1	0
1	2	2	1	0
1	3	2	1	0
1	4	2	1	0

Reduce *rankAll*-Index Size

- *F*-ranks: F_α = <a; x_a, y_a>
- 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_4	1	8
2	a_4	c_2	1	7
3	a_3	g_1	1	5
4	a_1	\$	-	1
5	a_2	c_1	2	3
6	c_2	a_3	2	6
7	c_1	a_1	3	2
8	g_1	a_2	4	4

→

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

$F_\$$ = <\$; 1, 1>
F_a = <*a*; 2, 5>
F_c = <*c*; 6, 7>
F_g = <*g*; 8, 8>

\+

L
a_4
c_2
g_1
\$
c_1
a_3
a_1
a_2

\+

$A_\$$	A_a	A_c	A_g	A_t
0	1	0	0	0
0	1	1	0	0
0	1	1	1	0
1	1	1	1	0
1	1	2	1	0
1	2	2	1	0
1	3	2	1	0
1	4	2	1	0

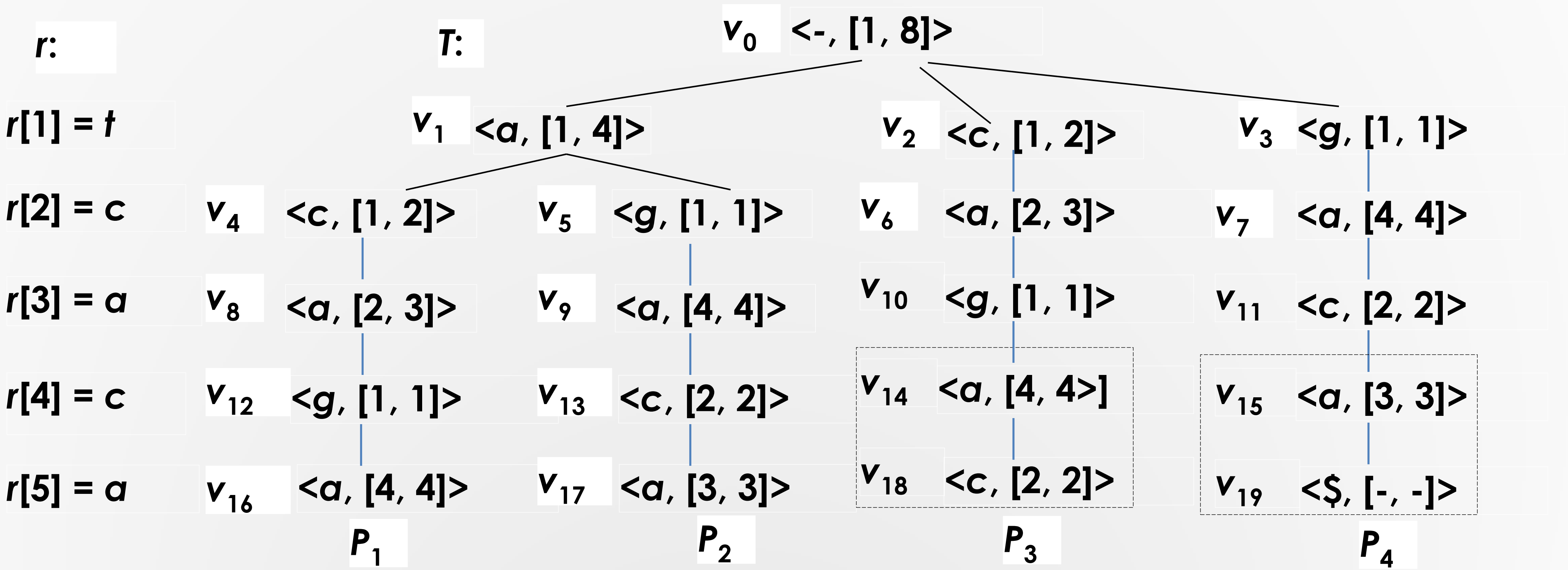
\+

*SA**
8
7
5
1
3
6
2
4

String Matching with *k* Mismatches

- ## Search Trees

pattern: *r* = *tcaca*; target: *s* = *acagaca*; *k* = 2.

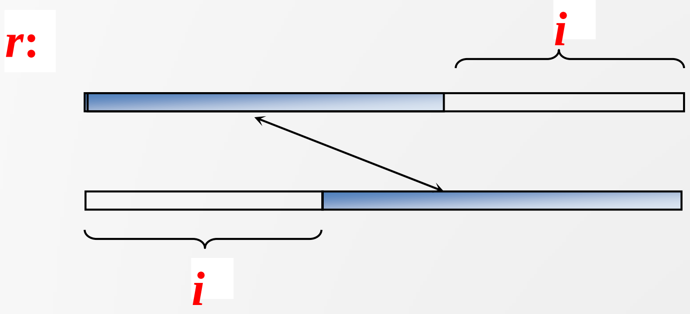


String Matching with *k* Mismatches

- **Mismatching information**

  R – mismatching table for r with $|r| = m$.

  R_{ij} – containing the positions of the first $2k + 1$ mismatches between $r[i \mathinner{..} m - q + i]$ and $r[j \mathinner{..} m - q + j]$, where $q = \max\{i, j\}$, such that if $R_{ij}[l] = x$ $(\neq \infty)$ then $r[i + x - 1] \neq r[j + x - 1]$ or one of them does not exist, and it is the l-th mismatch between them.



