Efficient Variants of the Backward-Oracle-Matching Algorithm

Simone Faro[1] and Thierry Lecroq[2]

[1]Dipartimento di Matematica e Informatica, Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[2]Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract

In this article we present two efficient variants of the BOM string matching algorithm which are more efficient and flexible than the original algorithm. We also present bit-parallel versions of them obtaining an efficient variant of the BNDM algorithm. Then we compare the newly presented algorithms with some of the most recent and effective string matching algorithms. It turns out that the new proposed variants are very flexible and achieve very good results, especially in the case of large alphabets.

The String Matching Problem

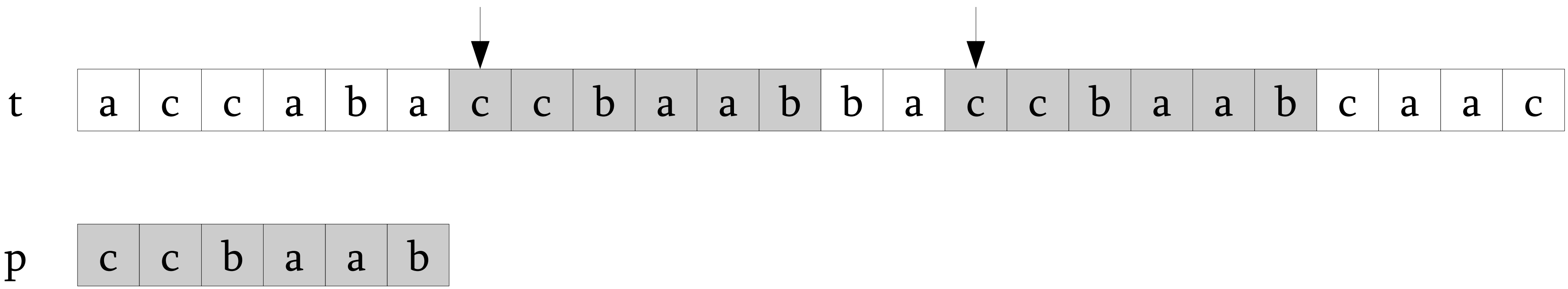
Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the **string matching problem** consists in finding all occurrences of the pattern p in the text t

t	a	c	c	a	b	a	c	c	b	a	a	b	b	a	c	c	b	a	a	b	c	a	a	c

p	c	c	b	a	a	b

The String Matching Problem

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the **string matching problem** consists in finding all occurrences of the pattern p in the text t



The String Matching Problem

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the **string matching problem** consists in finding all occurrences of the pattern p in the text t

Applications:

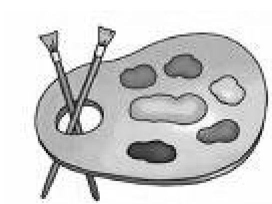
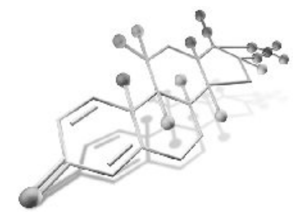


Image processing

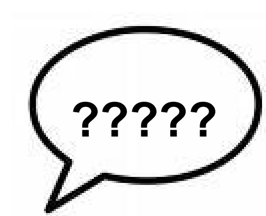
Signal processing

Computational biology



Computational chemestry

Speech analysis



Speech recognition

The String Matching Problem

Brute Force
DFA
Karp-Rabin
Shift Or
Morris-Pratt
Knuth-Morris-Pratt
Simon
Colussi
Galil-Giancarlo
Apostolico-Crochemore
Not So Naive
Boyer-Moore
Turbo BM
Apostolico-Giancarlo
Reverse Colussi
Horspool
Quick Search
Tuned Boyer-Moore

Zhu-Takaoka
Berry-Ravindran
Smith
Raita
Reverse Factor
Turbo Reverse Factor
Forward Dawg Matching
BNDM
BOM
Galil-Seiferas
Two Way
Optimal Mismatch
Maximal Shift
Skip Search
KMP Skip Search
Alpha Skip Search
Fast Search
Forward Fast Search

The String Matching Problem

- Brute Force
- DFA
- Karp-Rabin
- Shift Or
- Morris-Pratt
- Knuth-Morris-Pratt
- Simon
- Colussi
- Galil-Giancarlo
- Apostolico-Crochemore
- Not So Naive
- **Boyer-Moore**
- Turbo BM
- Apostolico-Giancarlo
- Reverse Colussi
- Horspool
- Quick Search
- Tuned Boyer-Moore
- Zhu-Takaoka
- Berry-Ravindran
- Smith
- Raita
- Reverse Factor
- Turbo Reverse Factor
- Forward Dawg Matching
- BNDM
- BOM
- Galil-Seiferas
- Two Way
- Optimal Mismatch
- Maximal Shift
- Skip Search
- KMP Skip Search
- Alpha Skip Search
- Fast Search
- Forward Fast Search

The String Matching Problem

- Brute Force
- <u>DFA</u>
- Karp-Rabin
- <u>Shift Or</u>
- Morris-Pratt
- **<u>Knuth-Morris-Pratt</u>**
- Simon
- Colussi
- Galil-Giancarlo
- Apostolico-Crochemore
- Not So Naive
- Boyer-Moore
- Turbo BM
- Apostolico-Giancarlo
- Reverse Colussi
- Horspool
- Quick Search
- Tuned Boyer-Moore
- Zhu-Takaoka
- Berry-Ravindran
- Smith
- Raita
- <u>Reverse Factor</u>
- <u>Turbo Reverse Factor</u>
- <u>Forward Dawg Matching</u>
- <u>BNDM</u>
- <u>BOM</u>
- Galil-Seiferas
- Two Way
- Optimal Mismatch
- Maximal Shift
- Skip Search
- KMP Skip Search
- Alpha Skip Search
- Fast Search
- Forward Fast Search

Automata

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata, data structures which identify all factors of a word.

- **BOM** (Backward Oracle Matching) algorithm **[2]** is the most efficient, especially for long patterns.

- **BNDM** (Backward Nondeterministic Dawg Match) algorithm **[3]**, is very efficient for short patterns.

[2] C. Allauzen, M. Crochemore, andM. Raffinot. Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartosek, editors, SOFSEM'99, Theory and Practice of Informatics, number 1725 in Lecture Notes in Computer Science, pages 291–306, Milovy, Czech Republic, 1999. Springer-Verlag, Berlin.

[3] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, number 1448, pages 14–33, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

Factor Automaton

The factor automaton of a pattern p, Aut(p), is also called the factor DAWG of p (for Directed Acyclic Word Graph). Such an automaton recognizes all the factors of p. Formally the language recognized by Aut(p) is defined as follows

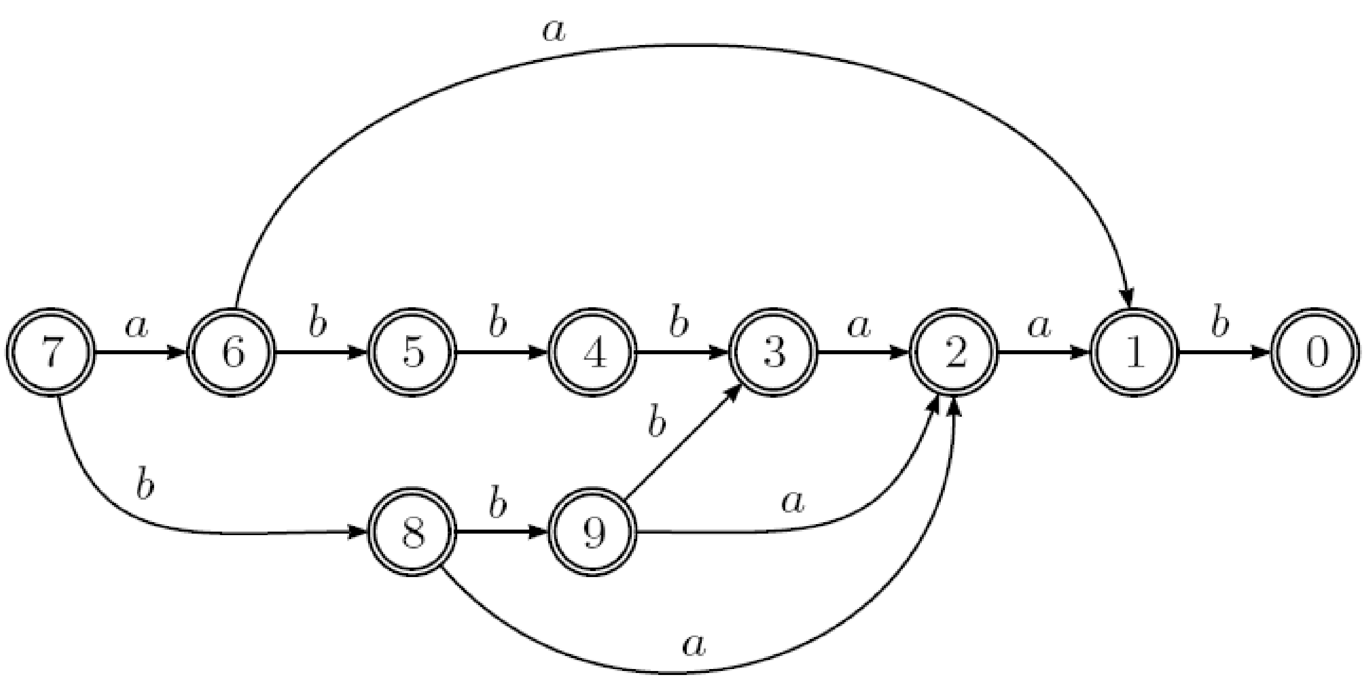
$$\mathcal{L}(Aut(p)) = \{u \in \Sigma^* : \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}.$$

Factor Automaton

The factor automaton of a pattern p, Aut(p), is also called the factor DAWG of p (for Directed Acyclic Word Graph). Such an automaton recognizes all the factors of p. Formally the language recognized by Aut(p) is defined as follows

$$\mathcal{L}(Aut(p)) = \{u \in \Sigma^* : \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}.$$

The factor automaton of the pattern p = abbbaab



Factor Oracle

The factor oracle of a pattern p, Oracle(p), is a very compact automaton which recognizes at least all the factors of p and slightly more other words. Formally Oracle(p) is an automaton $\{Q, m, Q, \Sigma, \delta\}$ such that

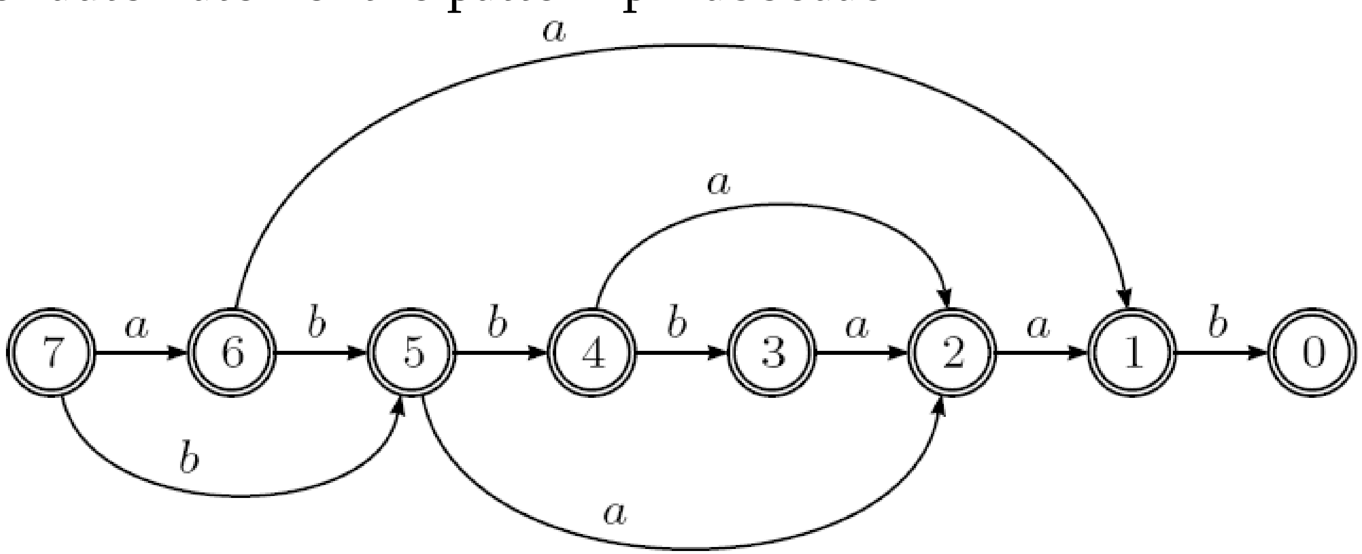
1. Q contains exactly m + 1 states, say $Q = \{0, 1, 2, 3, \ldots, m\}$
2. m is the initial state
3. all states are final
4. the language accepted by Oracle(p) is such that $L(Aut(p)) \subseteq L(Oracle(p))$

Factor Oracle

The factor oracle of a pattern p, Oracle(p), is a very compact automaton which recognizes at least all the factors of p and slightly more other words. Formally Oracle(p) is an automaton {Q,m,Q,Σ,δ} such that

1. Q contains exactly m + 1 states, say Q = {0, 1, 2, 3, . . . ,m}
2. m is the initial state
3. all states are final
4. the language accepted by Oracle(p) is such that L(Aut(p)) ⊆ L(Oracle(p))

