Pattern matching

Eric Rivals

LIRMM, CNRS & U. Montpellier

November 29, 2021

Outline

1. Introduction
2. From simple to complex: Different types of patterns
3. Exact pattern matching: a primer
4. Set Pattern Matching: searching for several motifs
   - Example of a tree indexing motifs
   - Off-line search: indexing the text for optimal search time
5. PM using bit parallelism
   - Principle
   - Algorithme Shift-OR
6. Probability of occurrence
7. Approximate Pattern Matching

Pattern matching

- Important and classical question in computer science
- In practice it arises in many application contexts bioinformatics, text processing, databases, etc.
- Different formulations depending on whether one searches for
  1. one or several words
  2. exactly or approximately
  3. regular expression
  4. set of similar words

Different algorithmic approaches to exact pattern matching

- Automaton based

  e.g., [Aho Corasick 75]

- Window scan algorithms

  e.g., [Horspool 80]

- Algorithms using bit parallelism

  e.g., [Baeza-Yates Perleberg 86]

- Fingerprints

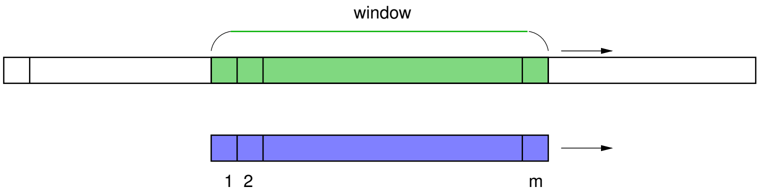
  e.g., [Karp Rabin 87]

From simple to complex: Different types of patterns

Exact Pattern Matching

1. a *text T* of length *n*
2. a *pattern M* of length *m*, and generally $m << n$.

For single word: window scan algorithm



Exact Pattern Matching

1. a *text* T of length n

2. a *pattern* M of length m, and generally $m << n$.

Example: $M := tgtg$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g

Exact Pattern Matching

1. a *text* T of length n
2. a *pattern* M of length m, and generally $m << n$.

Example: $M := tgtg$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										

Exact Pattern Matching

1. a *text* T of length n
2. a *pattern* M of length m, and generally $m << n$.

Example: $M := tgtg$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								

Exact Pattern Matching

1. a *text* T of length n
2. a *pattern* M of length m, and generally $m << n$.

Example: $M := tgtg$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
												t	g	t	g

Solution: $\{2, 4, 12\}$

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T := ctgtgtgtacatgtg$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			
											a	t	g		

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			
											a	t	g		
												t	g	t	g

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T := \mathit{ctgtgtgtacatgtg}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			
											a	t	g		
												t	g	t	g

Solutions:
M_1 at positions $\{2, 4, 12\}$

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T := ctgtgtgtacatgtg$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			
											a	t	g		
												t	g	t	g

Solutions:
M_1 at positions $\{2, 4, 12\}$
M_2 at position $\{11\}$

Exact Set Pattern Matching

Search simultaneously for occurrences of a set of words in a text.

Input: a set $\mathcal{M} := \{\text{tgtg}, \text{atg}, \text{cat}\}$ of words and a text $T :=$ *ctgtgtgtacatgtg*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
		t	g	t	g										
				t	g	t	g								
										c	a	t			
											a	t	g		
												t	g	t	g

Solutions:
M_1 at positions $\{2, 4, 12\}$
M_2 at position $\{11\}$
M_3 at position $\{10\}$

Approximate Pattern Matching

1. Idem: a *text T* of length *n*, a *pattern M* of length *m*

2. a maximum number of allowed differences.

Example: $M := tgtg$ **One mismatch allowed (at most)**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g

Approximate Pattern Matching

1. Idem: a *text* T of length n, a *pattern* M of length m
2. a maximum number of allowed differences.

Example: $M := tgtg$ **One mismatch allowed (at most)**

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T:	c	t	g	t	g	t	g	t	a	c	a	t	g	t	g
						t	g	t	g						

Additional occurrence with one mismatch at pos. 6 in T
mismatch at pos. 4 in M
Solution: $\{2, 4, 6, 12\}$

Probabilistic motif search: Position Weight Matrix (PWM)

Definition: Position Weight Matrix (PWM)

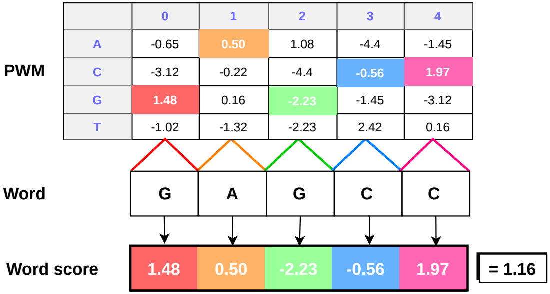
For DNA, a PWM M is $4 \times m$ matrix:

- for $\alpha \in \Sigma$ and position i, entry $M[\alpha, i] :=$ the score of nuc. α at position i
- The score of a word is the sum of scores at all positions.

Problem:

Given a text T, a PWM M, a score threshold s, find all substrings of T whose score is $> s$.