r_1: tcaca
r_2: cacg

r_1: tcacg
r_3: acg

r_1: tcacg
r_4: cg

r_1: tcaca
r_5: g

R_{12}:	1	2	3	4	∞
R_{13}:	1	3	∞	∞	∞
R_{14}	1	2	∞	∞	∞
R_{15}	1	∞	∞	∞	∞

String Matching with k Mismatches

- **Derivation of mismatching information**

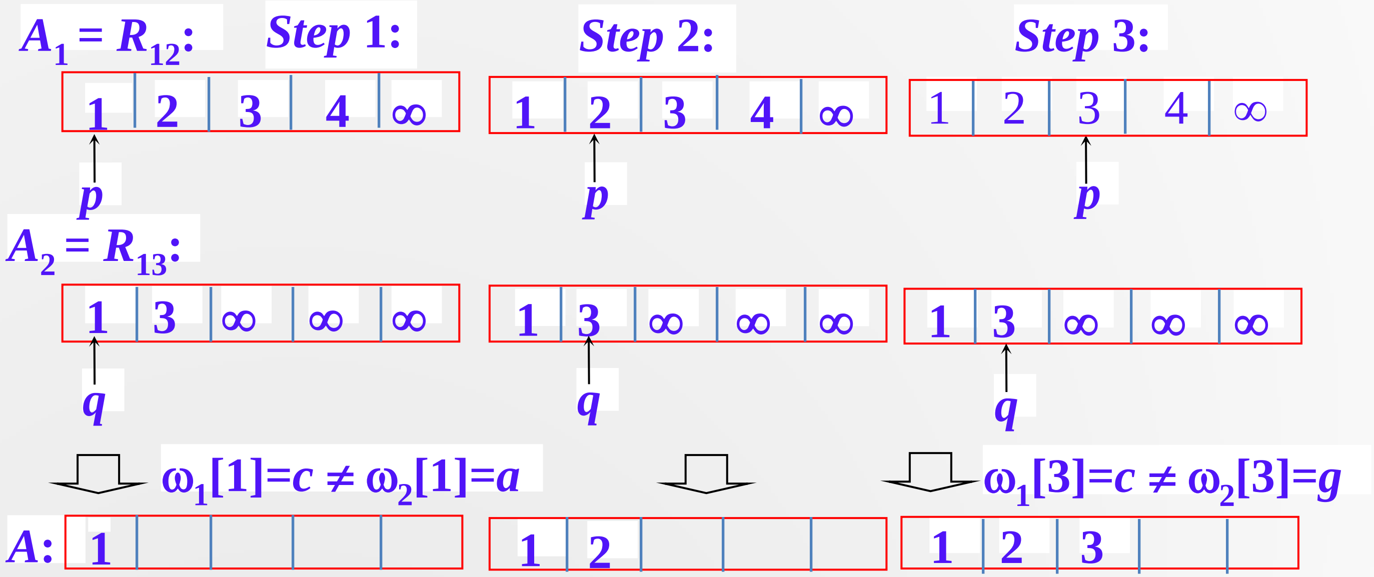
We store only part of mismatching information, specifically: R_{12}, …, R_{1m}, while all the other mismatching information will be dynamically derived.

Derive the mismatching information between

$\omega_1 = r[2 .. 4] = cacg$ and

$\omega_2 = r[3 .. 5] = acg$

from R_{12} and R_{13}.



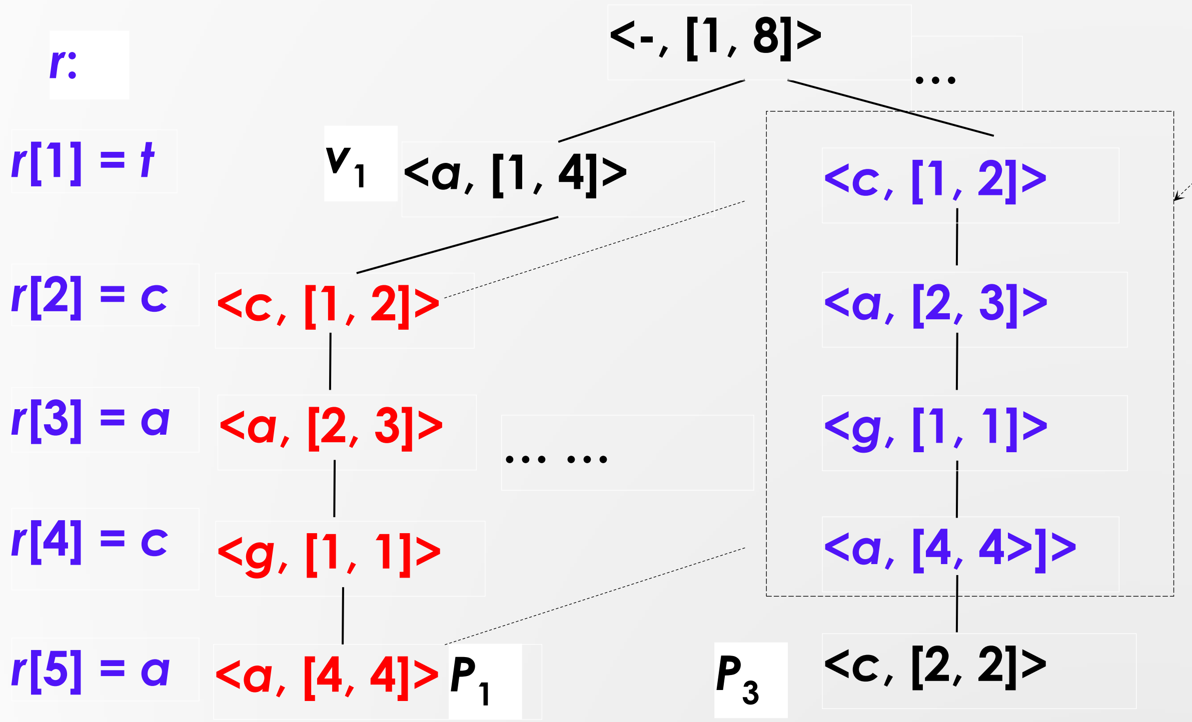
String Matching with *k* Mismatches

- **Algorithm for Derivation of mismatching information**
  - Let ω, ω_1 and ω_2 be three strings. Let A_1 and A_2 be two arrays containing all the positions of mismatches between ω and ω_1, and ω and ω_2, respectively.
  - Create a new array A such that if $A[i] = j$ ($\neq \infty$), then $\omega_1[j] \neq \omega_1[j]$, or one of them does not exists. It is the ith mismatch between them.