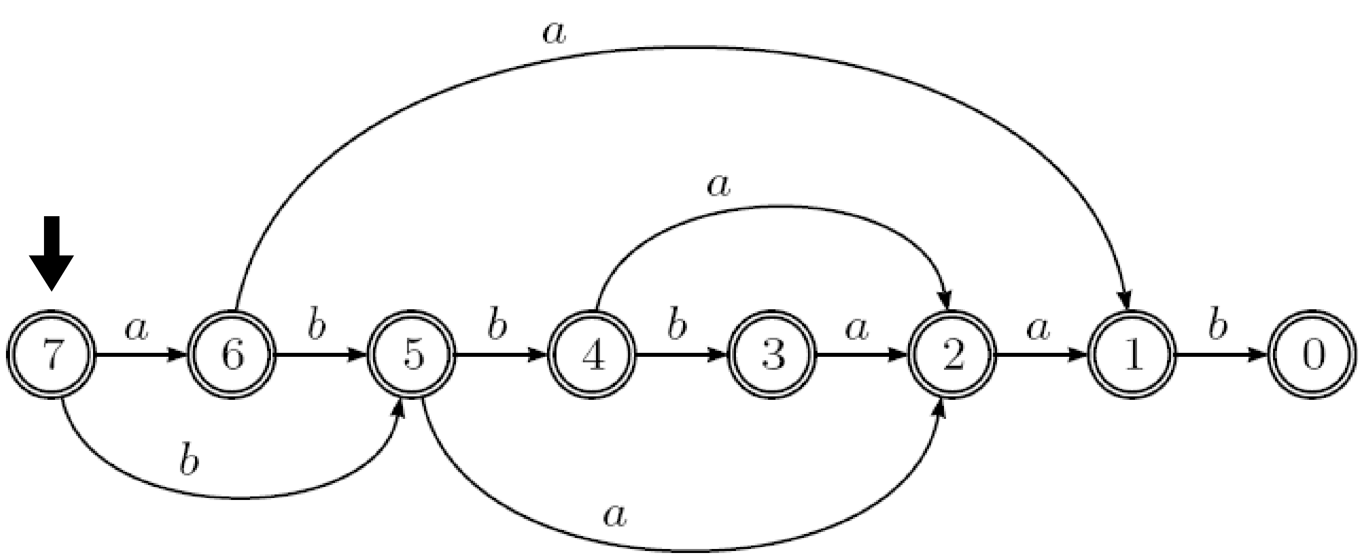
The factor automaton of the pattern p = abbbaab



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

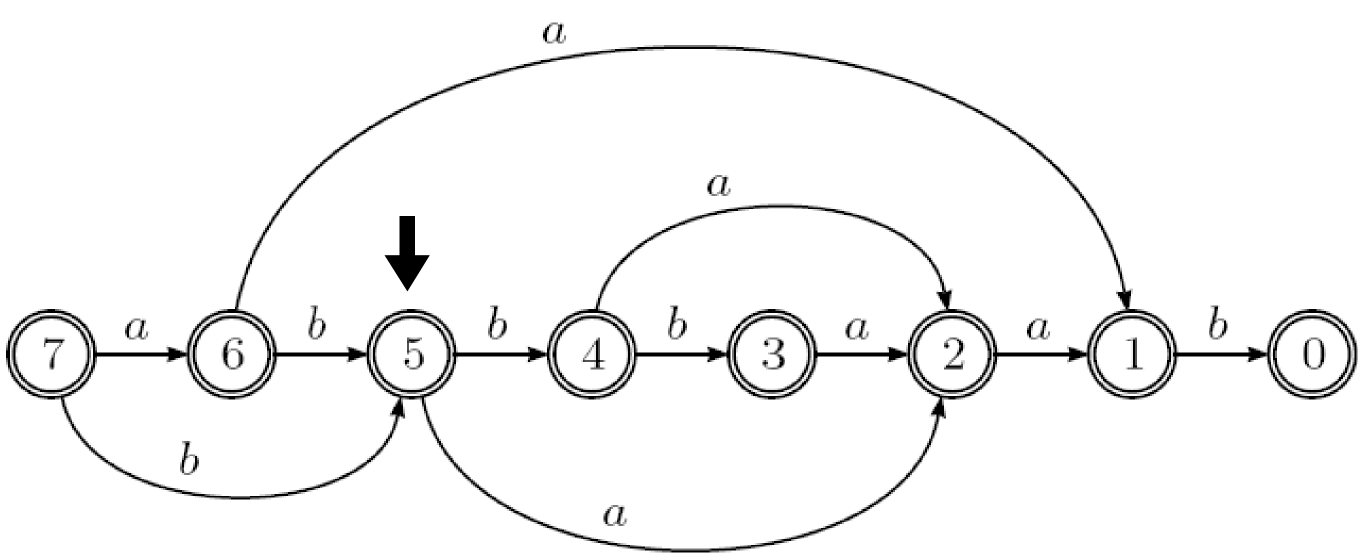
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

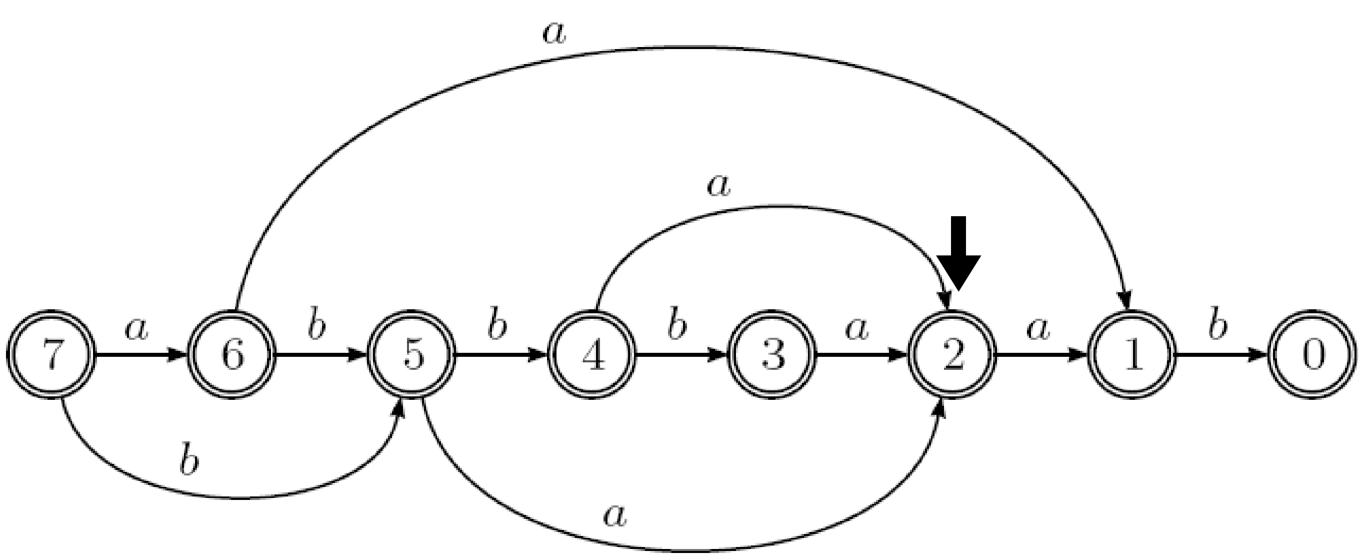
t																								
	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

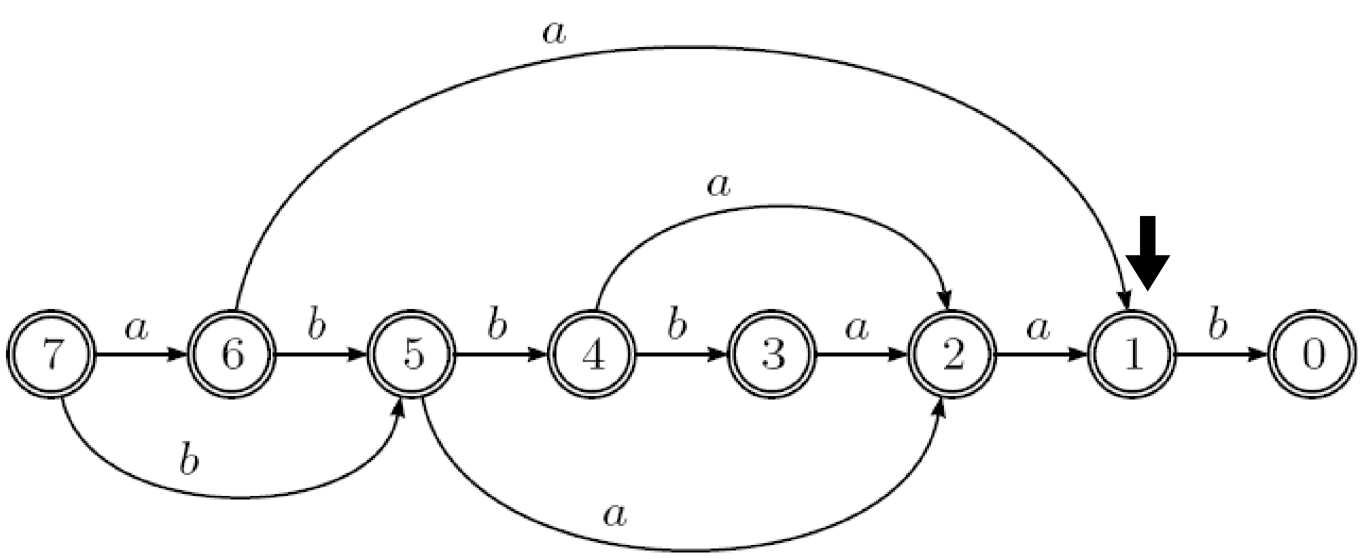
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

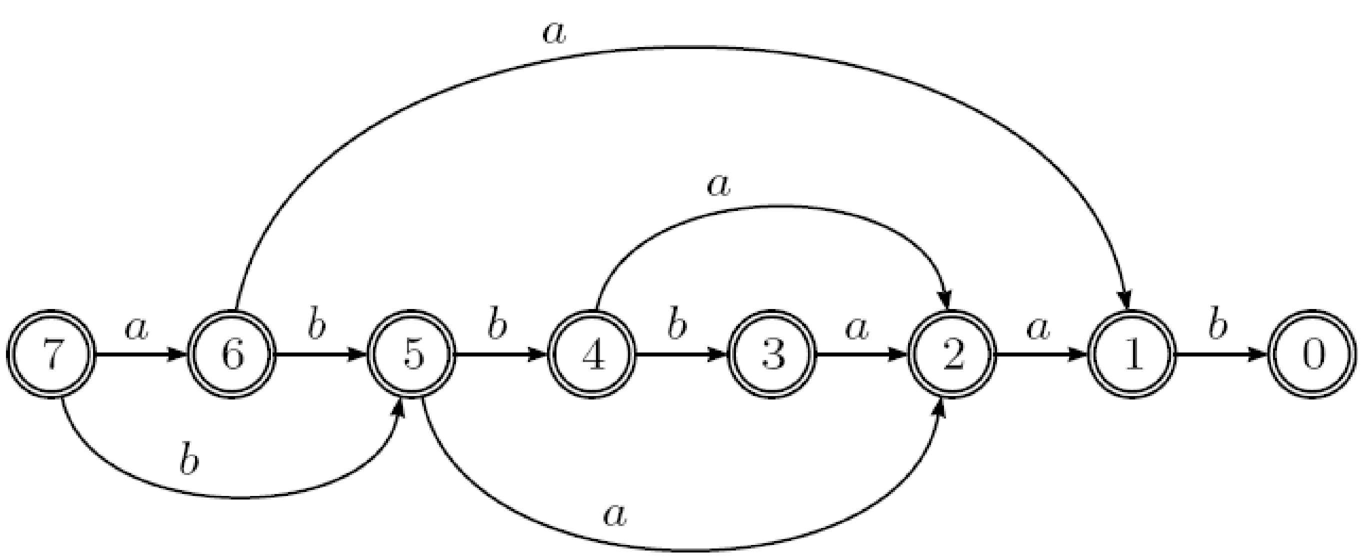
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

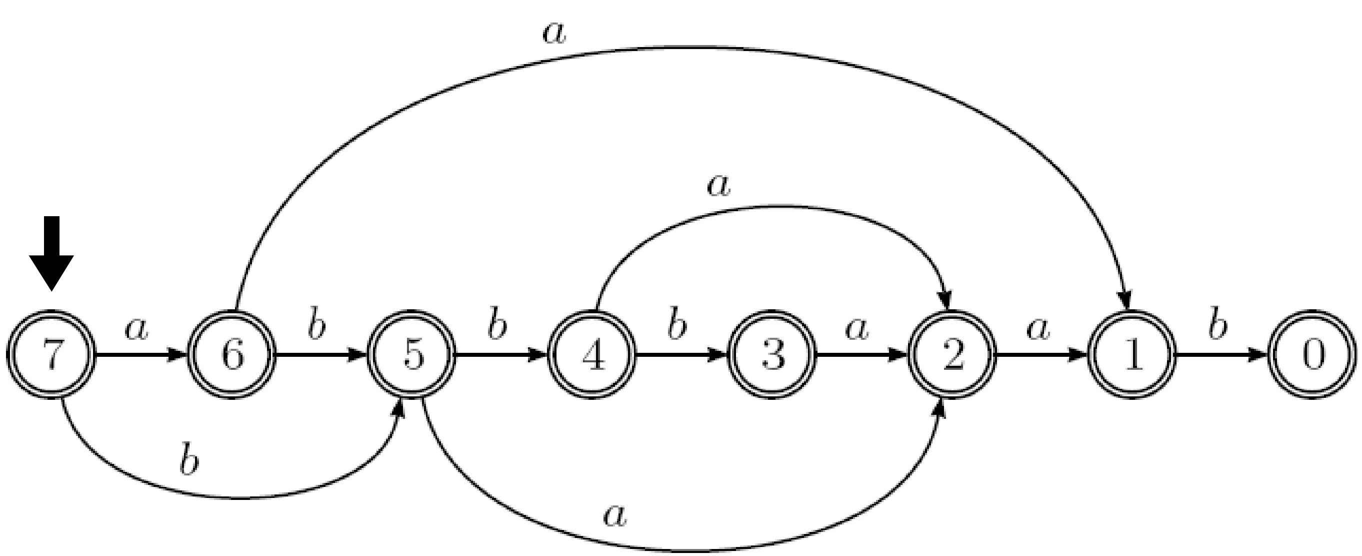
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