Scoring of a word
(or a substring of T)



PWM

	0	1	2	3	4
A	-0.65	0.50	1.08	-4.4	-1.45
C	-3.12	-0.22	-4.4	-0.56	1.97
G	1.48	0.16	-2.23	-1.45	-3.12
T	-1.02	-1.32	-2.23	2.42	0.16

Searching for probabilistic motifs

PWMs are the simplest model

More complex motifs exist

Searching for probabilistic motifs

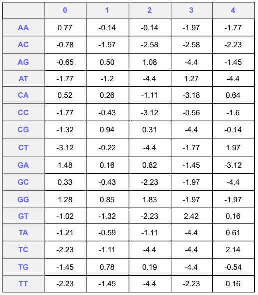
PWMs are the simplest model

More complex motifs exist

dinucleotidic PWMs

HOCOMOCO database
for Transcription Factors

di-PWM



	0	1	2	3	4
AA	0.77	-0.14	-0.14	-1.97	-1.77
AC	-0.78	-1.97	-2.58	-2.58	-2.23
AG	-0.65	0.50	1.08	-4.4	-1.45
AT	-1.77	-1.2	-4.4	1.27	-4.4
CA	0.52	0.26	-1.11	-3.18	0.64
CC	-1.77	-0.43	-3.12	-0.56	-1.6
CG	-1.32	0.94	0.31	-4.4	-0.14
CT	-3.12	-0.22	-4.4	-1.77	1.97
GA	1.48	0.16	0.82	-1.45	-3.12
GC	0.33	-0.43	-2.23	-1.97	-4.4
GG	1.28	0.85	1.83	-1.97	-1.97
GT	-1.02	-1.32	-2.23	2.42	0.16
TA	-1.21	-0.59	-1.11	-4.4	0.61
TC	-2.23	-1.11	-4.4	-4.4	2.14
TG	-1.45	0.78	0.19	-4.4	-0.54
TT	-2.23	-1.45	-4.4	-2.23	0.16

Searching for probabilistic motifs

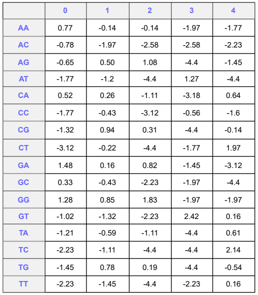
PWMs are the simplest model

More complex motifs exist

dinucleotidic PWMs

HOCOMOCO database
for Transcription Factors

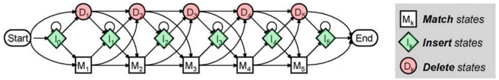
di-PWM



	0	1	2	3	4
AA	0.77	-0.14	-0.14	-1.97	-1.77
AC	-0.78	-1.97	-2.58	-2.58	-2.23
AG	-0.65	0.50	1.08	-4.4	-1.45
AT	-1.77	-1.2	-4.4	1.27	-4.4
CA	0.52	0.26	-1.11	-3.18	0.64
CC	-1.77	-0.43	-3.12	-0.56	-1.6
CG	-1.32	0.94	0.31	-4.4	-0.14
CT	-3.12	-0.22	-4.4	-1.77	1.97
GA	1.48	0.16	0.82	-1.45	-3.12
GC	0.33	-0.43	-2.23	-1.97	-4.4
GG	1.28	0.85	1.83	-1.97	-1.97
GT	-1.02	-1.32	-2.23	2.42	0.16
TA	-1.21	-0.59	-1.11	-4.4	0.61
TC	-2.23	-1.11	-4.4	-4.4	2.14
TG	-1.45	0.78	0.19	-4.4	-0.54
TT	-2.23	-1.45	-4.4	-2.23	0.16

profile Hidden
Markov Models (HMMs)
PFAM database

Profile-HMM


Start
End
D_1 D_2 D_3 D_4 D_5
I_0 I_1 I_2 I_3 I_4 I_5
M_1 M_2 M_3 M_4 M_5
M_k Match states
I_k Insert states
D_k Delete states

Searching for probabilistic motifs

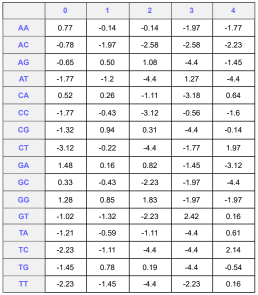
PWMs are the simplest model

More complex motifs exist

dinucleotidic PWMs

HOCOMOCO database
for Transcription Factors

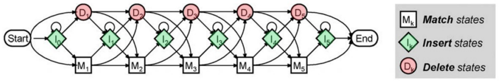
di-PWM



	0	1	2	3	4
AA	0.77	-0.14	-0.14	-1.97	-1.77
AC	-0.78	-1.97	-2.58	-2.58	-2.23
AG	-0.65	0.50	1.08	-4.4	-1.45
AT	-1.77	-1.2	-4.4	1.27	-4.4
CA	0.52	0.26	-1.11	-3.18	0.64
CC	-1.77	-0.43	-3.12	-0.56	-1.6
CG	-1.32	0.94	0.31	-4.4	-0.14
CT	-3.12	-0.22	-4.4	-1.77	1.97
GA	1.48	0.16	0.82	-1.45	-3.12
GC	0.33	-0.43	-2.23	-1.97	-4.4
GG	1.28	0.85	1.83	-1.97	-1.97
GT	-1.02	-1.32	-2.23	2.42	0.16
TA	-1.21	-0.59	-1.11	-4.4	0.61
TC	-2.23	-1.11	-4.4	-4.4	2.14
TG	-1.45	0.78	0.19	-4.4	-0.54
TT	-2.23	-1.45	-4.4	-2.23	0.16

profile Hidden
Markov Models (HMMs)
PFAM database

Profile-HMM


Start
End
D_1 D_2 D_3 D_4 D_5
I_0 I_1 I_2 I_3 I_4 I_5
M_1 M_2 M_3 M_4 M_5
M_k Match states
I_k Insert states
D_k Delete states

Other motif representations: gapped motifs, Covariance Matrix [Durbin et al. 98].