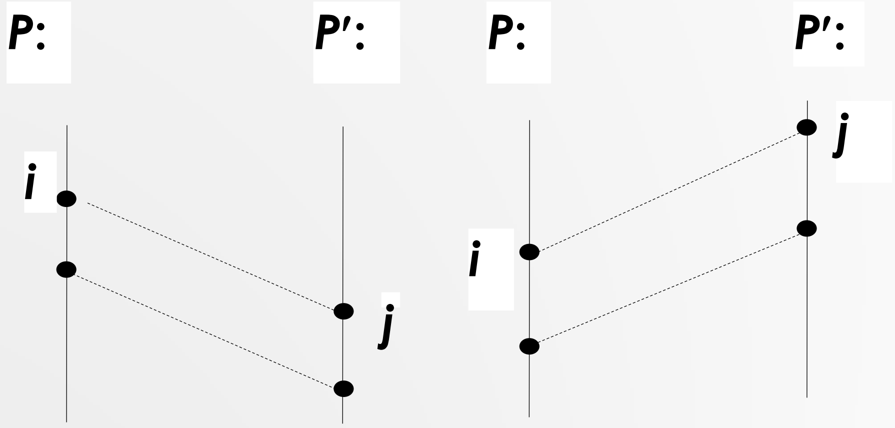
1. $p := 1$; $q := 1$; $l := 1$;
2. If $A_2[q] < A_1[p]$, then {$A[l] := A_2[q]$; $q := q + 1$; $l := l + 1$;}
3. If $A_1[p] < A_2[q]$, then {$A[l] := A_1[p]$; $p := p + 1$; $l := l + 1$;}
4. If $A_1[p] = A_2[q]$, then {if $\omega_1[p] \neq \omega_2[q]$, then {$A[l] := q$; $l := l + 1$;} $p := p + 1$; $q := q + 1$;}
5. If $p > |A_1|$, $q > |A_2|$, or both $A_1[p]$ and $A_2[q]$ are ∞, stop (if A_1 (or A_2) has some remaining elements, which are not ∞, first append them to the rear of A, and then stop.)
6. Otherwise, go to (2).

String Matching with *k* Mismatches

- Derivation of mismatching information for paths in a search tree.



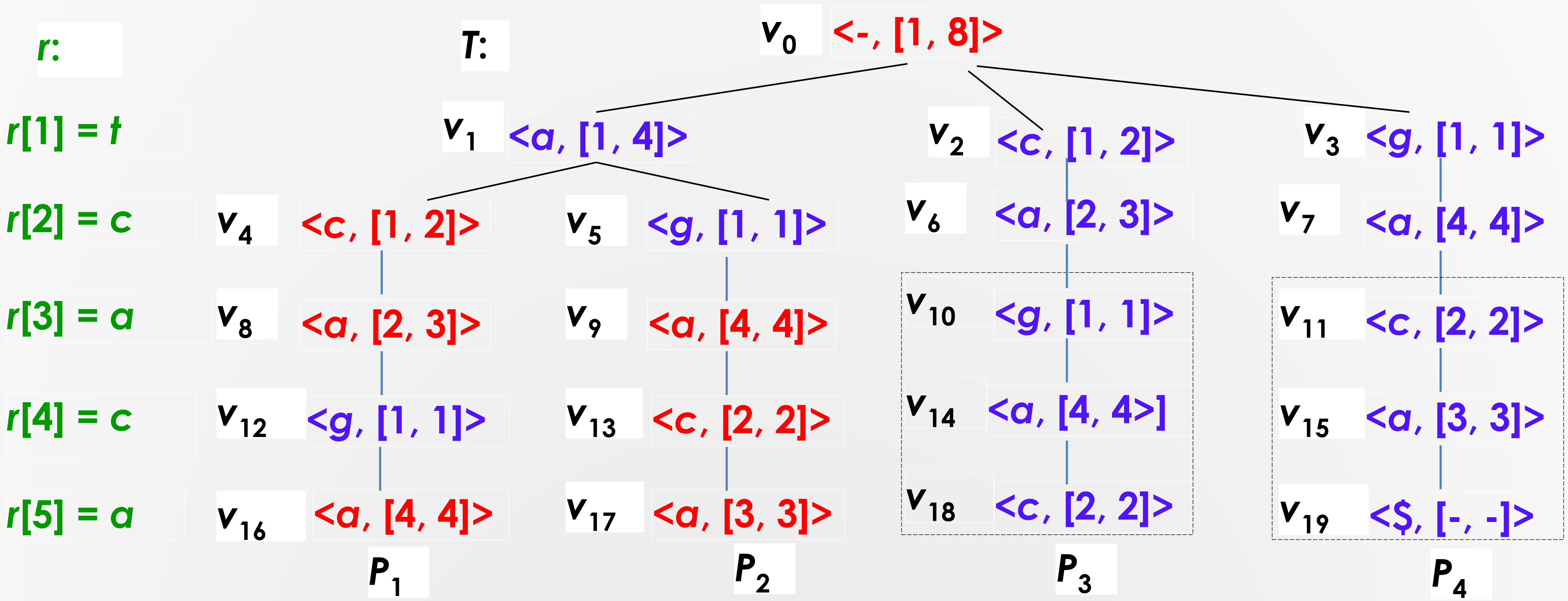
This part of P_3 will not be created. We derive the mismatching information for it according to P_1 and R_{21}.



String Matching with *k* Mismatches

- ## Mismatching trees

In a search tree, we distinguish between matching and mismatching nodes.



String Matching with *k* Mismatches

- Mismatching trees

r:

$r[1] = t$

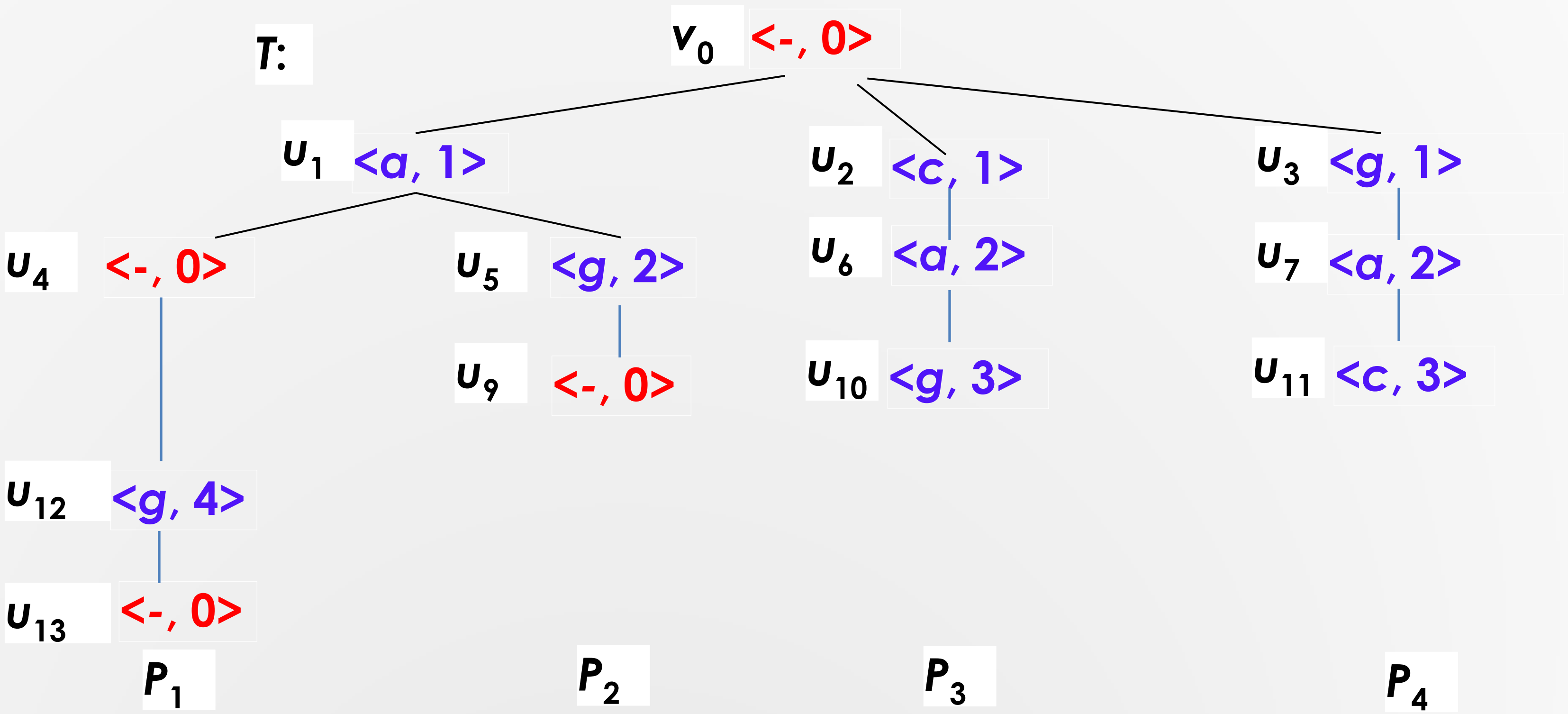
$r[2] = c$

$r[3] = a$

$r[4] = c$

$r[5] = a$

T:

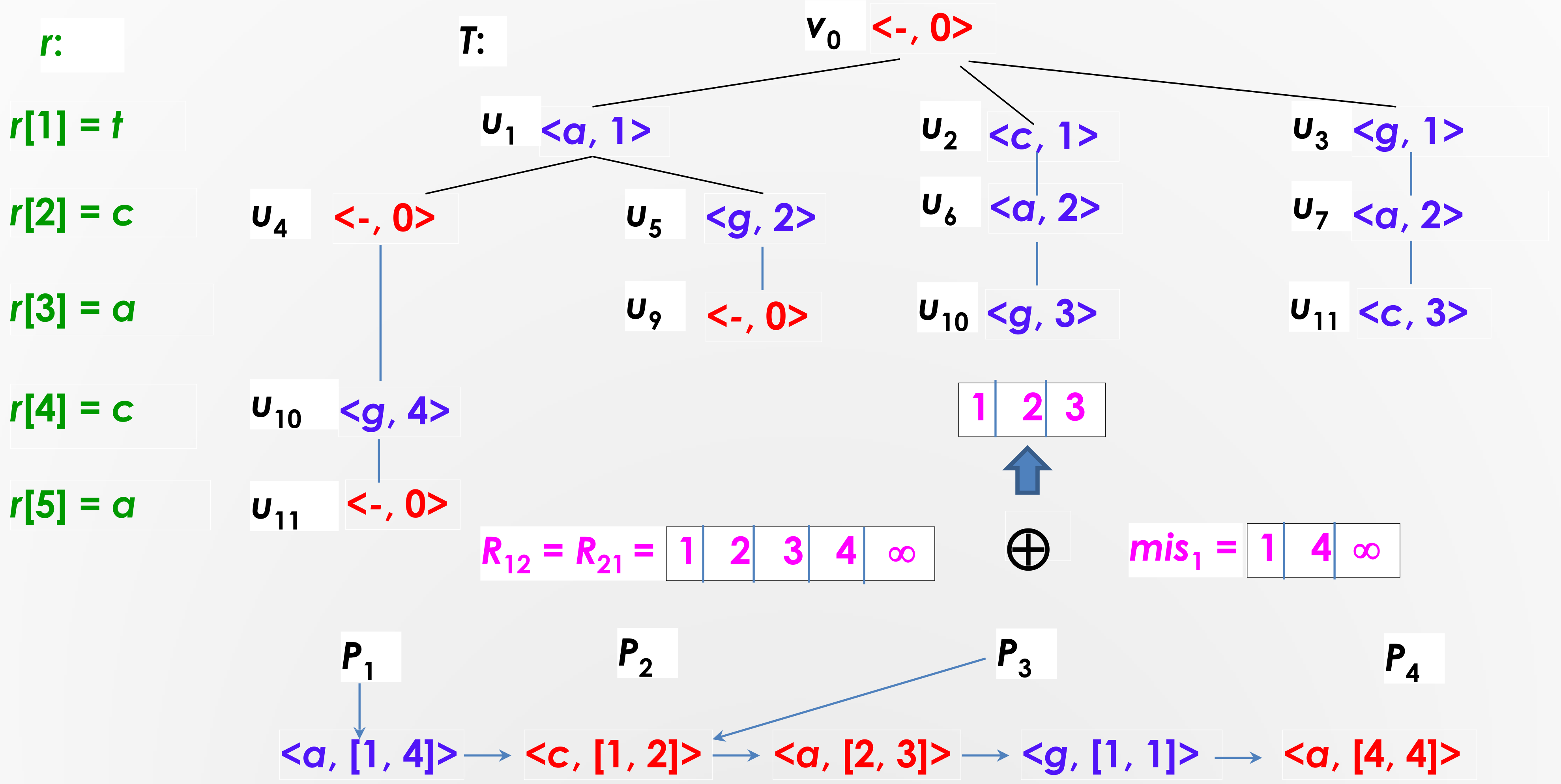


String Matching with *k* Mismatches

- Algorithm

  - Mismatching tree generation
  - Derivation of mismatching information for paths

Derivation of Mismatching Information



Algorithm

Generation of mismatching trees

- In order to generate a mismatching tree D, we will use a stack S to control the process, in which each entry is a quadruple (v, j, κ, u), where
  - v – a node inserted into the hash table.
  - j – j is an integer to indicate that v is the jth node on a path in T (counted from the root with the root as the 0th node).
  - κ – the number of mismatches between the path and $r[0 .. j]$ (recall that $r[0]$ = '-').
  - u – the parent of a node in D to be created for v.