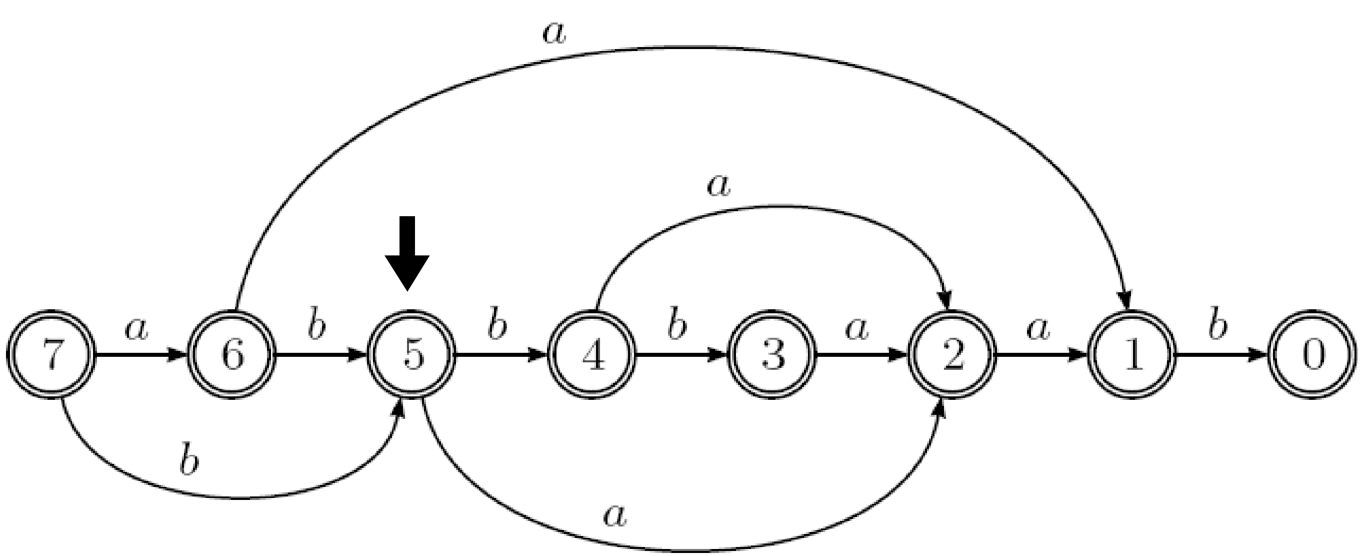
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

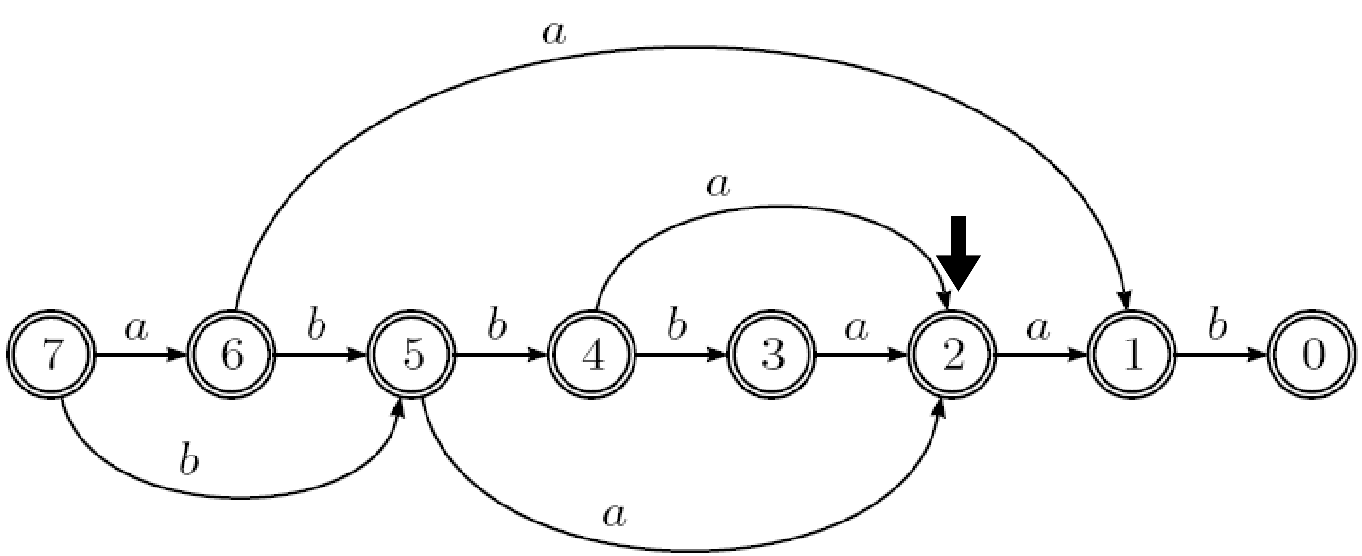
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

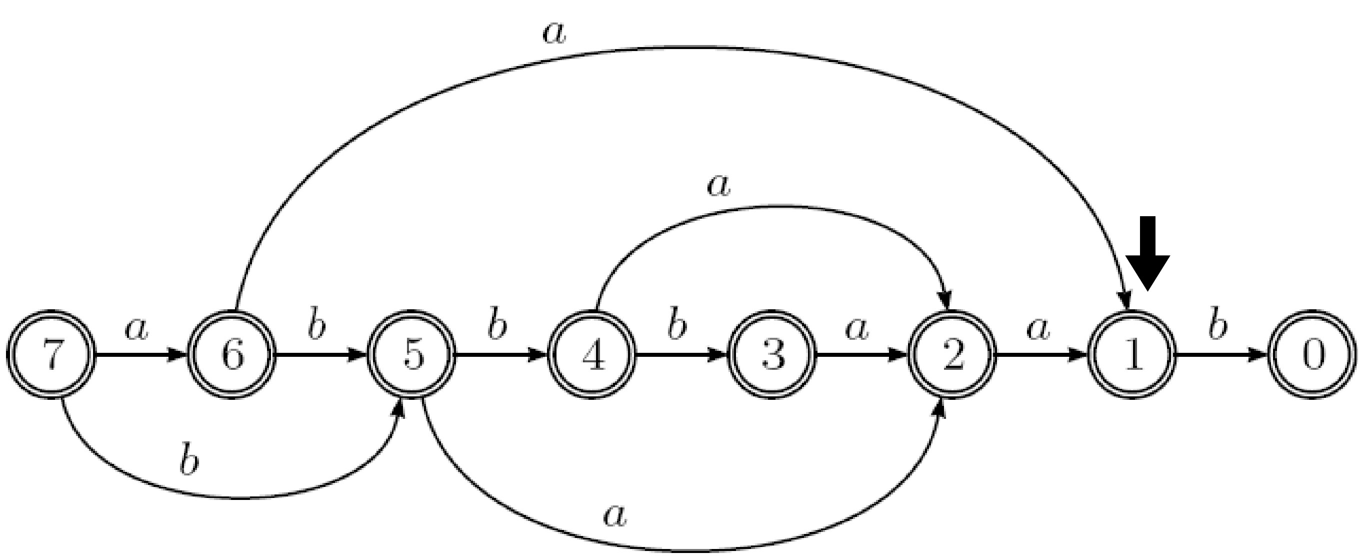
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

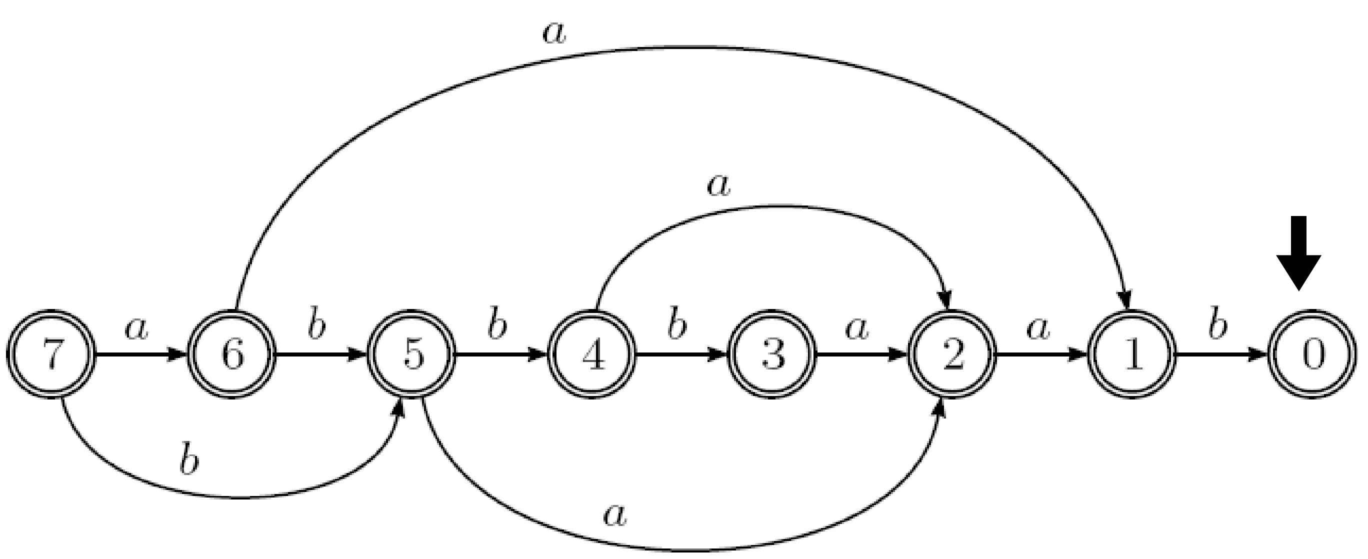
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

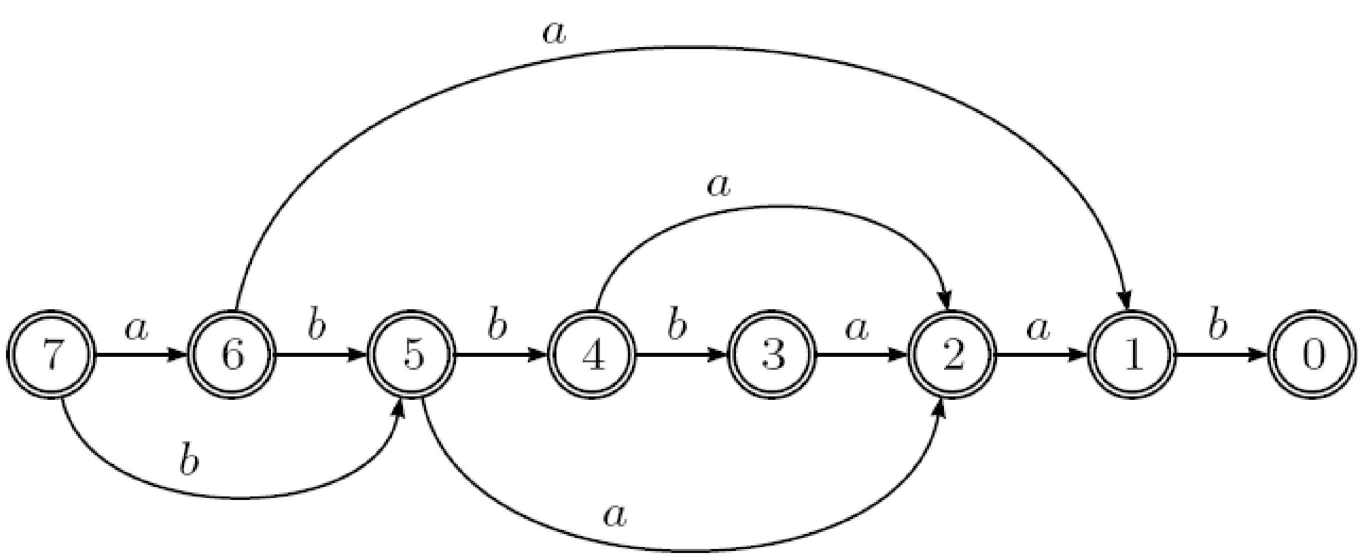
t																								
	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

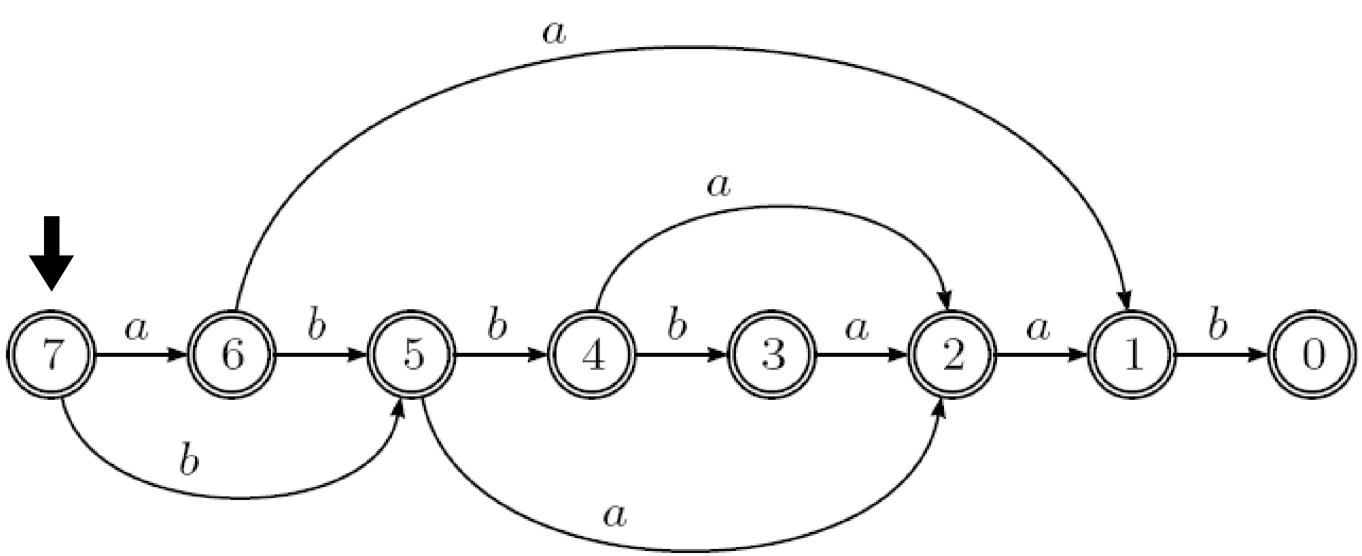
t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Bacward Oracle Matching

The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

t	a	c	c	a	a	a	b	b	a	a	b	b	b	a	a	c	b	a	a	b	c	a	a	c



Extending BOM

Now we present an extension of the BOM algorithm by introducing a fast-loop with the aim of obtaining better results on the average. We discuss the application of different variations of the fast-loop and present experimental results in order to identify the best choice.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.

