An efficient matching algorithm for encoded DNA sequences and binary strings

Simone Faro and Thierry Lecroq

faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Dipartimento di Matematica e Informatica, Università di Catania, Italy

University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France

Combinatorial Pattern Matching

22 – 24 June 2009 – Lille, France

Outline

Outline

1. Introduction
2. A new algorithm
3. Experimental Results

Problem

Searching for all exact occurrences of a pattern p ($|p| = m$) in a text t ($|t| = n$)
where
both p and t are bitstreams

Example

$p =$ 1100101100100100101100010001 and
$t =$ 0101010001010101010100100111001011001001001011001000101001010011001001

Requirement

Avoid the access to individual bits $\longrightarrow$ access to blocks of k bits

Special cases

Each character of p and t consists of

- a single bit $\longrightarrow$ binary sequences
- a couple of bits $\longrightarrow$ encoded DNA sequences

Problem

Searching for all exact occurrences of a pattern p ($|p| = m$) in a text t ($|t| = n$)
where
both p and t are bitstreams

Example

$p =$ 11001011001001001011001000 1 and

$t =$ 0101010001010101010100100111001011001001001011001000101001010011001001

Requirement

Avoid the access to individual bits $\longrightarrow$ access to blocks of k bits

Special cases

Each character of p and t consists of

- a single bit $\longrightarrow$ binary sequences
- a couple of bits $\longrightarrow$ encoded DNA sequences

Existing solutions

S. T. Klein and M. K. Ben-Nissan
Accelerating Boyer Moore searches on binary texts
CIAA, LNCS 4783, pp 130–143, 2007

J. W. Kim, E. Kim, and K. Park
Fast matching method for DNA sequences
Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, LNCS 4614, pp 271–281, 2007

S. Faro and T. Lecroq
Efficient pattern matching on binary strings
SOFSEM, poster, 2009

Outline

Preprocessing

The algorithm computes

1. a table of k copies of p, in order to process text and pattern block by block (as in [Klein & Ben-Nissan 2007])
2. bit-mask vectors to implement a multi-pattern version of the BNDM algorithm
3. an index-list table to identify candidate alignments during the searching phase
4. a shift table based on the bad-character heuristic to increase the length of the shifts

Byte

- We suppose that the block size k is fixed
- All references to both text and pattern will only be to entire blocks of k bits
- We refer to a k-bit block as a byte though larger values than $k = 8$ could be supported
- $T[i]$ and $P[i]$ denote, respectively, the $(i + 1)$-th byte of the text and of the pattern
- The last byte may be only partially defined.
- We suppose that the undefined bits of the last byte are set to 0.

k copies of p

- We define k copies, denoted by $Patt[i]$ of the pattern p shifted by i position to the right, for $0 \leq i < k$
- $i \in \mathbf{P} = \{0, 1, \ldots, k-1\}$
- In each pattern $Patt[i]$, the i leftmost bits of the first byte remain undefined and are set to 0
- Similarly the rightmost $((k - ((m+i) \mod k) \mod k)$ bits of the last byte are set to 0

Example

$p =$110010110010010010110010001 **of length 27**

Patt	0	1	2	3	4
i 0	11001011	00100100	10110010	00100000	
1	01100101	10010010	01011001	00010000	
2	00110010	11001001	00101100	10001000	
3	00011001	01100100	10010110	01000100	
4	00001100	10110010	01001011	00100010	
5	00000110	01011001	00100101	10010001	
6	00000011	00101100	10010010	11001000	10000000
7	00000001	10010110	01001001	01100100	01000000

Additional information to the k copies

- b_i : the index of the first byte in $Patt[i]$ containing a k-substring of p
- e_i : the index of the last byte of the pattern $Patt[i]$.
- m_i : the number of bytes in $Patt[i]$ containing k-substrings of p
- $F1[i]$: bit mask for the first byte of $Patt[i]$
- $F2[i]$: bit mask for the last byte of $Patt[i]$