Algorithm

Mismatching tree generation

- Each time an entry $e = (v, j, \kappa, u)$ with $v = \langle x, [\alpha, \beta]\rangle$ is popped out from S, we will check whether $x = r[j]$.
  - If $x = r[j]$, we will generate a node $u' = \langle x, j\rangle$ and link it to u as a child.
  - If $x \neq r[j]$, we will check whether u is a node of the form $\langle -, 0\rangle$. If it is not the case, generate a node $u' = \langle -, 0\rangle$.
  - Otherwise, set u' to be u.
  - Using *search*( ) to find all the children of v: $v_1, \ldots, v_l$. Then, push each $(v_i, j + 1, \kappa', u')$ into S with κ' being κ or $\kappa + 1$, depending on whether $y_i = r[j + 1]$, where $v_i = \langle y_i, [\alpha_i, \beta_i]\rangle$.

Algorithm

- **Mismatching information derivation for paths**

  - As with the basic process, each time a node $v = <x, [\alpha, \beta]>$ (compared to $r[j]$) is encountered, which is the same as a previous one $v' = <x', [\alpha', \beta']>$ (compared to $r[i]$), we will not create a subtree in T in a way as for v', but create a new node u for v in D (mismatching tree) and then go along $p(v')$ (the link associated with v') to find the corresponding nodes u' in D and search $D[u']$ in the breadth-first manner to generate a subtree rooted at u in D by simulating the merge operation discussed in Subsection *B*.
  - In other words, $D[u]$ (to be created) corresponds to the mismatch arrays for all the paths going though v in T, which will not be actually produced.

Algorithm

- **Mismatching information derivation for paths**
  - To this end, a queue data structure Q is used to do a breadth-first search of $D[u']$, and at the same time generate $D[u]$. In Q, each entry e is a pair (w, γ) with w being a node in $D[u']$, and γ an entry in R_{ij}. Initially, put $(u', R_{ij}[1])$ into Q, where $u' = <x, i>$. In the process, when e is dequeued from Q (taken out from the front), we will make the following operations (simulating the steps in *merge*( )):

1) Let $e = (w, R_{ij}[l])$. Assume that $w = <z, f>$ and $R_{ij}[l] = val$. If $<z, f> = <\text{-}, 0>$, then create a copy of w added to $D[u]$. If w is not a leaf node, let $w_1, \ldots, w_h$ be the children of w and enqueue $(w_1, R_{ij}[l]), \ldots, (w_h, R_{ij}[l])$ into Q (append at the end) in turn. If $<z, f> \neq <\text{-}, 0>$, do (2), (3), or (4).
2) If $f < i + val - 1$, add $<z, f - i + j>$ to $D[u]$. If w is not a leaf node, enqueue $(w_1, R_{ij}[l]), \ldots, (w_h, R_{ij}[l])$ into Q.

Algorithm

❑ Mismatching information derivation for paths

3) If $f = i + val - 1$, we will distinguish between two subcases: $z \neq r[j + val - 1]$ and $z = r[j + val - 1]$. If $z \neq r[j + val - 1]$, we have a mismatching and add a node $<z, j + val - 1>$ to $D[u]$. If $z = r[j + val - 1]$, create a node $<-, 0>$ added to $D[u]$. (If its parent is $<-, 0>$, it should be merged into its parent.)

4) If w is not a leaf node, enqueue $<w_1, R_{ij}[l + 1]), \ldots, < w_h, R_{ij}[l + 1])$ into Q.

5) If $f > i + val - 1$, we will scan R_{ij} starting from $R_{ij}[l]$ until we meet $R_{ij}[l']$ such that $f \leq i + R_{ij}[l'] - 1$. For each $R_{ij}[g]$ ($l \leq g < l'$), we create a new node $<r[j + R_{ij}[g] - 1], j + R_{ij}[g] - 1>$ added to $D[u]$. Enqueue $<w, R_{ij}[l']>$ into Q.

Experiments

- **Compare 4 different approaches**
  - *BWT-based* [34] (BWT for short),
  - *Amir's method* [2] (Amir for short),
  - *Cole's method* [14] (Cole for short),
  - *Algorithm A discussed in this paper* (*A*( ) for short)

Experiments

Test Environments:

- Implementation in C++, compiled by GNU make utility with optimization of level 2
- 64-bit Ubuntu operating system
- run on a single core of a 2.40GHz Intel Xeon E5-2630 processor with 32GB RAM