Generally a fast-loop is implemented by iterating the bad character heuristic in a checkless cycle, in order to quickly locate an occurrence of the rightmost character of the pattern.

$$bc(c) = \min(\{0 \leq k < m \mid p[m-1-k] = c\} \cup \{m\})$$

(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
    $j = j + k$
    $k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.

t	a	c	c	a	a	a	b	b	a	a	b	c	b	a	a	c

p	a	b	a	c	c

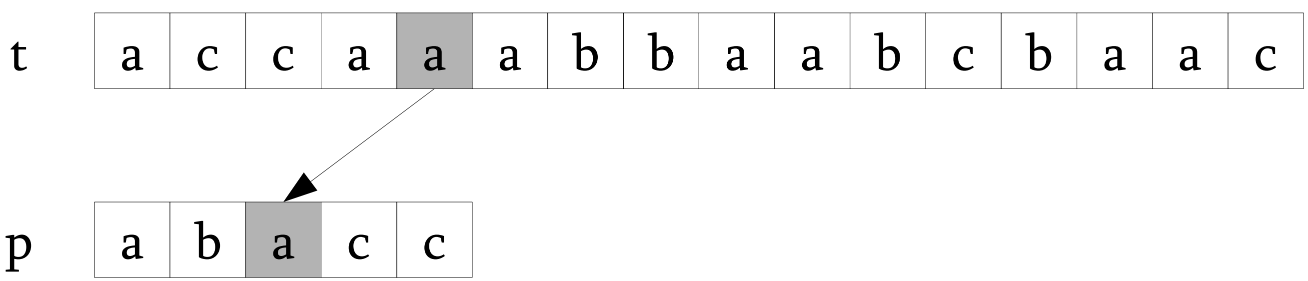
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



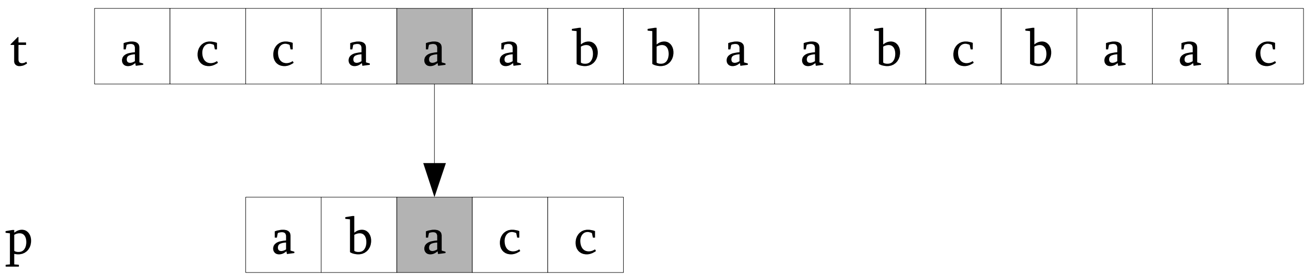
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



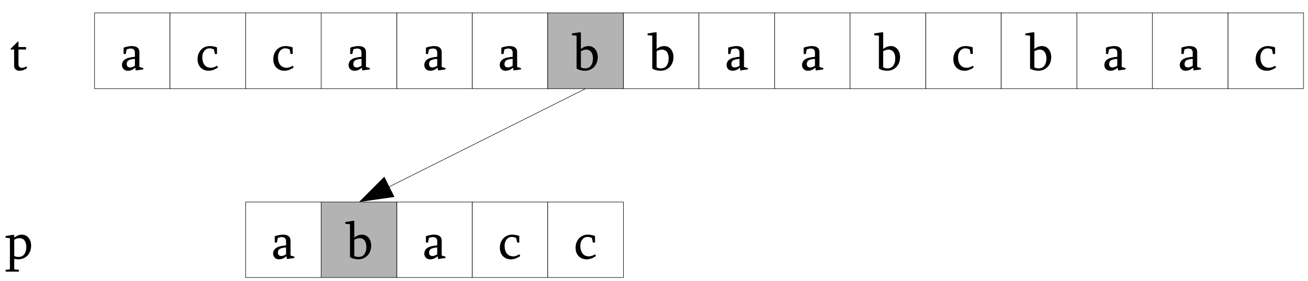
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



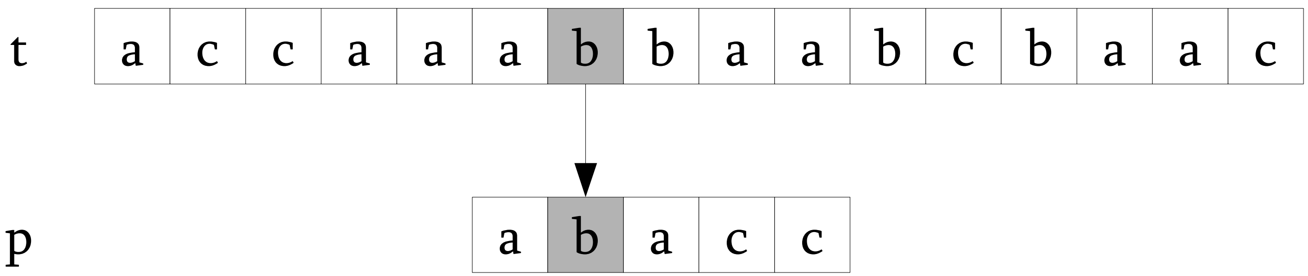
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



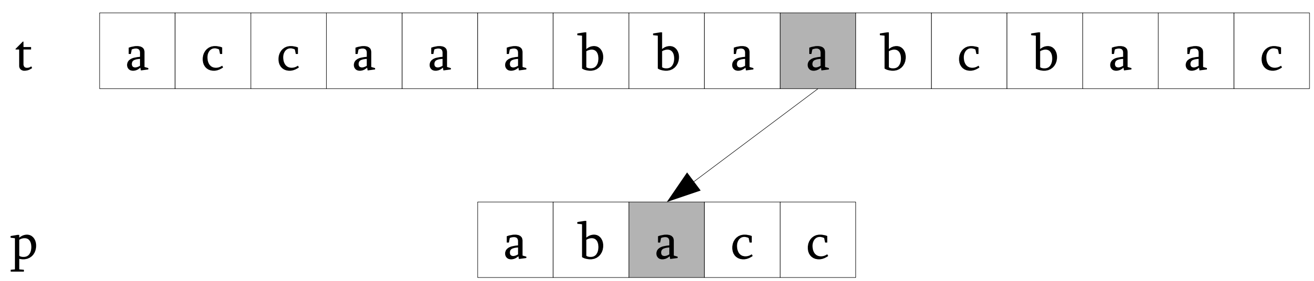
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
  $j = j + k$
  $k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



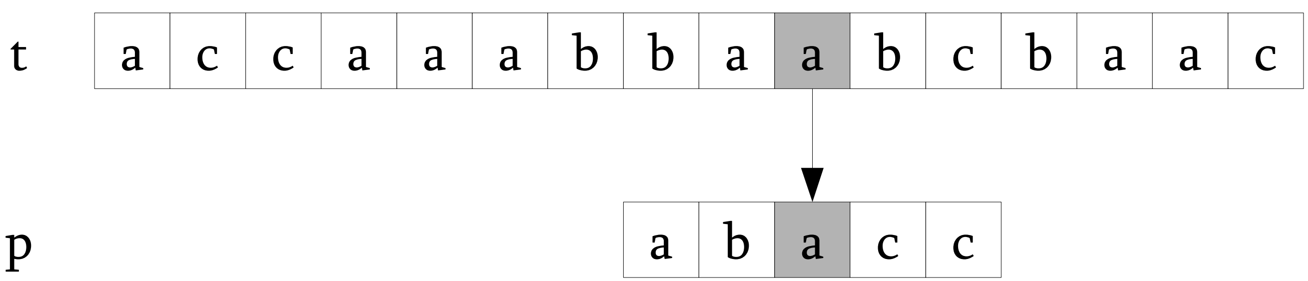
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



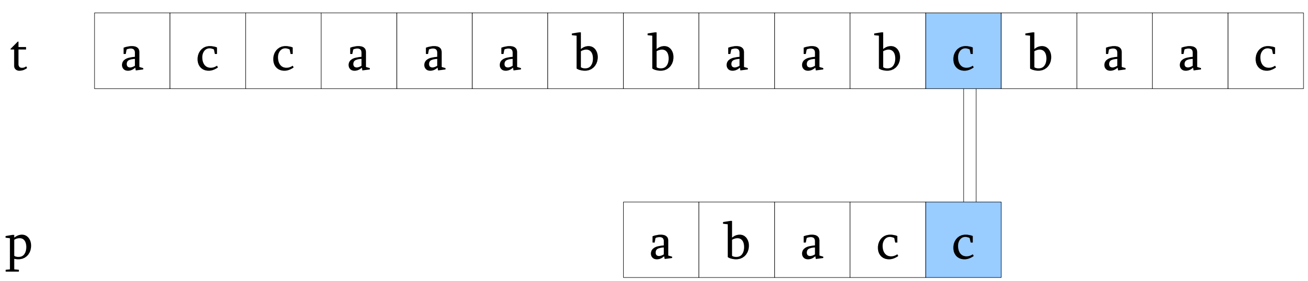
(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
$\quad j = j + k$
$\quad k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm **[4]** and later largely used in almost all variations of the Boyer-Moore algorithm.



(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
  $j = j + k$
  $k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

The classical fast-loop

The classical fast-loop has first introduced in the Tuned-Boyer-Moore algorithm [4] and later largely used in almost all variations of the Boyer-Moore algorithm.

Experimental results with $\sigma = 8$

m	BOM	(A)
4	157.62	95.95
8	85.48	58.66
16	43.04	43.36
32	26.63	35.00
64	17.39	28.05
128	15.28	23.68
256	10.79	19.86
512	6.18	14.29
1024	3.29	8.20

Experimental results with $\sigma = 16$

m	BOM	(A)
4	103.28	66.81
8	71.59	38.72
16	39.61	26.57
32	18.68	21.80
64	12.67	20.09
128	14.22	19.38
256	8.81	19.05
512	4.62	17.73
1024	2.35	11.49

Experimental results with $\sigma = 32$

m	BOM	(A)
4	78.76	55.23
8	51.68	30.37
16	35.40	19.92
32	20.62	16.12
64	12.11	14.84
128	12.60	15.63
256	7.58	16.73
512	4.29	17.90
1024	2.87	14.19

Experimental results with $\sigma = 64$

m	BOM	(A)
4	64.84	50.93
8	39.35	27.44
16	26.09	17.12
32	19.45	14.09
64	13.15	13.58
128	13.11	17.67
256	6.25	18.04
512	2.91	18.00
1024	2.71	16.89

(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
  $j = j + k$
  $k = bc(t_j)$

[4] A. Hume and D. M. Sunday. Fast string searching. Softw. Pract. Exp., 21(11):1221–1248, 1991.

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

(B)

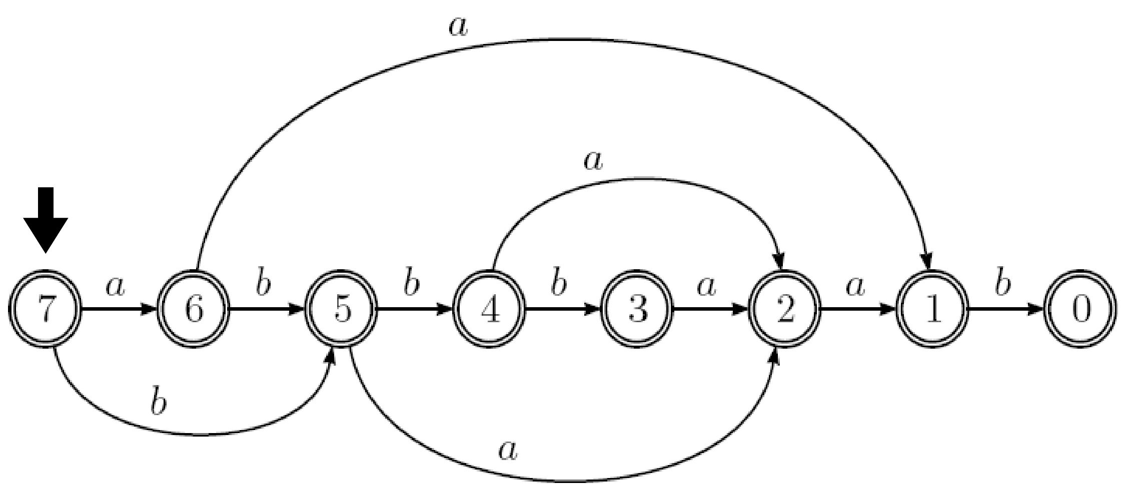
$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
    $j = j + m$
    $q = \delta(m, t_j)$

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

t	a	c	c	a	a	a	c	b	a	a	b	c	b	a	a	c

p	b	a	a	b	b	b	a



(B)

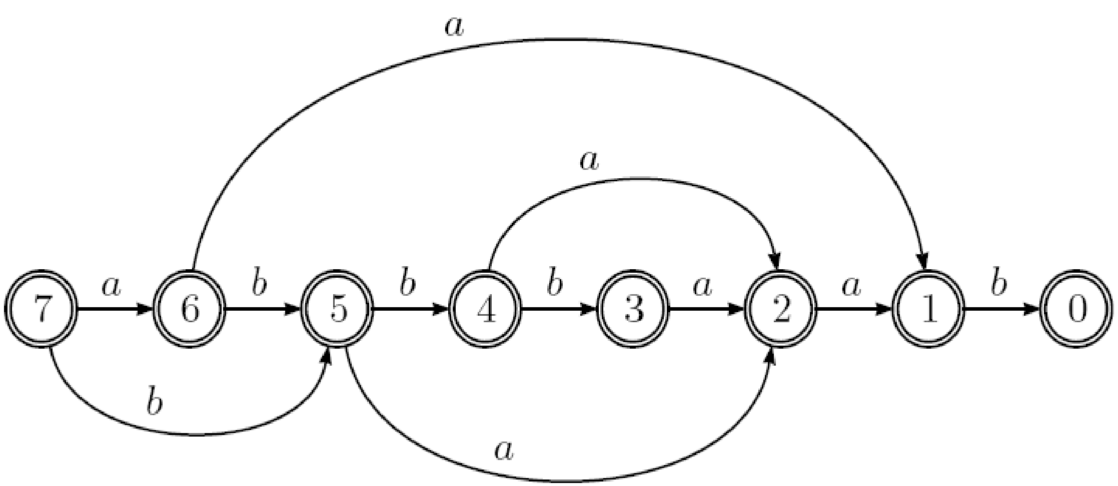
$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
$\quad j = j + m$
$\quad q = \delta(m, t_j)$