Exact pattern matching: a primer

Outline

1. The problem
2. Naive algorithm
3. Linear time algorithms
4. Text indexing approach
5. Filtration approach

Exact Pattern Matching

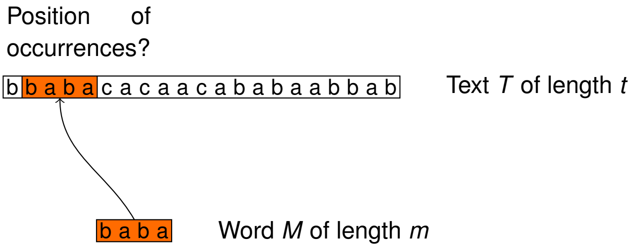
$\boxed{\text{b b a b a c a c a a c a b a b a a b b a b}}$ Text T of length t

Exact Pattern Matching

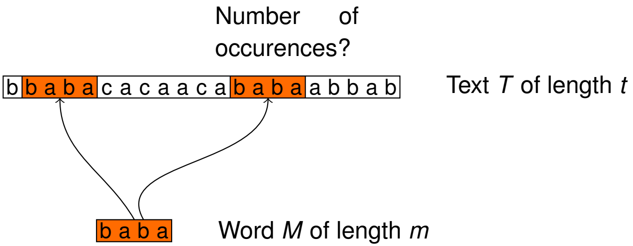
b b a b a c a c a a c a b a b a a b b a b   Text T of length t

b a b a   Word M of length m

Exact Pattern Matching



Exact Pattern Matching



Naive and involved algorithms

- Naive algorithm:
  for each window m pairwise symbol comparisons
  about n windows

  Total time proportional to $n * m$ (complexity)

- Linear time solutions:
  Idea: exploit results on a window to ease that of overlapping windows
  Boyer-Moore or Knuth Morris Pratt algorithms in the 70's

  Total time proportional to $n + m$

Naive and involved algorithms

- Naive algorithm:
  for each window m pairwise symbol comparisons
  about n windows

  Total time proportional to $n * m$ (complexity)

- Linear time solutions:
  Idea: exploit results on a window to ease that of overlapping windows
  Boyer-Moore or Knuth Morris Pratt algorithms in the 70's

  Total time proportional to $n + m$

Limitations: single query and exact match

Naive and involved algorithms

- Naive algorithm:
  for each window m pairwise symbol comparisons
  about n windows

  Total time proportional to $n * m$ (complexity)

- Linear time solutions:
  Idea: exploit results on a window to ease that of overlapping windows
  Boyer-Moore or Knuth Morris Pratt algorithms in the 70's

  Total time proportional to $n + m$

Limitations: single query and exact match

Answers: indexing text and filtration approaches

Naive algorithm : scan and shift

- basic operation: char. equality test $O(1)$
- compare M to a window of size m testing for char equality
- repeat for all $(n - m)$ possible windows
- time complexity $O((n - m) \times m)$
- *scanning* direction in T: usually left to right
- *verification* of current window: left to right
- *shift*: distance between two windows considered successively
- naive algorithm: shifts equal one, which is minimum

Forward search: Morris Pratt

- [Morris, Pratt, 76]
- both scanning and verification from left to right
- *safe shift*: a shift that do not skip over potentially valid windows
- verification: prefix wise, from left to right
- scanning left to right
- shift use borders of prefixes of M
- Later improved by Knuth into Knuth-Morris-Pratt algorithm [Knuth, Morris, Pratt, 77].

Morris-Pratt Algorithm

Algorithme 1 : Morris-Pratt

```
**Input :** Text $T$ of length $n$ and pattern $M$ of length $m$
Precompute the Shift table of $M$;
$i := 1; j := 1;$
while $(i < n - m)$ do
    $j := 0;$
    while $(j < m)$ et $(M[j] = T[i + j])$ do
        $j := j + 1;$
    if $(j = m)$ then
        print (occurrence of $M$ at position $i$);
    $i := i + Shift[j];$
    $j := max(0, j - Shift[j]);$
```

Backward search: Horspool

- Simplification of Boyer Moore algorithm
  Verification from right to left – [Horspool, 80]
- uses a single rule for shifts: improved *Bad Character rule*
  considers only the last symbol in the current window

Complexity - efficiency

- Boyer Moore worst case time complexity is $O(n + m)$
- Horspool: worst case $O(m \times n)$!
- However efficient in practice: sublinear expected running time

Horspool Algorithm

i: offset for index in pattern and in text, *j*: index text current window

Algorithme 2 : Horspool

```
Input : Text $T$ of length $n$ and pattern $M$ of length $m$
Precompute $L$ the shift table of $M$;
$j := 0$;
while $(j < n - m)$ do
    $i := m - 1$;
    while $(i > 0)$ et $(M[i+1] = T[j+i+1])$ do
        $i := i - 1$;
    if $(i \leq 0)$ then
        print (occurrence of $M$ at position $j+1$);
    $j := j + m - L[T[j+m]]$;
```

Horspool preprocessing

Shift table L for pattern M
for a symbol c: gives the position of the rightmost occurrence of c in $M[1;m-1]$

Algorithme 3 : Horspool preprocessing

```
Input : Pattern $M$ of length $m$
Output: returns table $L$ of length $\sigma$
Precompute $L$ the shift table of $M$;
for $c \in \Sigma$ do
    $L[c] := 0$;
for $i := 1..m-1$ do
    $L[M[i]] := i$;
```

Set Pattern Matching: searching for several motifs

Multiple motifs search

Given

- $\mathcal{M} = \{M_1, \ldots, M_z\}$ a set of z words of length $m_1, \ldots, m_z$,
- T a text of length n.