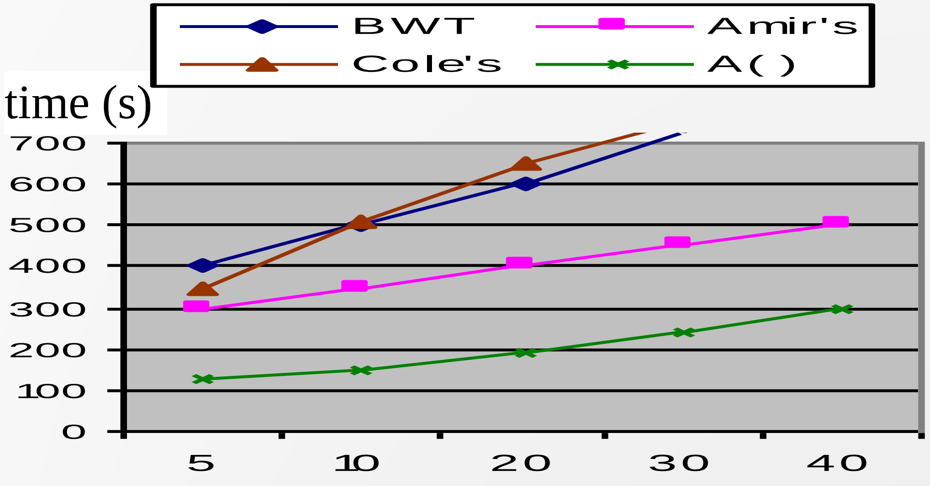
Experiments

TABLE I. CHARACTERISTICS OF GENOMES

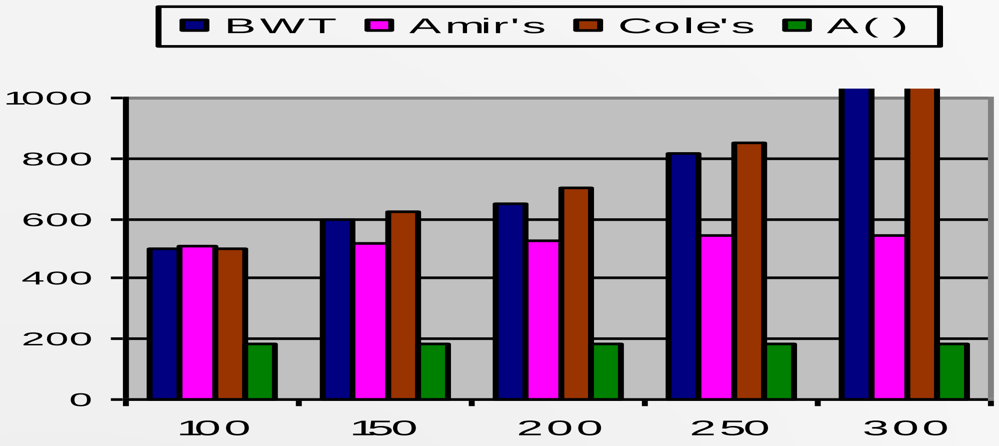
Genomes	Genome sizes (bp)
Rat chr1 (Rnor_6.0)	290,094,217
C. merolae (ASM9120v1)	16,728,967
C. elegans (WBcel235)	103,022,290
Zebra fish (GRCz10)	1,464,443,456
Rat (Rnor_6.0)	2,909,701,677

Tests with Real Data

➢ Tests with varying length of reads (over Rat genome)



varying values of *k*



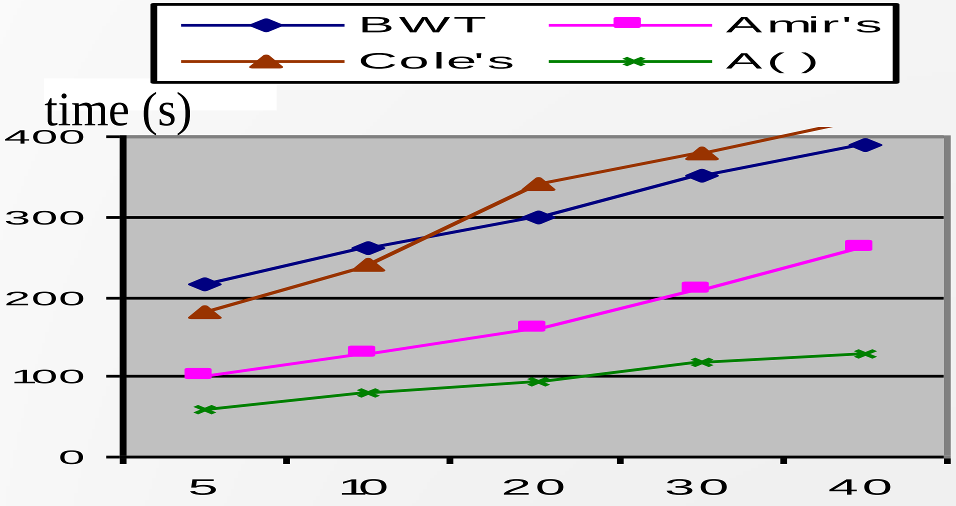
varying length of reads

Number of leaf nodes of *S*-trees

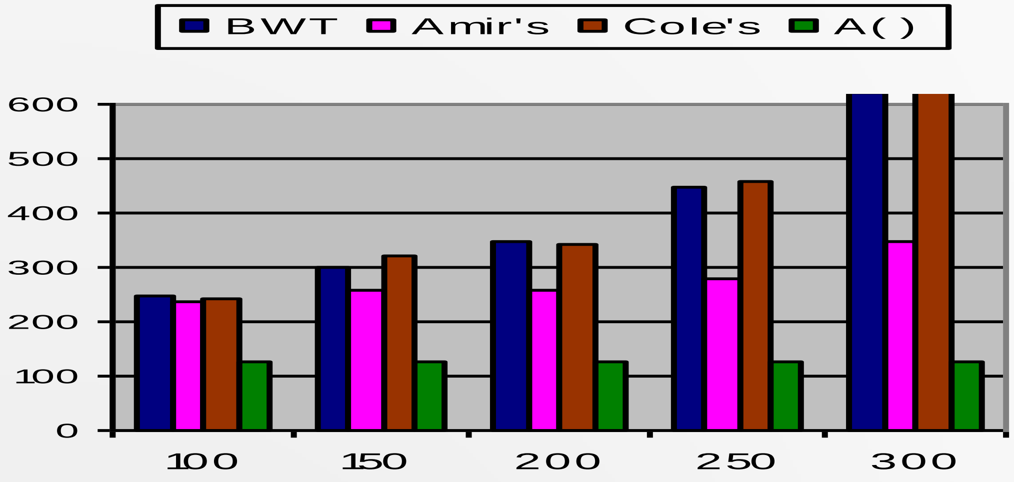
k/length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	2K	0.7M	16.5M	102M