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

t	a	c	c	a	a	a	c	b	a	a	b	c	b	a	a	c

p	b	a	a	b	b	b	a



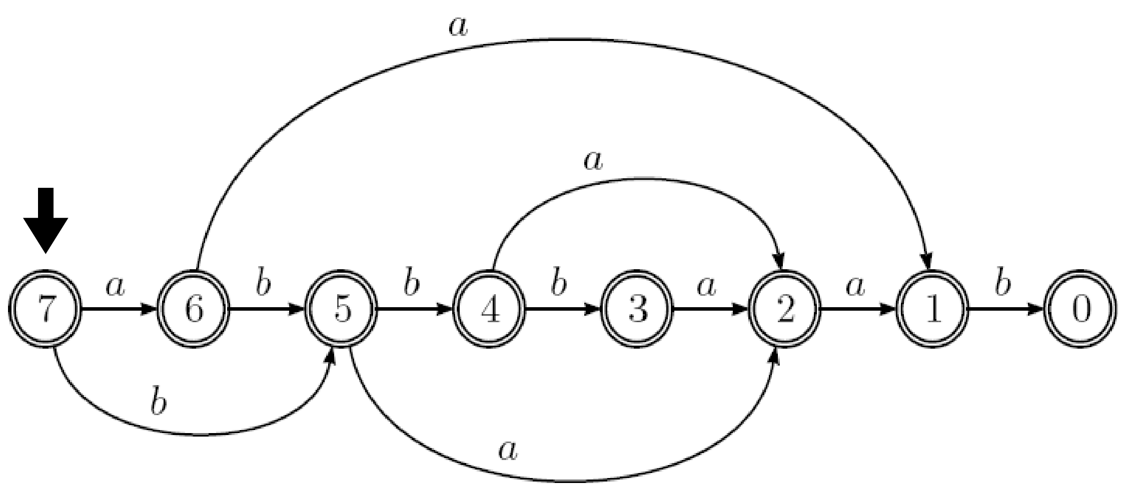
(B)

$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
$\quad j = j + m$
$\quad q = \delta(m, t_j)$

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

t	a	c	c	a	a	a	c	b	a	a	b	c	b	a	a	c
p								b	a	a	b	b	b	a		



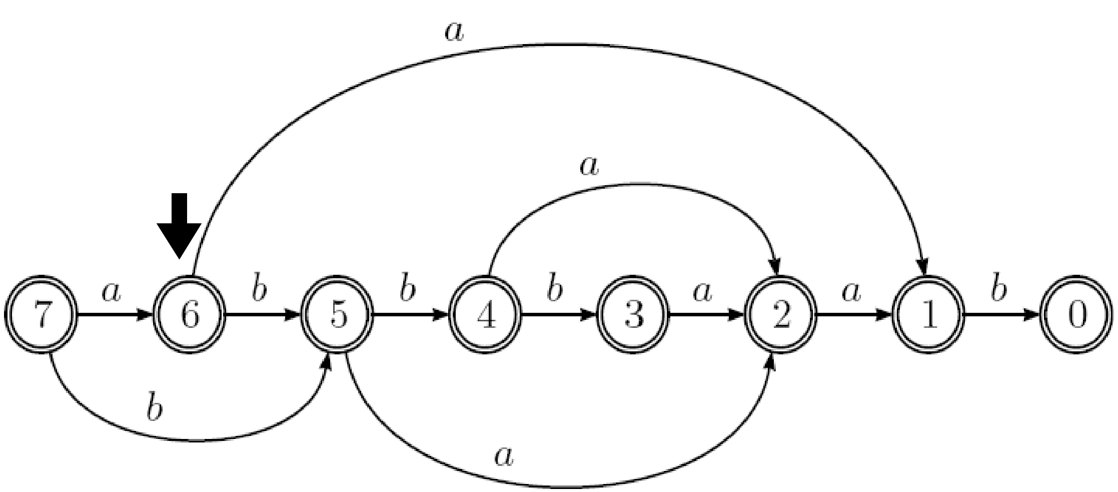
(B)

$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
$\quad j = j + m$
$\quad q = \delta(m, t_j)$

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

t	a	c	c	a	a	a	c	b	a	a	b	c	b	a	a	c
p							b	a	a	b	b	b	a			



(B)

$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
$j = j + m$
$q = \delta(m, t_j)$

A fast-loop over transitions

We can translate the idea of the fast-loop over automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text.

Experimental results with $\sigma = 8$

m	BOM	(A)	(B)
4	157.62	95.95	135.95
8	85.48	58.66	78.70
16	43.04	43.36	43.00
32	26.63	35.00	28.29
64	17.39	28.05	17.13
128	15.28	23.68	15.75
256	10.79	19.86	9.60
512	6.18	14.29	6.11
1024	3.29	8.20	3.45

Experimental results with $\sigma = 16$

m	BOM	(A)	(B)
4	103.28	66.81	86.28
8	71.59	38.72	60.27
16	39.61	26.57	35.70
32	18.68	21.80	18.82
64	12.67	20.09	12.55
128	14.22	19.38	14.14
256	8.81	19.05	8.12
512	4.62	17.73	4.62
1024	2.35	11.49	2.66

(B)

$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
$\quad j = j + m$
$\quad q = \delta(m, t_j)$

Experimental results with $\sigma = 32$

m	BOM	(A)	(B)
4	78.76	55.23	57.75
8	51.68	30.37	42.03
16	35.40	19.92	30.18
32	20.62	16.12	19.34
64	12.11	14.84	11.55
128	12.60	15.63	11.26
256	7.58	16.73	6.32
512	4.29	17.90	3.73
1024	2.87	14.19	2.67

Experimental results with $\sigma = 64$

m	BOM	(A)	(B)
4	64.84	50.93	42.34
8	39.35	27.44	29.29
16	26.09	17.12	22.03
32	19.45	14.09	17.11
64	13.15	13.58	12.28
128	13.11	17.67	10.86
256	6.25	18.04	5.79
512	2.91	18.00	3.12
1024	2.71	16.89	2.58

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

(C)

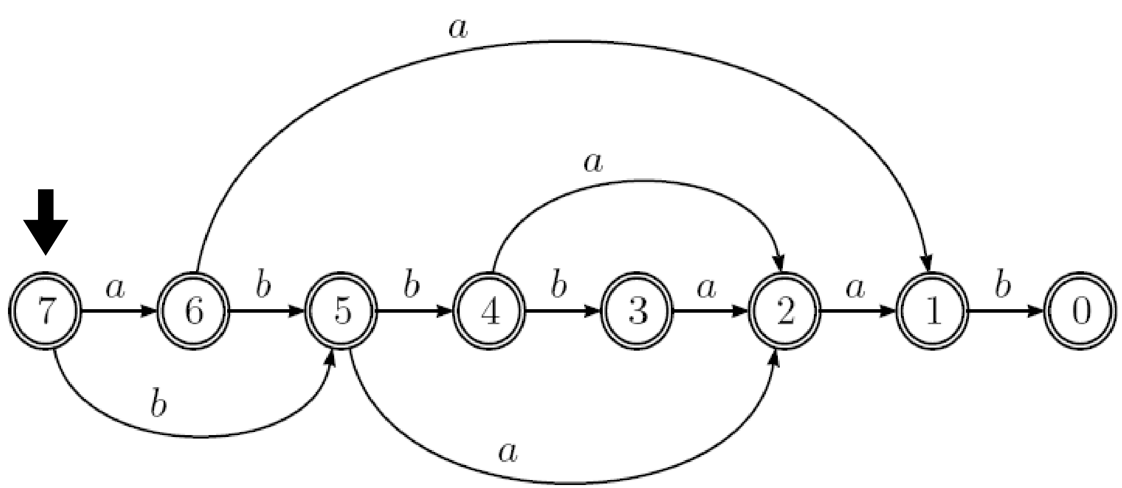
$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a



(C)

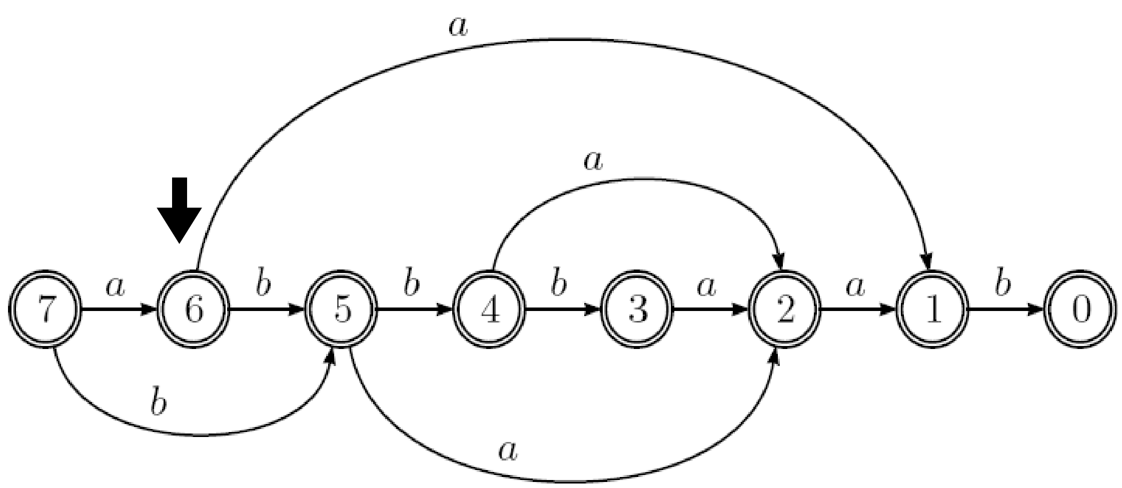
$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
    $j = j + m - 1$
    $q = \delta(m, t_j)$
    if $q \neq \perp$ **then**
        $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a



(C)

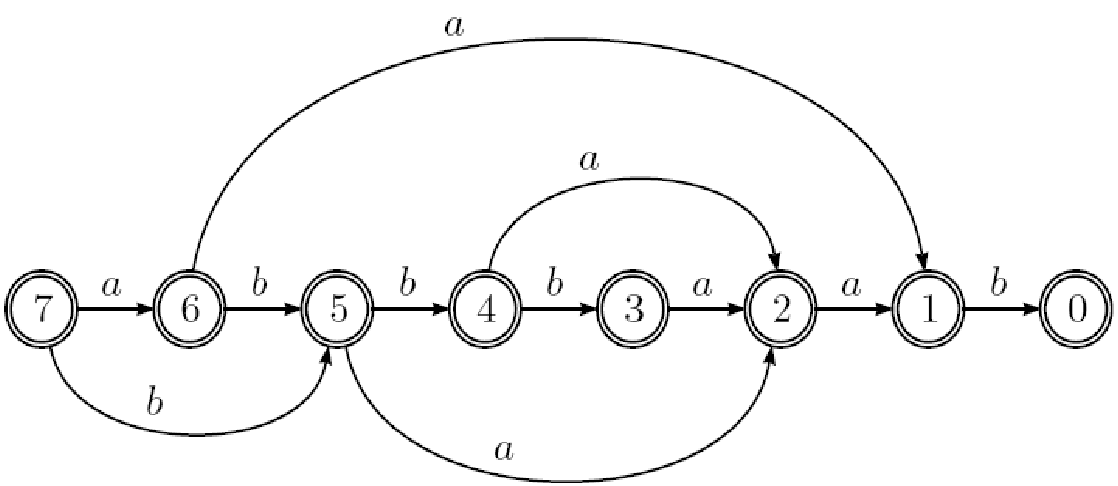
$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a



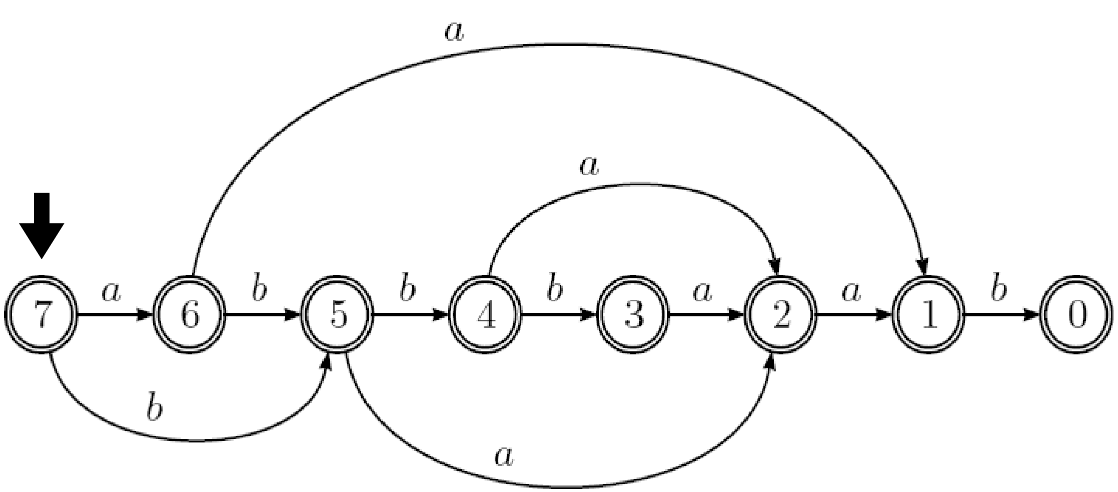
(C)

$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c
p							b	a	a	b	b	b	a			



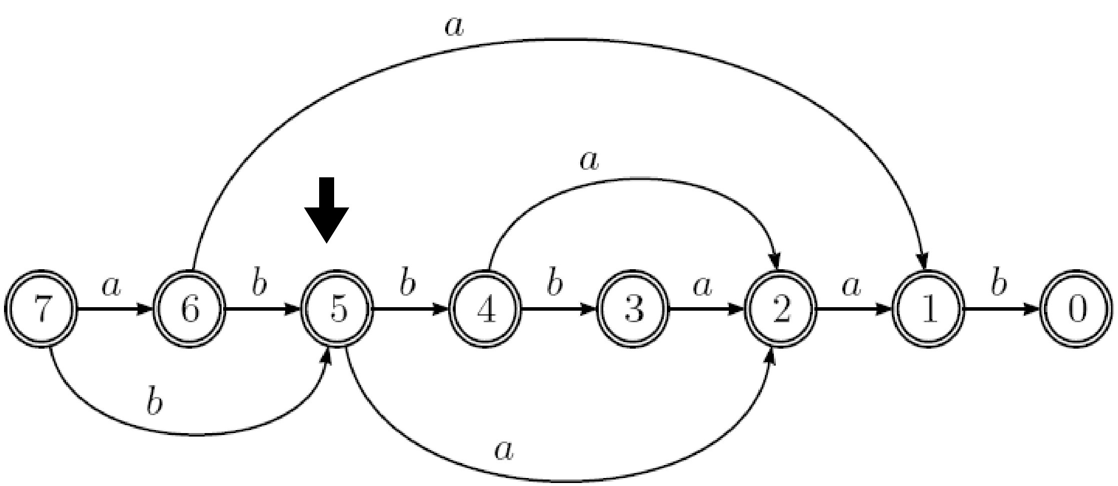
(C)