Example

$p =$110010110010010010110010001 **of length 27**

$Patt$	0	1	2	3	4
i 0	11001011	00100100	10110010	00100000	
1	01100101	10010010	01011001	00010000	
2	00110010	11001001	00101100	10001000	
3	00011001	01100100	10010110	01000100	
4	00001100	10110010	01001011	00100010	
5	00000110	01011001	00100101	10010001	
6	00000011	00101100	10010010	11001000	10000000
7	00000001	10010110	01001001	01100100	01000000

b_i	e_i	m_i
0	3	3
1	3	2
1	3	2
1	3	2
1	3	2
1	3	3
1	4	3
1	4	3

$F1$
11111111
01111111
00111111
00011111
00001111
00000111
00000011
00000001

$F2$
11100000
11110000
11111000
11111100
11111110
11111111
10000000
11000000

Example

$p =$110010110010010010110010001 **of length 27**

Patt	0	1	2	3	4
i 0	11001011	00100100	10110010		
1		10010010	01011001		
2		11001001	00101100		
3		01100100	10010110		
4		10110010	01001011		
5		01011001	00100101	10010001	
6		00101100	10010010	11001000	
7		10010110	01001001	01100100	

Example

p =110010110010010010110010001 **of length 27**

Patt	0	1	2	3	4
i 0	11001011	00100100			
1		10010010	01011001		
2		11001001	00101100		
3		01100100	10010110		
4		10110010	01001011		
5		01011001	00100101		
6		00101100	10010010		
7		10010110	01001001		

Bit-parallelism

- The algorithm uses bit-parallelism to simulate the behavior of a NFA constructed over the set of patterns $Patt[i]$
- However, in order to let the automaton fit in a single machine word of size ω, only the substrings $Patt[i][b_i \mathinner{.\,.} b_i + m - 1]$ are handled by the automaton
- $m = \min(\{m_i\} \cup \{\omega\})$
- $\mathcal{P}$=set of remaining k patterns of length m

Bit-parallelism

- $m + 1$ different states: $Q = \{0, 1, 2, 3, \ldots, m\}$
- m different transitions: state q, with $0 < q \leq m$, has a transition towards state $q - 1$ labeled with the class of characters $\{Patt[i][s_i + q]\}$
- m is the initial state

$p =$110010110010010010110010001 of length 27

$Patt$	0	1	2	3	4
i 0	11001011=L	00100100=A			
1		10010010=H	01011001=F		
2		11001001=K	00101100=C		
3		01100100=G	10010110=I		
4		10110010=J	01001011=E		
5		01011001=F	00100101=B		
6		00101100=C	10010010=H		
7		10010110=I	01001001=D		

$\omega = 32$

	M
00100100=A	00000000000000000000000000000001
00100101=B	00000000000000000000000000000001
00101100=C	00000000000000000000000000000011
01001001=D	00000000000000000000000000000001
01001011=E	00000000000000000000000000000001
01011001=F	00000000000000000000000000000011
01100100=G	00000000000000000000000000000010
10010010=H	00000000000000000000000000000011
10010110=I	00000000000000000000000000000011
10110010=J	00000000000000000000000000000010
11001001=K	00000000000000000000000000000010
11001011=L	00000000000000000000000000000010
$c \notin \{A, B, C, D, E, F, G, H, I, J, K, L\}$	00000000000000000000000000000000

Index list

- The NFA recognizes also words that are not substrings of the pattern
- However, in order to make a filter the algorithm maintains, for each block $B \in \{0, \ldots, 2^k - 1\}$, a linked list λ which is used to find candidate patterns
- In particular, for each block $B \in \{0, \ldots, 2^k - 1\}$:
  $\lambda[B] = \{i \mid Patt[i, b_i + m - 1] = B\}$
- When a block sequence is recognized by the automaton, ending at block position j of the text, the algorithm naively checks for the occurrence of any pattern $Patt[g]$, with $g \in \lambda[T[j]]$