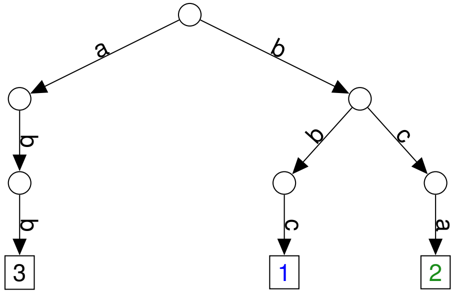
Question: find all occurrences of each motif in T.

- **Aho-Corasick** (AC) algorithm: uses a tree to represent the motifs
- **Motifs trie**: each branch of the tree spells out one motif
  Failure links: links a suffix of a window to the largest prefix of some M_i
  preprocessing is linear in the sum of the motif lengths.
- Naive Algorithm: scan the text T using the tree without failure links
  takes quadratic time.
- AC algorithm scans the text T only once, takes linear time $O(n)$. [Aho Corasick 75]

Aho Corasick automaton: example 1

Let $\mathcal{M} = \{abb, bbc, bca\}$

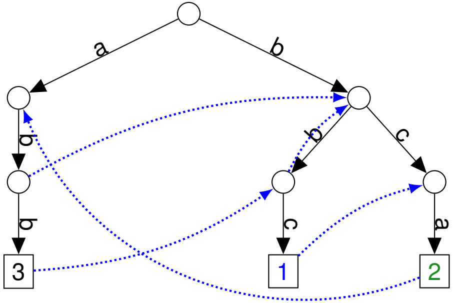
Trie of words in $\mathcal{M}$
with failure links as dotted blue lines.



Aho Corasick automaton: example 1

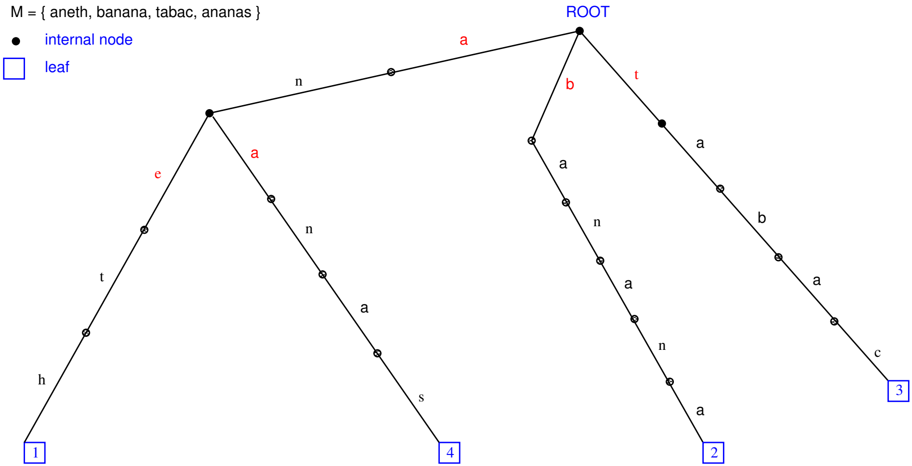
Let $\mathcal{M} = \{abb, bbc, bca\}$

Trie of words in $\mathcal{M}$
with failure links as dotted blue lines.



Example 2

M = { aneth, banana, tabac, ananas }



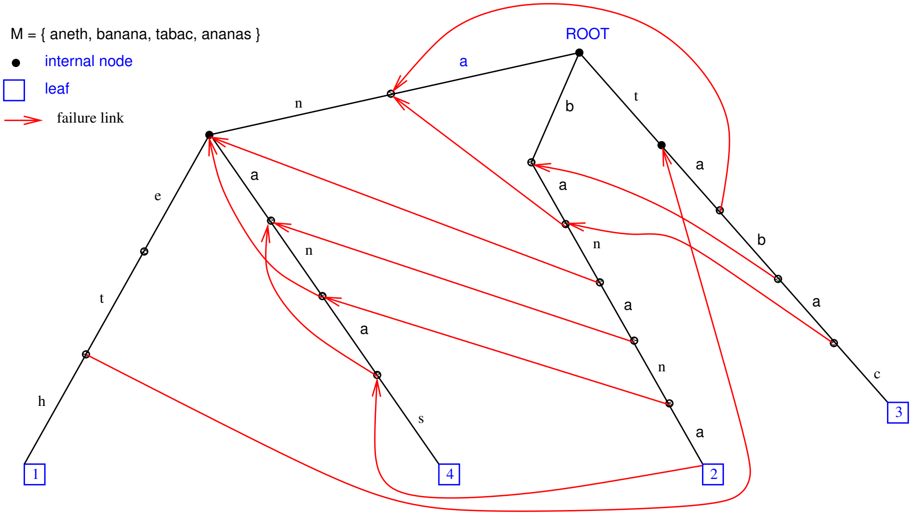
Example 2

M = { aneth, banana, tabac, ananas }

● internal node

□ leaf

failure link



Off-line search: indexing the text for optimal search time

Iterated PM on an indexed text

Matching in two steps:

1. preprocessing the text T in time $O(n)$
   build and store a data structure: an index
   enables exact search query

2. search for each pattern in the index in $O(m)$ time (optimal)

Text indexing data structures

For a text of length n, a **good** index:

1. occupancy memory in $O(n)$
2. construction time in $O(n)$ units
3. enables exact motif search in $O(m)$ time for a motif of length m

Three historical structures:

1. **compact suffix tree** [Wiener 73, McCreight 76, Ukkonen 92]
2. **suffix array**: construction in $O(n)$ [Manber & Myers 90, Kärkkäinen & Sanders 03]
3. **DAWG** (Directed Acyclic Word Graph) [Blumer et al. 85]

Breakthrough in text indexing

With historical index structures,

1. you need the text and the index
2. both in main memory to keep it fast

Around 2000, the advent of compressible “self indexing structures”:

1. a self-index replaces the text and the classical index
2. its size can be modulated in function of available memory.