$q = \delta(m, t_j)$
if $q \neq \bot$ **then**
    $p = \delta(q, t_{j-1})$
while $(p == \bot)$ **do**
    $j = j + m - 1$
    $q = \delta(m, t_j)$
    if $q \neq \bot$ **then**
        $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c
p							b	a	a	b	b	b	a			



(C)

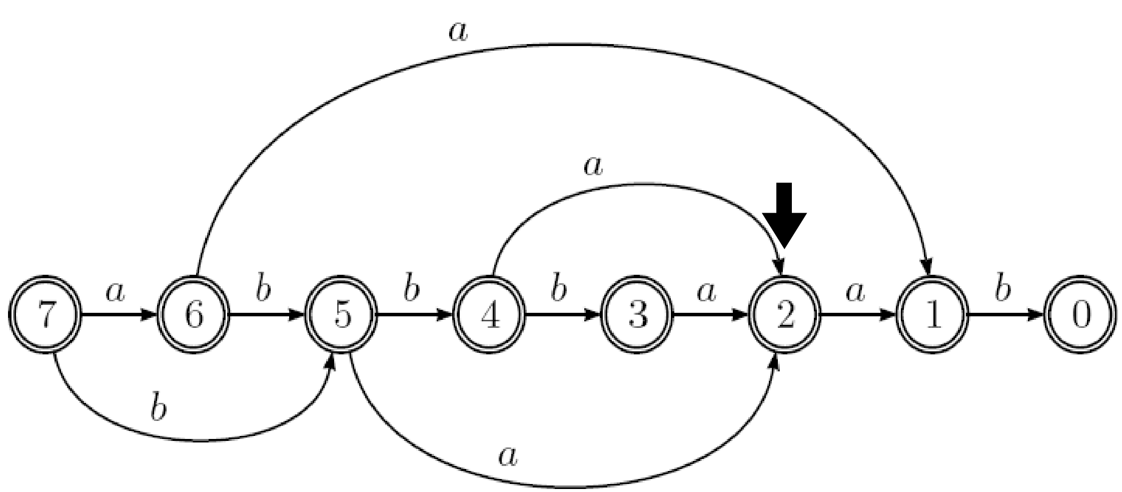
$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a



(C)

$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition.

Experimental results with $\sigma = 8$

m	BOM	(A)	(B)	(C)
4	157.62	95.95	135.95	109.03
8	85.48	58.66	78.70	58.63
16	43.04	43.36	43.00	37.15
32	26.63	35.00	28.29	25.93
64	17.39	28.05	17.13	17.00
128	15.28	23.68	15.75	15.87
256	10.79	19.86	9.60	9.76
512	6.18	14.29	6.11	5.76
1024	3.29	8.20	3.45	3.35

Experimental results with $\sigma = 16$

m	BOM	(A)	(B)	(C)
4	103.28	66.81	86.28	93.53
8	71.59	38.72	60.27	44.02
16	39.61	26.57	35.70	23.68
32	18.68	21.80	18.82	15.71
64	12.67	20.09	12.55	12.73
128	14.22	19.38	14.14	12.35
256	8.81	19.05	8.12	7.83
512	4.62	17.73	4.62	4.53
1024	2.35	11.49	2.66	2.89

Experimental results with $\sigma = 32$

m	BOM	(A)	(B)	(C)
4	78.76	55.23	57.75	88.57
8	51.68	30.37	42.03	39.84
16	35.40	19.92	30.18	20.34
32	20.62	16.12	19.34	12.20
64	12.11	14.84	11.55	10.63
128	12.60	15.63	11.26	10.01
256	7.58	16.73	6.32	5.90
512	4.29	17.90	3.73	3.83
1024	2.87	14.19	2.67	2.79

Experimental results with $\sigma = 64$

m	BOM	(A)	(B)	(C)
4	64.84	50.93	42.34	88.52
8	39.35	27.44	29.29	38.84
16	26.09	17.12	22.03	20.07
32	19.45	14.09	17.11	11.81
64	13.15	13.58	12.28	10.37
128	13.11	17.67	10.86	9.76
256	6.25	18.04	5.79	5.55
512	2.91	18.00	3.12	5.32
1024	2.71	16.89	2.58	2.42

(C)

$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
    $j = j + m - 1$
    $q = \delta(m, t_j)$
    if $q \neq \perp$ **then**
        $p = \delta(q, t_{j-1})$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

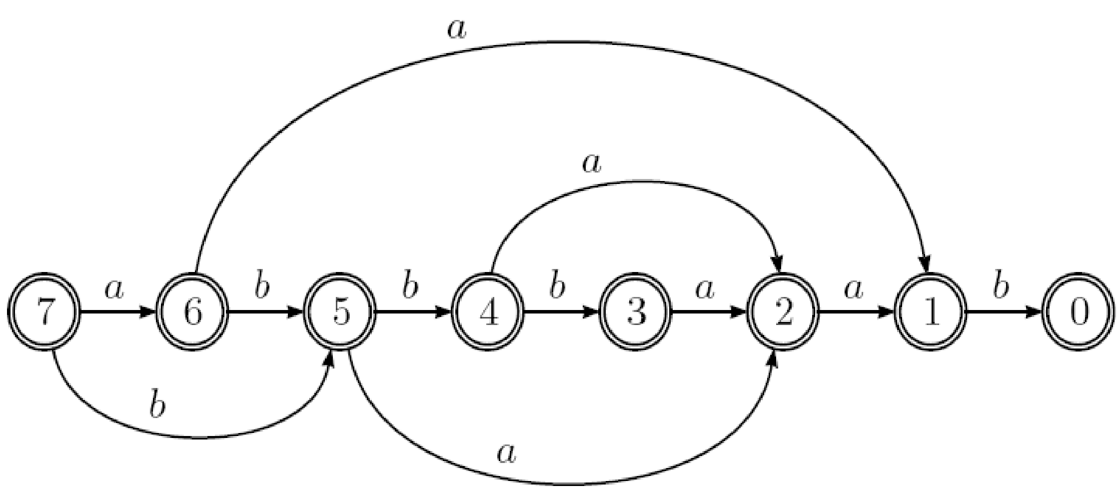
(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
    $j = j + m - 1$
    $q = \lambda(t_j, t_{j-1})$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$



(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
  $j = j + m - 1$
  $q = \lambda(t_j, t_{j-1})$

λ	a	b	c
a	1	5	$\perp$
b	2	4	$\perp$
c	$\perp$	$\perp$	$\perp$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
$\quad j = j + m - 1$
$\quad q = \lambda(t_j, t_{j-1})$

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a

λ	a	b	c
a	1	5	$\perp$
b	2	4	$\perp$
c	$\perp$	$\perp$	$\perp$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c

p	b	a	a	b	b	b	a

λ	a	b	c
a	1	5	$\boxed{\perp}$
b	2	4	$\perp$
c	$\perp$	$\perp$	$\perp$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
  $j = j + m - 1$
  $q = \lambda(t_j, t_{j-1})$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
    $j = j + m - 1$
    $q = \lambda(t_j, t_{j-1})$

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c
p							b	a	a	b	b	b	a			

λ	a	b	c
a	1	5	$\perp$
b	2	4	$\perp$
c	$\perp$	$\perp$	$\perp$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
    $j = j + m - 1$
    $q = \lambda(t_j, t_{j-1})$

t	a	c	c	a	a	c	a	b	a	a	b	a	b	a	a	c
p							b	a	a	b	b	b	a			

λ	a	b	c
a	1	5	$\perp$
b	$\boxed{2}$	4	$\perp$
c	$\perp$	$\perp$	$\perp$

A fast-loop over two transitions

We could encapsulate the two first transitions of the oracle in the function

$$\lambda(a,b) = \begin{cases} \perp & \text{if } \delta(m,a) = \perp \\ \delta(\delta(m,a),b) & \text{otherwise.} \end{cases}$$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
  $j = j + m - 1$
  $q = \lambda(t_j, t_{j-1})$

Experimental results with $\sigma = 8$

m	BOM	(A)	(B)	(C)	(D)
4	157.62	95.95	135.95	109.03	55.35
8	85.48	58.66	78.70	58.63	34.16
16	43.04	43.36	43.00	37.15	26.82
32	26.63	35.00	28.29	25.93	21.25
64	17.39	28.05	17.13	17.00	14.42
128	15.28	23.68	15.75	15.87	12.87
256	10.79	19.86	9.60	9.76	8.53
512	6.18	14.29	6.11	5.76	4.76
1024	3.29	8.20	3.45	3.35	2.64

Experimental results with $\sigma = 16$

m	BOM	(A)	(B)	(C)	(D)
4	103.28	66.81	86.28	93.53	40.63
8	71.59	38.72	60.27	44.02	21.73
16	39.61	26.57	35.70	23.68	14.91
32	18.68	21.80	18.82	15.71	12.73
64	12.67	20.09	12.55	12.73	12.49
128	14.22	19.38	14.14	12.35	10.38
256	8.81	19.05	8.12	7.83	6.88
512	4.62	17.73	4.62	4.53	3.60
1024	2.35	11.49	2.66	2.89	2.67

Experimental results with $\sigma = 32$

m	BOM	(A)	(B)	(C)	(D)
4	78.76	55.23	57.75	88.57	37.44
8	51.68	30.37	42.03	39.84	18.59
16	35.40	19.92	30.18	20.34	12.29
32	20.62	16.12	19.34	12.20	11.58
64	12.11	14.84	11.55	10.63	11.10
128	12.60	15.63	11.26	10.01	7.46
256	7.58	16.73	6.32	5.90	3.79
512	4.29	17.90	3.73	3.83	3.20
1024	2.87	14.19	2.67	2.79	2.01

Experimental results with $\sigma = 64$

m	BOM	(A)	(B)	(C)	(D)
4	64.84	50.93	42.34	88.52	37.04
8	39.35	27.44	29.29	38.84	17.99
16	26.09	17.12	22.03	20.07	11.57
32	19.45	14.09	17.11	11.81	10.76
64	13.15	13.58	12.28	10.37	10.70
128	13.11	17.67	10.86	9.76	6.35
256	6.25	18.04	5.79	5.55	3.60
512	2.91	18.00	3.12	5.32	1.98
1024	2.71	16.89	2.58	2.42	1.57

$\textsc{Extended-BOM}(p, m, t, n)$

$\delta \leftarrow \text{precompute-factor-oracle}(p)$
$\textbf{for } a \in \Sigma \textbf{ do}$
$\quad q \leftarrow \delta(m, a)$
$\quad \textbf{for } b \in \Sigma \textbf{ do}$
$\quad\quad \textbf{if } q = \perp \textbf{ then } \lambda(a, b) \leftarrow \perp$
$\quad\quad \textbf{else } \lambda(a, b) \leftarrow \delta(q, b)$
$t[n \mathinner{..} n + m - 1] \leftarrow p$
$j \leftarrow m - 1$
$\textbf{while } j < n \textbf{ do}$
$\quad q \leftarrow \lambda(t[j], t[j-1])$
$\quad \textbf{while } q = \perp \textbf{ do}$
$\quad\quad j \leftarrow j + m - 1$
$\quad\quad q \leftarrow \lambda(t[j], t[j-1])$
$\quad i \leftarrow j - 2$
$\quad \textbf{while } q \neq \perp \textbf{ do}$
$\quad\quad q \leftarrow \delta(q, t[i])$
$\quad\quad i \leftarrow i - 1$
$\quad \textbf{if } i < j - m + 1 \textbf{ then}$
$\quad\quad \text{output}(j)$
$\quad\quad i \leftarrow \imath + 1$
$\quad j \leftarrow j + i + m$

EXTENDED-BOM(p, m, t, n)

```
δ ← precompute-factor-oracle(p)
for a ∈ Σ do
    q ← δ(m, a)
    for b ∈ Σ do
        if q = ⊥ then λ(a, b) ← ⊥
        else λ(a, b) ← δ(q, b)
t[n .. n + m − 1] ← p
j ← m − 1
while j < n do
    q ← λ(t[j], t[j − 1])
    while q = ⊥ do
        j ← j + m − 1
        q ← λ(t[j], t[j − 1])
    i ← j − 2
    while q ≠ ⊥ do
        q ← δ(q, t[i])
        i ← i − 1
    if i < j − m + 1 then
        output(j)
        i ← ı + 1
    j ← j + i + m
```

Preprocessing of lambda

Searching phase

```
EXTENDED-BOM(p, m, t, n)
    δ ← precompute-factor-oracle(p)
    for a ∈ Σ do
        q ← δ(m, a)
        for b ∈ Σ do
            if q = ⊥ then λ(a, b) ← ⊥
            else λ(a, b) ← δ(q, b)
    t[n .. n + m − 1] ← p
    j ← m − 1
    while j < n do
        q ← λ(t[j], t[j − 1])
        while q = ⊥ do
            j ← j + m − 1
            q ← λ(t[j], t[j − 1])
        i ← j − 2
        while q ≠ ⊥ do
            q ← δ(q, t[i])
            i ← i − 1
        if i < j − m + 1 then
            output(j)
            i ← i + 1
        j ← j + i + m
```

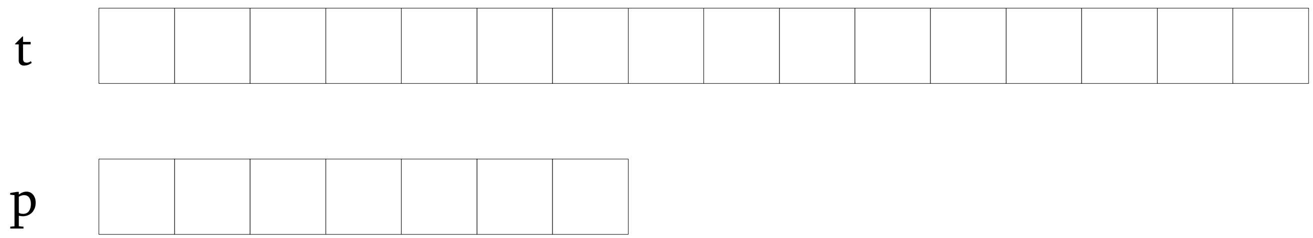
Preprocessing of lambda

fast-loop

Searching phase

Looking for the forward character

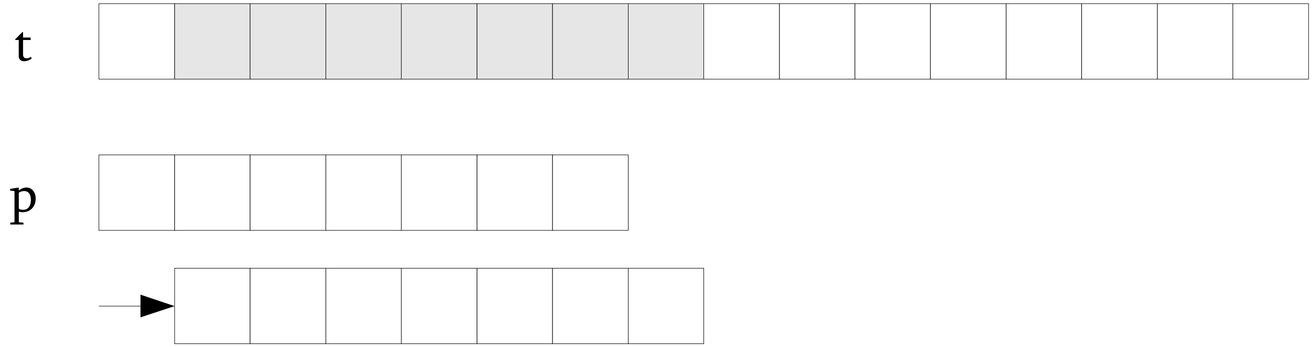
The idea of looking for the forward character for shifting has been originally introduced by Sunday in the **Quick-Search** algorithm **[5]** and then efficiently implemented in the **Forward-Fast-Search** algorithm **[6]** and in the **Shift-And-Sunday** algorithm **[7]**.