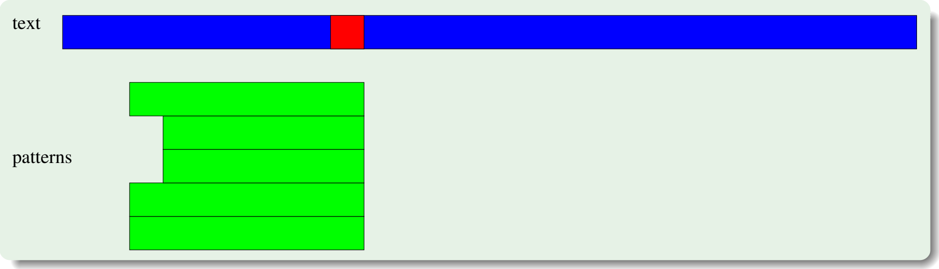
Example

p =110010110010010010110010001 **of length 27**

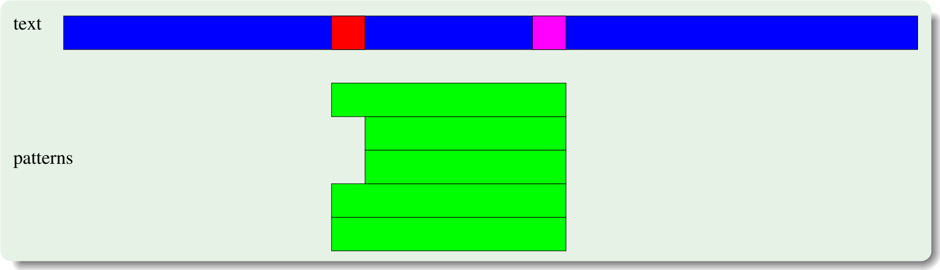
Patt	0	1	2	3	4
i 0	11001011=L	00100100=A			
1		10010010=H	01011001=F		
2		11001001=K	00101100=C		
3		01100100=G	10010110=I		
4		10110010=J	01001011=E		
5		01011001=F	00100101=B		
6		00101100=C	10010010=H		
7		10010110=I	01001001=D		

	λ
00100100=A	$\{0\}$
00100101=B	$\{5\}$
00101100=C	$\{2\}$
01001001=D	$\{7\}$
01001011=E	$\{4\}$
01011001=F	$\{1\}$
10010010=H	$\{6\}$
10010110=I	$\{3\}$
$c \notin \{A, B, C, D, E, F, H, I\}$	$\emptyset$

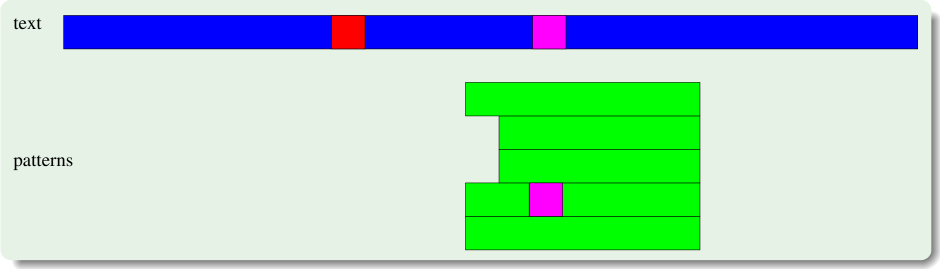
Shift table



Shift table



Shift table



Shift table

- The algorithm uses a *long shift* rule which is a multi-pattern version of the original bad-character shift heuristic, improved with an efficient look-ahead
- This shift rule is used when no substring is recognized while scanning the text from right to left
- In such a case the current window of the text can be safely advanced by m positions to the right

Shift table

- Observe that, if j is the right position of the current window of the text, the block at position $j+m$ is always involved in the next attempt, thus we can use it to compute the next window position
- $ls : \{0, \ldots, 2^k - 1\} \to \{m, \ldots, 2 \times m - 1\}$
  $ls[B] =$
  $m - 1 + \min\{$distance of B from the right end of a pattern in P$\}$
- the next right position of the window is $j + ls[T[j+m-1]]$

$p =$110010110010010010110010001 **of length 27**

Patt	0	1	2	3	4
i 0	11001011$=L$	00100100$=A$			
1		10010010$=H$	01011001$=F$		
2		11001001$=K$	00101100$=C$		
3		01100100$=G$	10010110$=I$		
4		10110010$=J$	01001011$=E$		
5		01011001$=F$	00100101$=B$		
6		00101100$=C$	10010010$=H$		
7		10010110$=I$	01001001$=D$		

	ls
00100100$=A$	$1+0$
00100101$=B$	$1+0$
00101100$=C$	$1+0$
01001001$=D$	$1+0$
01001011$=E$	$1+0$
01011001$=F$	$1+0$
01100100$=G$	$1+1$
10010010$=H$	$1+0$
10010110$=I$	$1+0$
10110010$=J$	$1+1$
11001001$=K$	$1+1$
11001011$=L$	$1+1$
$c \notin \{A, B, C, D, E, F, G, H, I, J, K, L\}$	$1+2$

The searching phase

2 parts:

- a match phase
- a shift phase

The shift phase

- The NFA is represented by a state vector D of size m (the first state of the automaton is not represented)
- Like in the SBNDM algorithm [Holub & Durian 2005], each iteration starts with a test of 2 consecutive text characters and implements a fast-loop to obtain better results on average
- Such a fast loop makes use of the long shift table to compute the next window alignment