Example

1. Burrows-Wheeler Transform or FM-index [Ferragina Manzini 00]
2. Enhanced Suffix Arrays [Ohlebusch, 13]
3. Various compressed *k*-mer indexes

Used in other contexts: overlap graphs, de Bruijn Graphs construction.

PM using bit parallelism

PM using state array and binary operations

- Often $|M| \ll |T|$ and M fits into a machine word
- safely shifting the window requires knowing which prefixes of M match the current window
- store this information in a state array
- shift is performed using binary operation

Avantage: fast

binary operations $\Rightarrow$ time efficiency

Example

Shift-OR [Baeza-Yates & Perleberg, 96]
Variation of Shift-And algorithm [Baeza-Yates & Gonnet, 92]

Binary operations

- unsigned integers (int): as binary vectors with 16 or 32 bits
- unsigned short int: 16 bits
- unsigned long int: 32 bits
- binary encoding from right to left

Examples of binary encodings

value	encoding on 16 bits
0	0000000000000000
1	0000000000000001
2	0000000000000010
3	0000000000000011
4	0000000000000100
2^{i-1}	1 at i^{th} position from right

Binary operations

- $\ll$: shift to the left
- $\gg$: shift to the right
- &: binary AND
- |: binary OR
- ^: binary XOR
- ~: binary complement (negation)

Example of operations

value	encoding
~0	1111111111111111
~3	1111111111111100
$1 \ll 1$	0000000000000010
$5 \gg 2$	0000000000000001

Shift-OR: principle

- a pattern M and a text T
- current window ends at position p in T
- State array E: a binary array of length m, as many as prefixes of M
- E_p encodes which prefixes of M are also suffixes of current window ending at pos. p
- if the bit corresponding to entire M equals zero $\Rightarrow$ occurrence of M
- using $T[p+1]$ and E_p, one easily computes E_{p+1}

Illustration of state array (1)

Consider

- a word $M = acat$ of length 4
- a text $T = gacat$ of length 5

T contains 2 windows of length 4: *gaca* puis *acat*

Illustration of state array (2)

The state array *E*

at position 4

Illustration of state array (2)

The state array *E*

1	2	3	4	E_4
g	a	c	a	

at position 4

Illustration of state array (2)

The state array *E*

1	2	3	4	E_4
g	a	c	a	
			a	0

at position 4

Illustration of state array (2)

The state array E

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1

Illustration of state array (2)

The state array *E*

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0

Illustration of state array (2)

The state array E

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

Illustration of state array (2)

The state array *E*

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

Illustration of state array (2)

The state array E

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

at position 4

2	3	4	5	E_5
a	c	a	t	

then at position 5

Illustration of state array (2)

The state array *E*

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

2	3	4	5	E_5
a	c	a	t	
			$\neq$	1

Illustration of state array (2)

The state array *E*

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

2	3	4	5	E_5
a	c	a	t	
			$\neq$	1
		a	$\neq$	1

Illustration of state array (2)

The state array E

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

2	3	4	5	E_5
a	c	a	t	
			$\neq$	1
		a	$\neq$	1
	$\neq$			1

Illustration of state array (2)

The state array *E*

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

2	3	4	5	E_5
a	c	a	t	
			$\neq$	1
		a	$\neq$	1
	$\neq$			1
a	c	a	t	0

Illustration of state array (2)

The state array E

at position 4

1	2	3	4	E_4
g	a	c	a	
			a	0
		$\neq$		1
	a	c	a	0
$\neq$				1

then at position 5

2	3	4	5	E_5
a	c	a	t	
			$\neq$	1
		a	$\neq$	1
	$\neq$			1
a	c	a	t	0

Question: how to compute E_5 from E_4 and $T[p+1]$?

Shift-OR principle: decomposition

Decomposing equality of prefix of M and a suffix of current window in T

Let c and p be two integers such that

- $1 < c \leq m$ and
- $0 < p \leq n$.

We get $M[1,c] = T[p-c+1,p]$

if and only if

Decomposing equality of prefix of *M* and a suffix of current window in *T*

Let c and p be two integers such that

- $1 < c \leq m$ and
- $0 < p \leq n$.

We get $M[1,c] = T[p-c+1,p]$

if and only if

$$M[1,c-1] = T[p-c+1,p-1] \text{ and } M[c] = T[p]$$

Shift-OR principle: decomposition

Decomposing equality of prefix of M and a suffix of current window in T

Let c and p be two integers such that

- $1 < c \leq m$ and
- $0 < p \leq n$.

We get $M[1,c] = T[p-c+1,p]$

if and only if

$$M[1,c-1] = T[p-c+1,p-1] \text{ and } M[c] = T[p]$$

if and only if

$$M[1,c-1] = T[p-c+1,p-1] \text{ and } T[p] \text{ matches the } c^{\text{th}} \text{ position of } M$$

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

for all c between 1 and m:

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

for all c between 1 and m:

$$L_a[c] := \begin{cases} 0 & \text{if} \quad M[c] = a \\ 1 & \text{else} \end{cases}$$

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

for all c between 1 and m:

$$L_a[c] := \begin{cases} 0 & \text{if} \quad M[c] = a \\ 1 & \text{else} \end{cases}$$

Example

$M = acat$

letter	bit mask (lightest bit on right)
a	$L_a = 1010$

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

for all c between 1 and m:

$$L_a[c] := \begin{cases} 0 & \text{if} \quad M[c] = a \\ 1 & \text{else} \end{cases}$$

Example

	letter	bit mask (lightest bit on right)
	a	$L_a = 1010$
$M = acat$	c	$L_c = 1101$

Shift-OR: bit masks

Idea: for each symbol α, a bit mask indicates the positions of α in M.