[5] D. M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, 1990.
[6] D. Cantone and S. Faro. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. J. Autom. Lang. Comb., 10(5/6):589–608, 2005.
[7] W. F. Smith, S. Wang and M. Yu. An Adaptative Hybrid Pattern-Matching Algorithm on Indeterminate Strings. Prague Stringology Conference 2008

Looking for the forward character

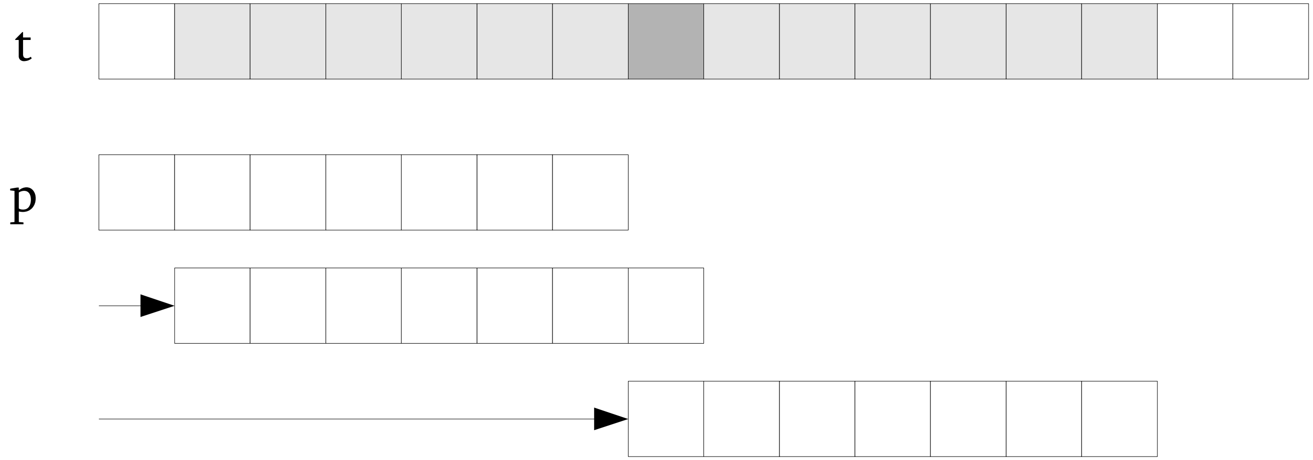
The idea of looking for the forward character for shifting has been originally introduced by Sunday in the **Quick-Search** algorithm **[5]** and then efficiently implemented in the **Forward-Fast-Search** algorithm **[6]** and in the **Shift-And-Sunday** algorithm **[7]**.



[5] D. M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, 1990.
[6] D. Cantone and S. Faro. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. J. Autom. Lang. Comb., 10(5/6):589–608, 2005.
[7] W. F. Smith, S. Wang and M. Yu. An Adaptative Hybrid Pattern-Matching Algorithm on Indeterminate Strings. Prague Stringology Conference 2008

Looking for the forward character

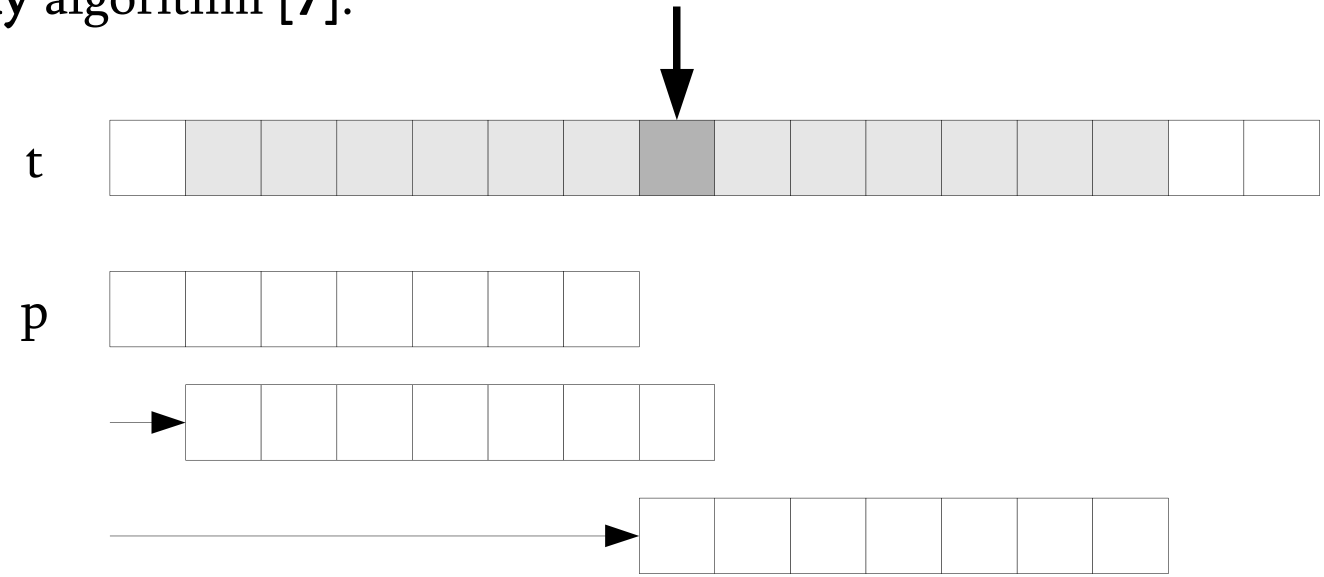
The idea of looking for the forward character for shifting has been originally introduced by Sunday in the **Quick-Search** algorithm **[5]** and then efficiently implemented in the **Forward-Fast-Search** algorithm **[6]** and in the **Shift-And-Sunday** algorithm **[7]**.



[5] D. M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, 1990.
[6] D. Cantone and S. Faro. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. J. Autom. Lang. Comb., 10(5/6):589–608, 2005.
[7] W. F. Smith, S. Wang and M. Yu. An Adaptative Hybrid Pattern-Matching Algorithm on Indeterminate Strings. Prague Stringology Conference 2008

Looking for the forward character

The idea of looking for the forward character for shifting has been originally introduced by Sunday in the **Quick-Search** algorithm **[5]** and then efficiently implemented in the **Forward-Fast-Search** algorithm **[6]** and in the **Shift-And-Sunday** algorithm **[7]**.



[5] D. M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, 1990.
[6] D. Cantone and S. Faro. Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm. J. Autom. Lang. Comb., 10(5/6):589–608, 2005.
[7] W. F. Smith, S. Wang and M. Yu. An Adaptative Hybrid Pattern-Matching Algorithm on Indeterminate Strings. Prague Stringology Conference 2008

The Forward-Factor-Oracle

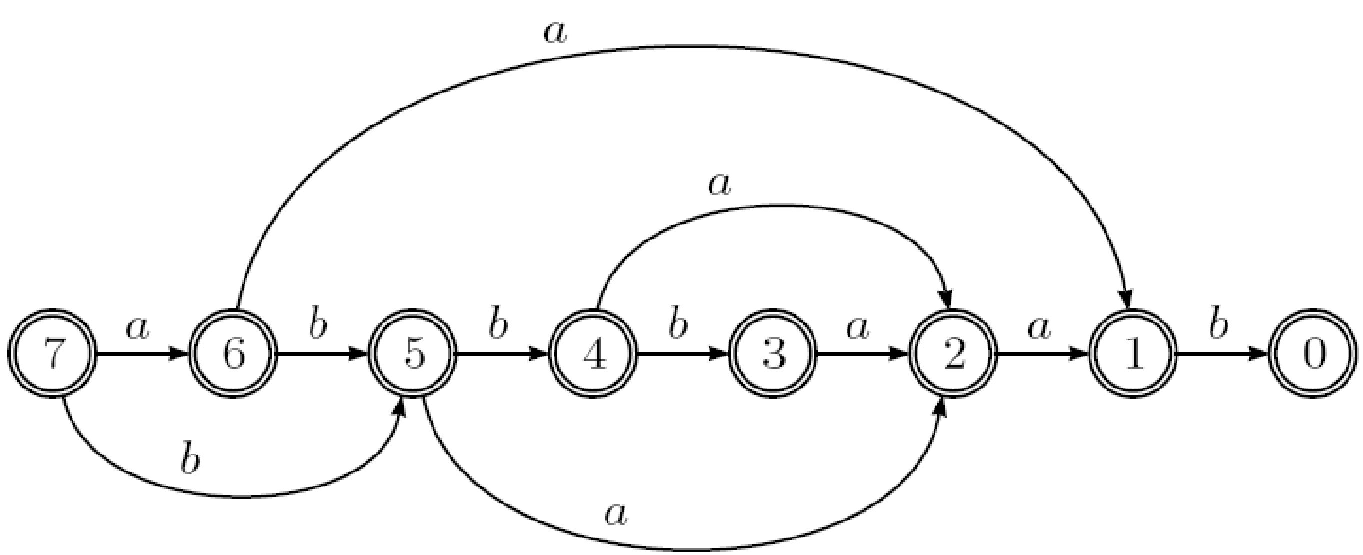
The forward factor oracle of a word p, FOracle(p), is an automaton which recognizes at least all the factors of p, eventually preceded by a word $x \in \Sigma \cup \{\varepsilon\}$. More formally the language recognized by FOracle(p) is defined by

$$\mathcal{L}(FOracle(p)) = \{xw \mid x \in \Sigma \cup \{\varepsilon\} \text{ and } w \in \mathcal{L}(Oracle(p))\}$$

The Forward-Factor-Oracle

Suppose Oracle(p) = {Q,m,Q,δ,Σ}, for a pattern p of length m.
FOracle(p) is an automaton {Q′, (m + 1),Q,Σ,δ′}, where

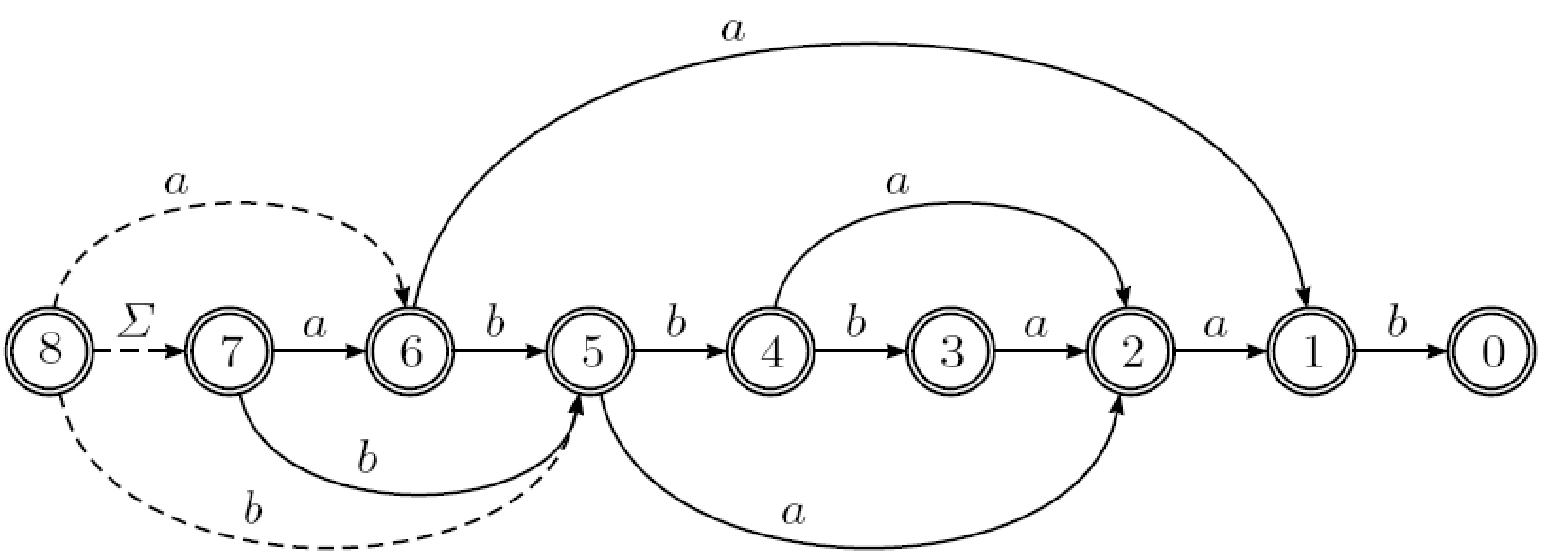
1. $Q' = Q \cup \{(m + 1)\}$
2. $(m + 1)$ is the initial state
3. all states are final
4. $\delta'(q, c) = \delta(q, c)$ for all $c \in \Sigma$, if $q \neq (m + 1)$
5. $\delta'(m + 1, c) = \{m, \delta(m, c)\}$ for all $c \in \Sigma$