The match phase

- Right to left scan in a window of size m, ending at position j in the text, as in the BNDM algorithm
- The state vector is updated in a similar fashion as in the SHIFT-AND algorithm [BYG92]
- If the state vector D is equal to 0 after $\ell + 1$ updates of D, then a word of length ℓ has been recognized by the automaton
- If $\ell = m$ a candidate alignment has been found and the algorithm naively checks the occurrence of any pattern contained in the index list $\lambda[T[j]]$
- In all cases, after the match phase, the index j is advanced of $m - \ell + 1$ to the right and the algorithm restarts its computation with the shift phase

The match phase

If a candidate alignment is found for a text position j, for each index $i \in \lambda[T[j]]$, the algorithm uses the precomputed table $Patt[i]$ to check whether $s = j - m - b_i + 1$ is a valid shift

Example

p =11001011001001001011001000 1

A=00100100 B=00100101 C=00101100 D=01001001 E=01001011 F=01011001
G=01100100 H=10010010 I=10010110 J=10110010 K=11001001 L=11001011

0101010001010101010100100111001011001001001011001000101001010011001001

Example

p =110010110010010010110010001

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

	j							
0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100

Example

p =1100101100100100101100010001

A=00100100 B=00100101 C=00101100 D=01001001 E=01001011 F=01011001
G=01100100 H=10010010 I=10010110 J=10110010 K=11001001 L=11001011

j

0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100

$D = 0$

Example

$p =$ 11001011001001001011001000 1

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

	j	$j+1$						
0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100

$\notin \{A, B, C, D, E, F, G, H, I, J, K, L\}$

$$ls = 3$$

Example

p =11001011001001001011001000 1

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

				j				
0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100

$D = 0$

Example

p =110010110010010010110010001

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

				j	$j+1$			
0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100
			$D=0$		$ls[C]=1$			

Example

p =1100101100100100101100100101

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

					j			
0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100

$$D \neq 0$$

Example

p =110010110010010010110010001

A=00100100	B=00100101	C=00101100	D=01001001	E=01001011	F=01011001
G=01100100	H=10010010	I=10010110	J=10110010	K=11001001	L=11001011

110010 11001001 00101100 10001

j

0	1	2	3	4	5	6	7	8
01010100	01010101	01010010	01110010	11001001	00101100	10001010	01010011	00100100
			00111111	K	C	11111000		
			$F1[2]$	$D \neq 0$		$F2[2]$		

$$\lambda(C) = 2$$

Complexity

	time	space
$Patt, b_i, e_i, m_i, F1, F2$	$O(k \times \lceil m/k \rceil) = O(m)$	$O(m)$
Bit masks	$O(2^k + km)$	$O(2^k)$
Index list	$O(2^k + k)$	$O(2^k)$
Shift table	$O(2^k + m)$	$O(2^k)$
Searching phase	$O(\lceil n/k \rceil \lceil m/k \rceil k) = O(n \times m)$	
Overall	$O(2^k + (n + k)m)$	$O(2^k + m)$

Handling encoded DNA sequences

- Each base is represented by a couple of bits
- Thus a DNA sequence γ can be represented with a bitstream of $(2 \times |\gamma|)$ bits
- Any occurrence of a given encoded pattern p starts at an even position of the text
- This suggests that only even alignments of the pattern have to be processed
- The only change to be applied, when handling encoded DNA sequences, is in the preprocessing of the set of patterns
- Specifically the set $\mathbf{P}$ is defined by

$$\mathbf{P} = \{i \mid 0 \leq i < k \text{ and } (i \mod 2) = 0\}$$

- For instance, if each block consists of $k = 8$ bits, we have $\mathbf{P} = \{0, 2, 4, 6\}$