Let a a letter of Σ.

We define a bit mask L_a of m bits such that

for all c between 1 and m:

$$L_a[c] := \begin{cases} 0 & \text{if} \quad M[c] = a \\ 1 & \text{else} \end{cases}$$

Example

	letter	bit mask (lightest bit on right)
	a	$L_a = 1010$
$M = acat$	c	$L_c = 1101$
	g	$L_g = 1111$
	t	$L_t = 0111$

Shift-OR principe: final

Let E_p be the state array of m bits such that for all $j \in [1, m]$ one has:

Shift-OR principe: final

Let E_p be the state array of m bits such that
for all $j \in [1,m]$ one has:

$$E_p[j] := \begin{cases} 0 & \text{if} \quad M[1,j] = T[p-j+1,p] \\ 1 & \text{else} \end{cases}$$

Shift-OR principe: final

Let E_p be the state array of m bits such that
for all $j \in [1, m]$ one has:

$$E_p[j] := \begin{cases} 0 & \text{if} \qquad M[1,j] = T[p-j+1, p] \\ 1 & \text{else} \end{cases}$$

Thanks to the decomposition, one gets:

$$\begin{aligned} E_p[c] &= E_{p-1}[c-1] \text{ OR } L_{T[p]}[c] \quad \text{if } c > 1 \text{ AND} \\ E_p[1] &= L_{T[p]}[1] \end{aligned}$$

Shift-OR principe: final

Let E_p be the state array of m bits such that
for all $j \in [1, m]$ one has:

$$E_p[j] := \begin{cases} 0 & \text{if} \qquad M[1,j] = T[p-j+1,p] \\ 1 & \text{else} \end{cases}$$

Thanks to the decomposition, one gets:

$$\begin{aligned} E_p[c] &= E_{p-1}[c-1] \text{ OR } L_{T[p]}[c] \quad \text{if } c > 1 \text{ AND} \\ E_p[1] &= L_{T[p]}[1] \end{aligned}$$

Idea: thanks to bit arrays, one compute all bits in parallel (with binary operations)

Shift-OR principe: final

Let E_p be the state array of m bits such that for all $j \in [1, m]$ one has:

$$E_p[j] := \begin{cases} 0 & \text{if} \qquad M[1,j] = T[p-j+1,p] \\ 1 & \text{else} \end{cases}$$

Thanks to the decomposition, one gets:

$$\begin{aligned} E_p[c] &= E_{p-1}[c-1] \text{ OR } L_{T[p]}[c] \quad \text{if } c > 1 \text{ AND} \\ E_p[1] &= L_{T[p]}[1] \end{aligned}$$

Idea: thanks to bit arrays, one compute all bits in parallel (with binary operations)

$$E_p := (E_{p-1} \ll 1) \text{ OR } L_{T[p]}.$$

Shift-OR Algorithm

Algorithme 4 : Shift-OR

```
Input : Text $T$ et pattern $M$ resp. of lengths $n$ and $m$
Preprocessing: compute bit masks in $L$;
$E := \underbrace{1\dots1}_{m \text{ times}}$;
for $p$ from 1 to $n$ do
    $E := (E \ll 1)$ or $L_{T[p]}$;
    si $(E < 2^{m-1})$ alors
        report an occurrence of $M$ ending at position $p$ in $T$;
```

Probability of occurrence

Word Autocorrelations

Consider all words of length q: the set $\Sigma^q := \{Q_i : 0 < i \leq \sigma^q\}$.

Denote by $\Pr(Q_i \notin T)$: the absence probability of a word in a random text T.
i.e., the fact that Q_i does not occur in T.

- The probability $\Pr(Q_i \notin T)$ is not the same for all Q_i.
  It depends on the form of Q_i.
  More precisely: On the *autocorrelation* of Q_i.

- **Autocorrelation**: binary vector storing the “Periodicity” of Q_i i.e., the set of periods.

Periods

Definition (Period)

Let $Q \in \Sigma^q$ and let p be a non-negative integer with $p < q$.
Then p is a period of Q iff:

$$\forall\, 0 \leq i < n - q \ : Q[i] = Q[i+p].$$

Example

Shift:	0	1	2	3	4	5	6	7	8	9	0
Word:	A	B	R	A	C	A	D	A	B	R	A

$\vdots$ $\vdots$

Periods

Definition (Period)

Let $Q \in \Sigma^q$ and let p be a non-negative integer with $p < q$.
Then p is a period of Q iff:

$$\forall\, 0 \leq i < n - q \ : Q[i] = Q[i+p].$$

Example

```
Shift: | 0 1 2 3 4 5 6 7 8 9 0 |
       |                      |
Word:  | A B R A C A D A B R A |              ⋮     ⋮
       |                      |
       |               A B R A|C A D A B R A        ⋮
```

Periods

Definition (Period)

Let $Q \in \Sigma^q$ and let p be a non-negative integer with $p < q$.
Then p is a period of Q iff:

$$\forall\, 0 \leq i < n - q \;:\; Q[i] = Q[i+p].$$

Example

```
Shift: | 0 1 2 3 4 5 6 7 8 9 0 |
       |                       |
Word:  | A B R A C A D A B R A |             :     :
       |                       |
       |         A B R A|C A D A B R A       :
       |               A|B R A C A D A B R A
```

Period set of $ABRACADABRA = \{0, 7, 10\}$.

Autocorrelation : example

- Q=ABRACADABRA. **M0 model** for random texts.