Tests with Real Data

➢ TESTS WITH VARYING LENGTH OF READS (OVER Zebra fish)



varying values of *k*



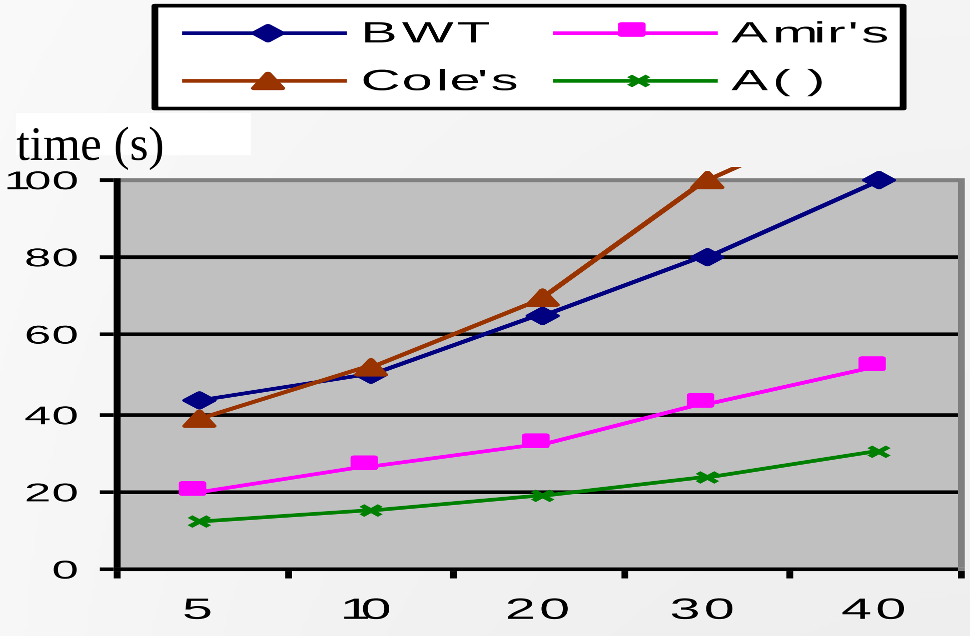
varying length of reads

Number of leaf nodes of *S*-trees

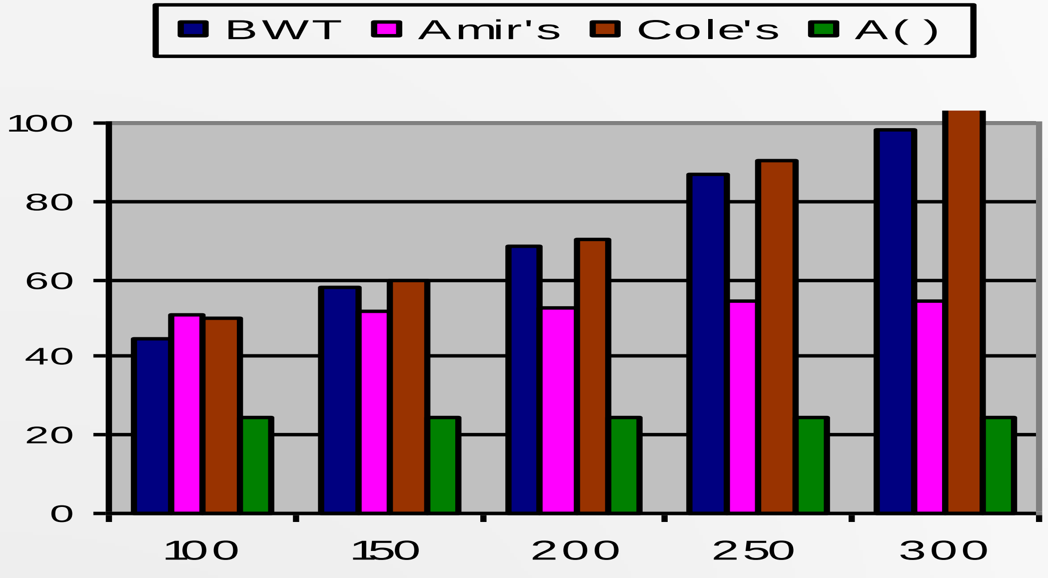
k/length-of-read	5/50	10/100	20/150	30/200
No. of leaf nodes	0.7K	0.30M	9.2M	89M

Tests with real Data

➢ Tests with varying length of reads (OVER Rat chr1)