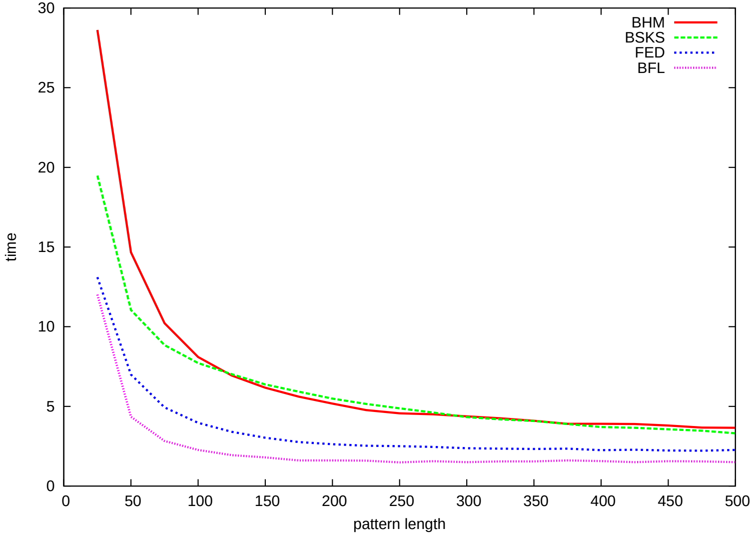
Outline

Experimental Results

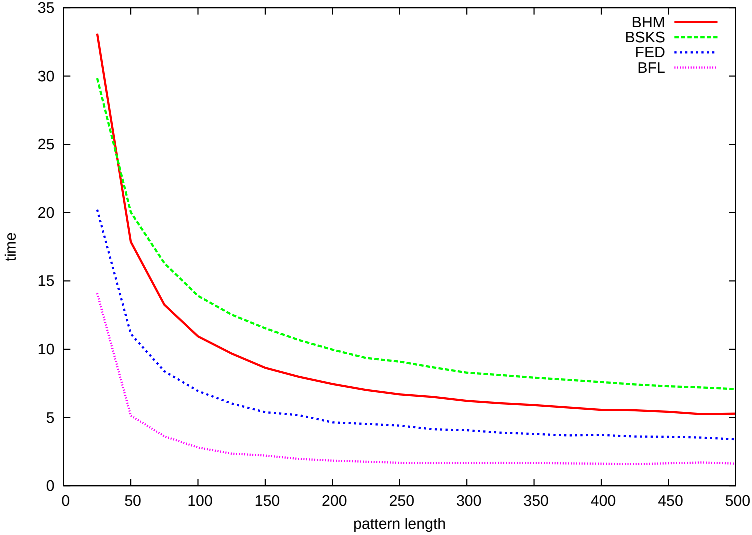
- BBM, Klein and Ben-Nissan, 2007
- FED, Kim, Kim and Park, 2007
- BHM, Faro and Lecroq, SOFSEM 2009
- BSKS, Faro and Lecroq, SOFSEM 2009
- BFL, Faro and Lecroq, CPM 2009

All algorithms have been implemented in the **C** programming language and were used to search for the same binary strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66GHz.

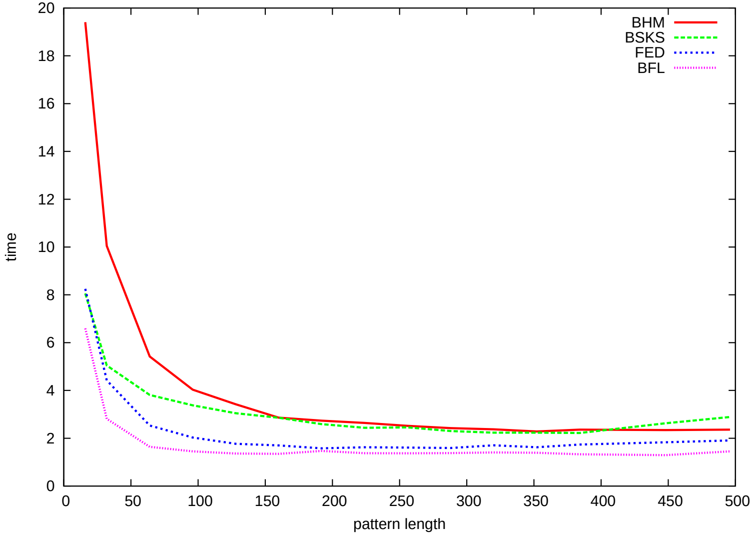
Experimental results for a Rand$(0/1)_{50}$ problem



Experimental results for a Rand$(0/1)_{70}$ problem



Experimental results for an encoded DNA sequence



Conclusion and perspectives

- algorithm for exact matching on binary strings and encoded DNA sequences
- combines a multi-pattern version of the BNDM algorithm with a simplified shift strategy of CW algorithm
- efficient in practical cases
- adapts easily to the exact multiple string matching case