  $P(\mathtt{A}) = 0.4$, $P(\mathtt{B}) = 0.2$, $P(\mathtt{C}) = 0.1$, $P(\mathtt{D}) = 0.1$, $P(\mathtt{R}) = 0.2$.

- How does Q overlap with itself?

Shift: 0	1 2 3 4 5 6 7 8 9 0	P(Tail)
A B R A C A D A B R A	⋮ ⋮	$P(\varepsilon) = 1$
A B R A	C A D A B R A ⋮	$P(\mathtt{CADABRA}) = \frac{256}{10^7}$
A	B R A C A D A B R A	$P(\mathtt{BRACADABRA}) = \frac{4096}{10^{10}}$

- Autocorrelation vector:

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

- Autocorrelation polynomial:

$$C_Q(z) := \sum_{j=0}^{q-1} A_Q(j)\, z^j = 1 + \frac{256}{10^7}\, z^7 + \frac{4096}{10^{10}}\, z^{10}$$

P(word Q does not occur in text T)

- Word Q with autocorrelation polynomial $C(z)$
  Text $T^{(n)}$ of length n
- **Theorem**: Generating function in M0-model:

$$P(z) = \frac{C(z)}{\Pr(Q)z^q + (1-z)\cdot C(z)} = \sum_{j\geq 0} p_j\, z^j$$

[Guibas & Odlyzko 81a,81b, Chrysaphinou & Papastavridis 91]

- When $P(z)$ is rational, in certain conditions, one can computes $\mathbf{p_n}$,
  i.e., the coefficient of the $(n+1)^{th}$ term of $P(z)$,
  that is the probability that Q does not occur in $T^{(n)}$.

P(word Q does not occur in text T)

- Word Q with autocorrelation polynomial $C(z)$
  Text $T^{(n)}$ of length n
- **Theorem**: Generating function in M0-model:

$$P(z) = \frac{C(z)}{\Pr(Q)z^q + (1-z)\cdot C(z)} = \sum_{j\geq 0} p_j \, z^j$$

  [Guibas & Odlyzko 81a,81b, Chrysaphinou & Papastavridis 91]

- When $P(z)$ is rational, in certain conditions, one can computes $\mathbf{p_n}$,
  i.e., the coefficient of the $(n+1)^{th}$ term of $P(z)$,
  that is the probability that Q does not occur in $T^{(n)}$.

Extensions to estimate **how many *q*-grams**

1. are **missing** from a random text
2. are **common** to two random texts.

[Rahmann Rivals 00 & 03]

Approximate Pattern Matching

Approximate Pattern Matching

Given:

- a pattern M of m characters
- a text T of n characters
- k an integer such that $k \leq m$
- a cost function $e(U, V)$ between two strings U, V
  the unit cost edit distance.

Definition : find the starting positions i in T of any substring P such that $e(M, P) \leq k$.

Complexity : worst case $O(kn)$ with $O(m^2)$ preprocessing of P.

Others :
algorithm with sublinear expected time [Chang & Lawler, 1990]
filtration algorithms are efficient in pratice.

Filtration

Two phases algorithm: filtration and checking
based on a necessary condition (NC) for a match

Filtration : find all substrings P' of T verifying the NC
P' is called a *potential match*

Checking : check if a potential match is a match
this for all potential matches, use dynamic programming in $O(nm)$ time

Advantage

if the NC is easy to compute and potential matches are seldom,
only few substrings are checked using dynamic programming algorithm
$\Rightarrow$ Gain of execution time

Methods of filtration

- Reduction to exact partitionning [Baeza-Yates, Perleberg, 92]
- Maximal matches distance [Chang, Lawler, 90] [Ukkonen, 92]
- *q*-gram distance [Owolabi,Mc Gregor, 88], [Jokinen, Ukkonen, 91]
- Double filtration with gapped tuple (Pevzner-Waterman 94)

Heuristics: BLAST, FASTA, d2 [Torney et al, 90]

Reduction to exact partitionning (Baeza-Yates Perleberg 92)

Idea

1. cut the pattern in $k+1$ adjacent substrings of lg $\lfloor \frac{m}{k+1} \rfloor$
2. search for all pieces
3. if at most k errors are allowed,
   at least one piece matches exactly

Generalisations : a) cut in $k+s$ pieces and search for s distinct pieces conserving order in the pattern,
b) cut in j pieces and search each piece with $\lfloor \frac{k}{j} \rfloor$ errors

With index : [Baeza-Yates Navarro 96] use a table of the q-grams occurrences and reduce pieces to $q - gram$

q-gram distance (OM88, JU91)

Definition: q-gram or q-mer

A *q-gram* is a string of length q over an alphabet Σ.

Idea

Let $q \leq \lfloor \frac{m}{k+1} \rfloor$.

- count the number of matching q-grams between P and P'
- when $e(P, P') \leq k$, each error kills at most q q-grams
- thus at least $m - (k+1)q + 1$ q-grams match between P and P'.

q-gram (or k-mer) distance

Definition : a *q-gram* is a string of length q over an alphabet Σ

Idea : Let $q \leq \lfloor \frac{m}{k+1} \rfloor$, count the nb of q-grams equal between M & M' si $e(M, M') \leq k$, each difference affect at most q q-grams.