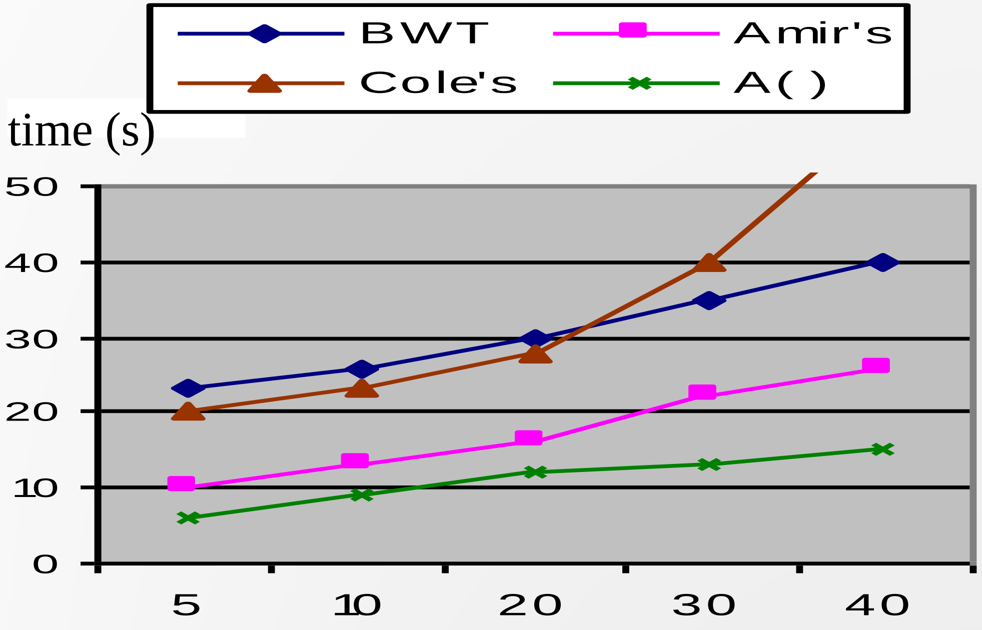
varying values of *k*



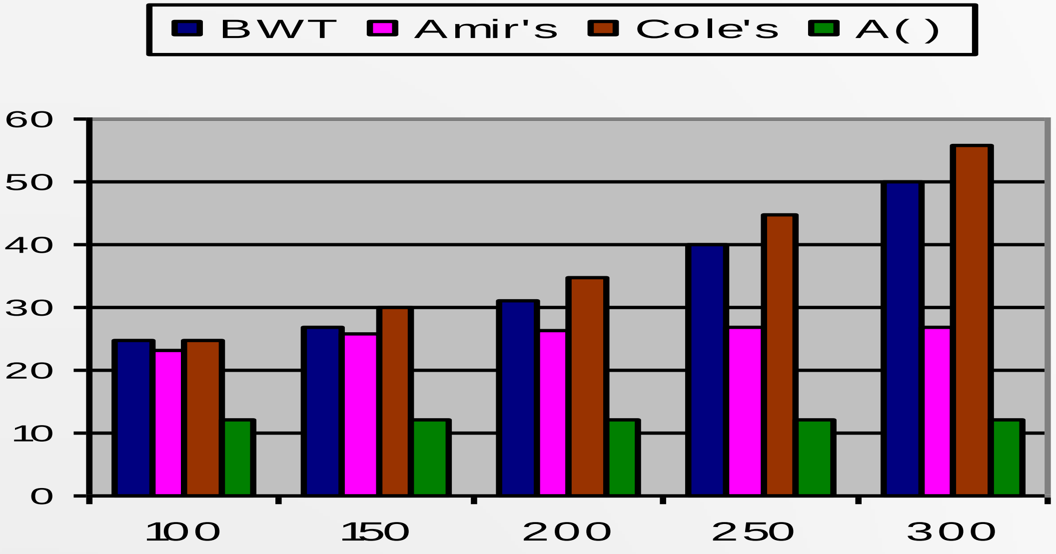
varying length of reads

Tests with Real Data

- Tests with varying length of reads (OVER C. elegans)



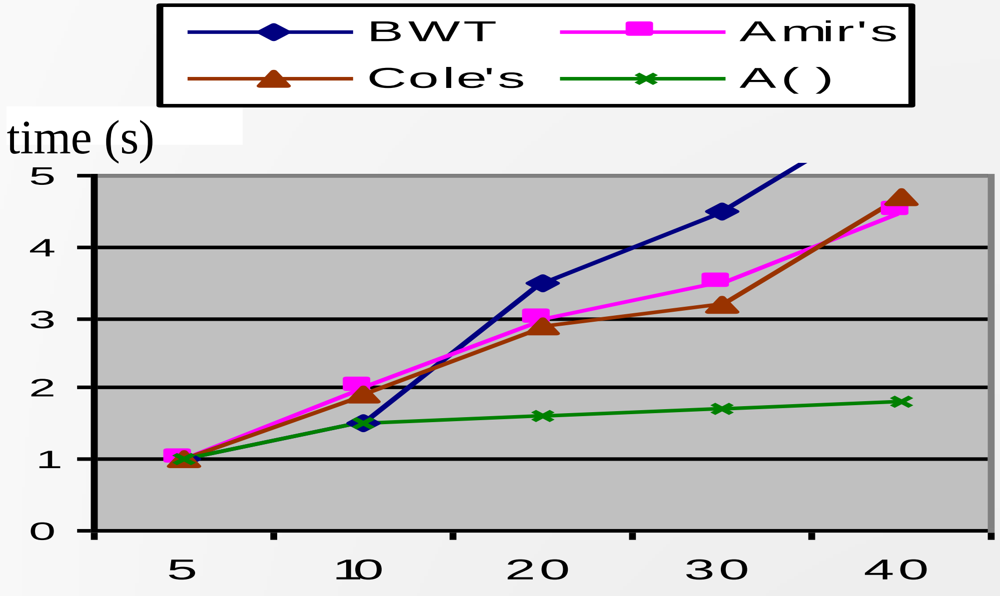
varying values of k



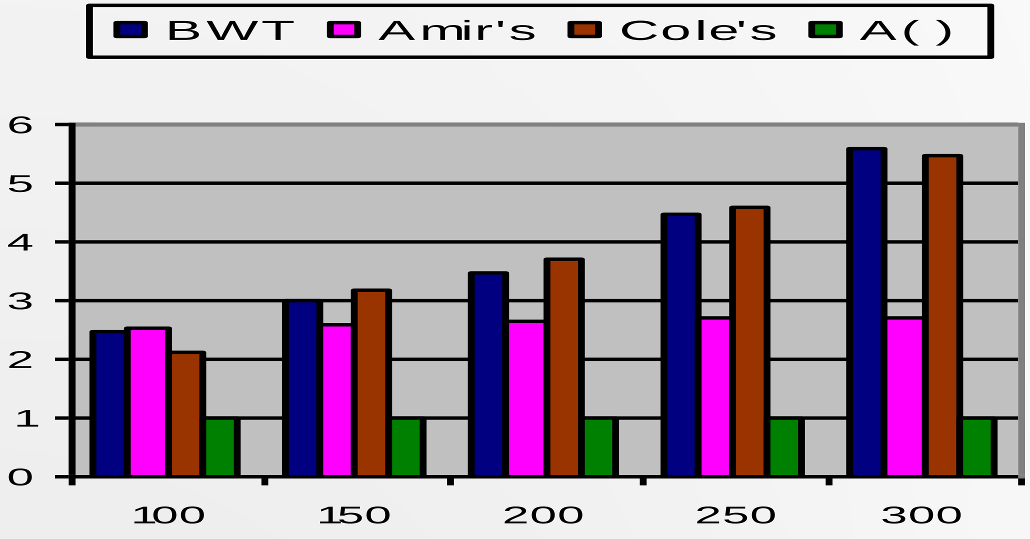
varying length of reads

Tests with Real Data

- Tests with varying length of reads (over C. merlae)



varying values of *k*



varying length of reads

Conclusion and Future Work

- **Main contribution**
  - Combination of derivation of mismatching information and BWT indexes for *k* mismatching problem
  - Concept of mismatching trees
  - Extensive tests
- **Future work**
  - String matching with *k* differences
  - String matching with *don't care* symbols

Thank you!