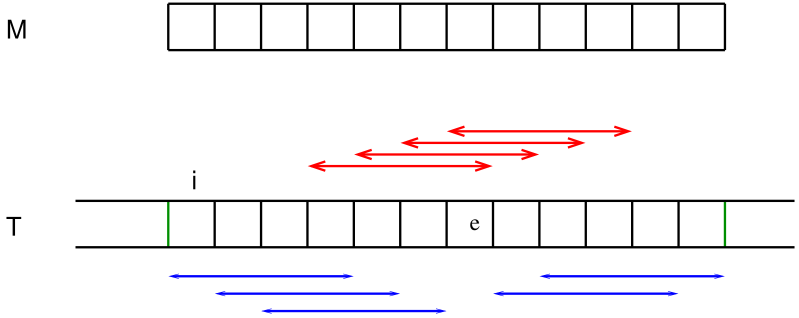
Worst case : $m - (k+1)q + 1$ q-grams match between M & M'.

q-gram filter

length of M: 12; $q := 4$

$\longleftrightarrow$: equal q-grams

$\longleftrightarrow$: different q-grams



[Owolabi, McGregor, 88]

Funding and acknowledgments

Thanks for your attention

Questions?

Bibliography: Exact Pattern Matching

Alfred Aho and Margaret Corasick.
Efficient string matching: an aid to bibliographic search.
Communications of the ACM, 18:333–340, 1975.

R. A. Baeza-Yates and C. H. Perleberg.
Fast and practical approximate string matching.
Information Processing Letters, 59(1):21–27, 1996.

R.S. Boyer and J.S. Moore.
A fast string searching algorithm.
Communications of the ACM, 20:762–772, 1977.

R. Nigel Horspool.
Practical fast searching in strings.
Softw. Pract. Exp., 10(6):501–506, 1980.
doi:10.1002/spe.4380100608.

D.E. Knuth, J.H. Morris, and V.R. Pratt.
Fast pattern matching in strings.
SIAM Journal of Computing, 6:323–350, 1977.

Bibliography: Indexing data structures

A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. Chen, and J. Seiferas.
The smallest automaton recognizing the subwords of a text.
Theoretical Computer Science, 40:31–55, 1985.

P. Ferragina and G. Manzini.
Opportunistic data structures with applications.
In *Proc. of FOCS*, pages 390–398, 2000.

U. Manber and G. W. Myers.
Suffix Arrays: A New Method for On-Line String Searches.
In *Proc. of ACM-SIAM SODA*, pages 319–327, Jan 22-24 1990. SIAM.

E. Ukkonen.
On-line construction of suffix-trees.
Algorithmica, 14(3):249–260, 1995.

P. Weiner.
Linear pattern matching algorithms.
In *Conf. Record of the 14th Annual Symposium on Swithcing and Automata Theory*, 1973.

Bibliography: Set Pattern Matching

A. Aho and M. Corasick.
Efficient string matching: an aid to bibliographic search.
Communications of the ACM, 18:333–340, 1975.
doi:10.1145/360825.360855.

E. Rivals, L. Salmela, P. Kalsi, P. Kiiskinen, and J. Tarhio.
Mpscan: fast localisation of multiple reads in genomes.
Proc. 9th Workshop Algorithms in Bioinformatics (WABI'09), volume 5724 of *LNCS*, pages 246–260, Sept 2009. Springer-Verlag.

Bibliography: Probabilistic Motifs (Position Weight Matrices)

M. Beckstette, R. Homann, R. Giegerich, and S. Kurtz.
Fast index based algorithms and software for matching position specific scoring matrices.
BMC Bioinformatics, 7(1), Aug 2006.
doi:10.1186/1471-2105-7-389.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson.
Biological Sequence Analysis.
Cambridge University Press, 1998.

J. Korhonen, P. Martinmäki, C. Pizzi, P. Rastas, and E. Ukkonen.
Moods: fast search for position weight matrix matches in dna sequences.
Bioinformatics, 25(23):3181–3182, 2009.
doi:10.1093/bioinformatics/btp554.

D. Martin, V. Maillol, and E. Rivals.
Fast and accurate genome-scale identification of dna-binding sites.
2018 IEEE Int Conf on Bioinformatics Biomedicine (BIBM), Dec 2018.
URL: doi:10.1109/bibm.2018.8621093.

Bibliography: Occurrence probabilities, periods

L. J. Guibas and A. M. Odlyzko.
String overlaps, pattern matching and nontransitive games.
J. of Combinatorial Theory series A, 30:183–208, 1981.

L. J. Guibas and A. M. Odlyzko.
Periods in strings. *J. of Combinatorial Theory series A*, 30:19–42, 1981.

S. Rahmann and E. Rivals.
Exact and Efficient Computation of the Expected Number of Missing and Common Words in Random Texts.
Proc. CPM, volume 1848 of *LNCS*, pages 375–387. 2000.
doi:10.1007/3-540-45123-4_31.

S. Rahmann and E. Rivals.
The number of missing words in random texts.
Combinatorics, Probability and Computing, 12:73–87, 2003.
doi:10.1017/s0963548302005473.

E. Rivals and S. Rahmann.
Combinatorics of Periods in Strings.
J. of Combinatorial Theory series A, 104(1):95–113, Oct 2003.
doi:10.1016/s0097-3165(03)00123-7.

Bibliography: Books

Maxime Crochemore, Christophe Hancart, and Thierry Lecroq.
Algorithms on strings.
Cambridge University Press, 2007.

Dan Gusfield.
Algorithms on Strings, Trees and Sequences.
Cambridge University Press, 1997.

Gonzalo Navarro and Matthieu Raffinot.
Flexible Pattern Matching in Strings - Practical on-line search algorithms for texts and biological sequences.
Cambridge Univ. Press, 2002.

Enno Ohlebusch.
Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.
Oldenbusch Verlag, 2013. URL: `http://www.oldenbusch-verlag.de/`.

Stéphane Robin, François Rodolphe, and Sophie Schbath.
DNA, Words and Models.
Cambridge Univ. Press, 2005.