The Forward-Factor-Oracle

Suppose Oracle(p) = {Q,m,Q,δ,Σ}, for a pattern p of length m.
FOracle(p) is an automaton {Q′, (m + 1),Q,Σ,δ′}, where

1. $Q' = Q \cup \{(m + 1)\}$
2. (m + 1) is the initial state
3. all states are final
4. $\delta'(q, c) = \delta(q, c)$ for all $c \in \Sigma$, if $q \neq (m + 1)$
5. $\delta'(m + 1, c) = \{m, \delta(m, c)\}$ for all $c \in \Sigma$



The Forward-BOM algorithm

The simulation of the forward factor oracle can be done by simply changing the computation of the table in the following way

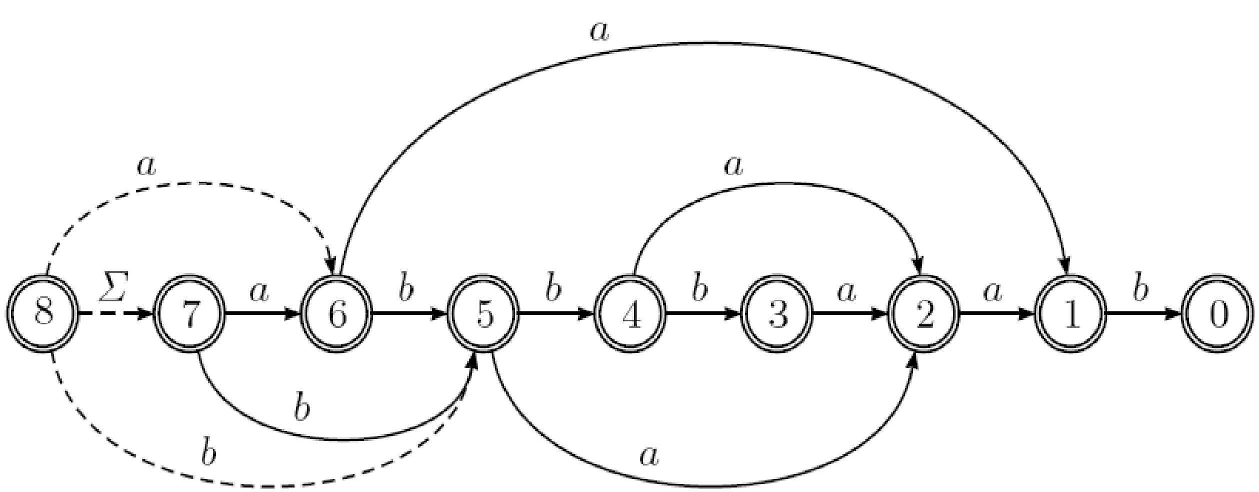
$$\lambda(a,b) = \begin{cases} \delta(m,b) & \text{if } \delta(m,a) = \perp \;\vee\; b = p[m-1] \\ \delta(\delta(m,a),b) & \text{otherwise} \end{cases}$$

The Forward-BOM algorithm

The simulation of the forward factor oracle can be done by simply changing the computation of the table in the following way

$$\lambda(a,b) = \begin{cases} \delta(m,b) & \text{if } \delta(m,a) = \perp \ \vee \ b = p[m-1] \\ \delta(\delta(m,a),b) & \text{otherwise} \end{cases}$$

λ	a	b	c
a	6	5	$\perp$
b	6	4	$\perp$
c	6	$\perp$	$\perp$



$\textsc{Forward-Bom}(p, m, t, n)$

$\delta \leftarrow$ precompute-factor-oracle(p)
for $a \in \Sigma$ **do**
  $q \leftarrow \delta(m, a)$
  for $b \in \Sigma$ **do**
    if $q = \perp$ **then** $\lambda(a, b) \leftarrow \perp$
    else $\lambda(a, b) \leftarrow \delta(q, b)$
$q \leftarrow \delta(m, p[m-1])$
for $a \in \Sigma$ **do** $\lambda(a, p[m-1]) \leftarrow q$
$t[n \,..\, n+m-1] \leftarrow p$
$j \leftarrow m - 1$
while $j < n$ **do**
  $q \leftarrow \lambda(t[j+1], t[j])$
  while $q = \perp$ **do**
    $j \leftarrow j + m$
    $q \leftarrow \lambda(t[j+1], t[j])$
  $i \leftarrow j - 1$
  while $q \neq \perp$ **do**
    $q \leftarrow \delta(q, t[i])$
    $i \leftarrow i - 1$
  if $i < j - m + 1$ **then**
    output(j)
    $i \leftarrow \imath + 1$
  $j \leftarrow j + i + m$

FORWARD-BOM(p, m, t, n)

```
δ ← precompute-factor-oracle(p)
for a ∈ Σ do
    q ← δ(m, a)
    for b ∈ Σ do
        if q = ⊥ then λ(a, b) ← ⊥
        else λ(a, b) ← δ(q, b)
q ← δ(m, p[m − 1])
for a ∈ Σ do λ(a, p[m − 1]) ← q
t[n .. n + m − 1] ← p
j ← m − 1
while j < n do
    q ← λ(t[j + 1], t[j])
    while q = ⊥ do
        j ← j + m
        q ← λ(t[j + 1], t[j])
    i ← j − 1
    while q ≠ ⊥ do
        q ← δ(q, t[i])
        i ← i − 1
    if i < j − m + 1 then
        output(j)
        i ← ı + 1
    j ← j + i + m
```

Preprocessing of lambda

Searching phase

FORWARD-BOM(p, m, t, n)

$\delta \leftarrow$ precompute-factor-oracle(p)
for $a \in \Sigma$ **do**
  $q \leftarrow \delta(m, a)$
  for $b \in \Sigma$ **do**
    if $q = \perp$ **then** $\lambda(a, b) \leftarrow \perp$
    else $\lambda(a, b) \leftarrow \delta(q, b)$

Preprocessing of lambda

$q \leftarrow \delta(m, p[m-1])$
for $a \in \Sigma$ **do** $\lambda(a, p[m-1]) \leftarrow q$
$t[n \,..\, n+m-1] \leftarrow p$
$j \leftarrow m - 1$
while $j < n$ **do**
  $q \leftarrow \lambda(t[j+1], t[j])$
  while $q = \perp$ **do**
    $j \leftarrow j + m$
    $q \leftarrow \lambda(t[j+1], t[j])$

fast-loop

  $i \leftarrow j - 1$
  while $q \neq \perp$ **do**
    $q \leftarrow \delta(q, t[i])$
    $i \leftarrow i - 1$
  if $i < j - m + 1$ **then**
    output(j)
    $i \leftarrow \imath + 1$
  $j \leftarrow j + i + m$

Searching phase

Experimental Results

- Extended-BOM algorithm (EBOM)
- Forward-BOM algorihtm (FBOM)
- Forward-SBNDM algorithm (FSBNDM)
- Fast-Search algorithm (FS)
- Forward-Fast-Search algorithm (FFS)
- BOM algorithm (BOM)
- q-Hash algorithms with q=3,5,8 (3-HASH, 5-HASH, 8-HASH)
- SBNDM algorihtm (SBNDM)

Experimental Results

- All algorithms have been implemented in the C programming language and were used to search for the same strings in large fixed text buffers

- The algorithms have been tested
    - on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64$;
    - on a genome sequence (Escherichia Coli);
    - on a protein sequence (from human genome);

- Searching have been performed for patterns of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$, and 1024.

- In the following tables, running times are expressed in hundredths of seconds.

Experimental Results

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	153.52	**129.07**	209.07	169.22	177.31	162.98	-	-	155.38	145.47
8	115.44	94.42	133.73	105.08	114.17	77.03	**67.91**	-	87.42	80.72
16	83.60	63.05	71.75	58.91	63.23	51.65	**25.27**	34.33	44.87	41.31
32	61.96	43.40	38.55	30.58	33.24	45.38	14.85	**13.50**	23.88	20.77
64	48.16	32.69	21.24	17.43	17.91	44.65	11.53	**7.42**	-	-
128	39.55	24.90	11.91	11.73	15.63	44.02	10.09	**8.34**	-	-
256	32.80	21.14	8.45	8.43	10.00	44.92	11.02	**6.86**	-	-
512	28.07	17.27	6.36	**4.87**	5.87	45.65	10.04	6.21	-	-
1024	23.39	15.47	4.00	**2.79**	3.95	44.72	10.59	5.14	-	-

Running times for a Rand2 problem

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	82.12	78.03	111.55	**58.93**	84.93	117.82	-	-	60.57	74.73
8	60.00	54.02	61.31	43.57	51.38	43.77	56.23	-	**40.46**	40.79
16	49.05	39.49	35.58	29.11	31.66	22.40	**20.54**	33.98	23.49	23.15
32	41.72	30.56	19.98	16.88	18.13	16.27	**10.60**	12.70	12.97	12.48
64	37.11	23.71	11.63	9.79	11.11	13.53	**7.05**	7.11	-	-
128	32.02	18.43	8.30	7.56	10.20	12.17	**7.05**	8.12	-	-
256	28.54	15.72	6.27	**5.72**	6.16	12.25	6.97	6.99	-	-
512	26.07	14.13	3.52	**3.31**	3.67	12.10	7.46	5.71	-	-
1024	22.14	12.97	**1.83**	2.25	2.78	11.46	8.02	4.76	-	-

Running times for a Rand4 problem

Experimental Results

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	46.34	44.32	78.94	**26.81**	49.87	105.12	-	-	33.82	40.05
8	29.61	27.46	43.23	**16.85**	30.11	37.30	54.47	-	18.70	23.09
16	22.46	20.67	21.72	13.34	19.01	18.07	18.90	33.90	**12.81**	14.71
32	19.97	16.91	13.70	10.15	12.29	10.89	9.92	13.14	9.92	**9.73**
64	18.93	14.14	8.70	**7.07**	7.95	8.71	7.87	7.09	-	-
128	17.85	12.10	6.99	**6.66**	7.85	7.11	7.81	7.98	-	-
256	17.15	11.13	5.26	**3.56**	4.70	7.68	6.48	7.43	-	-
512	16.02	11.29	3.25	2.61	**2.38**	7.75	6.53	6.03	-	-
1024	15.35	9.63	1.88	**1.55**	1.61	6.91	6.56	5.57	-	-

Running times for a Rand8 problem

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	33.17	32.13	52.02	**20.09**	39.26	102.31	-	-	28.16	27.98
8	18.52	18.91	35.48	**10.73**	21.87	34.74	54.09	-	14.04	15.22
16	13.48	13.01	19.61	**6.98**	13.76	16.33	18.71	33.78	7.66	9.18
32	11.41	10.83	9.33	**6.36**	8.29	9.46	8.64	13.35	6.80	6.43
64	10.54	9.57	6.74	**5.58**	7.12	6.79	6.21	7.29	-	-
128	10.39	9.14	7.58	**5.05**	9.99	6.25	8.52	7.93	-	-
256	9.88	9.08	5.00	**3.16**	4.45	6.84	6.98	7.07	-	-
512	10.23	9.10	2.55	**2.18**	2.61	6.22	5.90	6.44	-	-
1024	10.14	8.55	1.57	**1.18**	1.45	6.33	5.40	5.62	-	-

Running times for a Rand16 problem

Experimental Results

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	28.04	26.91	35.98	**19.03**	35.98	100.51	-	-	26.88	23.75
8	15.51	15.23	24.54	**8.98**	20.74	34.34	53.71	-	12.28	12.54
16	9.78	9.44	17.46	**6.18**	11.56	15.44	18.36	34.14	6.95	7.46
32	8.29	7.98	10.26	**5.46**	7.11	8.36	9.02	13.16	5.59	5.75
64	7.50	7.35	5.78	**5.58**	6.37	6.37	6.22	7.07	-	-
128	7.38	7.70	6.21	**3.36**	10.62	7.58	8.21	8.32	-	-
256	7.59	8.33	3.62	**2.38**	5.94	6.73	6.95	6.75	-	-
512	7.89	8.91	1.96	**1.41**	3.28	6.28	5.78	6.40	-	-
1024	7.84	7.73	1.57	1.45	**1.39**	5.91	5.31	5.83	-	-

Running times for a Rand32 problem

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	23.55	27.38	29.10	**18.79**	35.38	97.23	-	-	25.05	23.67
8	13.48	13.82	18.51	**8.76**	19.41	33.79	53.80	-	12.15	11.37
16	8.06	8.44	12.64	**5.69**	11.35	15.07	18.56	33.32	6.72	6.72
32	7.04	6.47	9.33	**5.14**	7.20	8.09	9.00	13.15	5.55	5.25
64	6.44	6.68	6.34	**5.16**	6.52	6.13	6.09	7.23	-	-
128	8.41	8.24	6.05	**3.84**	9.85	8.51	7.72	8.45	-	-
256	8.82	8.49	3.19	**1.96**	5.59	7.08	6.52	7.21	-	-
512	8.52	9.14	1.99	**1.28**	3.21	6.05	5.79	6.07	-	-
1024	8.60	8.36	2.41	**1.33**	1.64	6.25	4.10	5.67	-	-

Running times for a Rand64 problem

Experimental Results

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	18.64	16.91	23.25	**12.65**	19.09	25.48	-	-	12.96	17.30
8	13.85	11.63	13.04	10.27	11.40	9.90	12.34	-	**8.73**	9.01
16	11.48	8.47	7.73	6.77	6.47	4.76	**4.39**	7.74	5.28	5.50
32	9.58	6.44	4.53	3.52	4.07	3.20	2.77	2.85	3.04	**2.62**
64	8.56	4.92	2.50	1.95	2.42	2.65	**1.60**	1.84	-	-
128	7.05	4.01	1.74	**1.73**	1.91	2.42	1.84	2.08	-	-
256	6.41	3.35	1.33	**1.32**	1.33	2.90	1.60	1.41	-	-
512	5.66	3.20	0.94	0.82	**0.78**	2.39	1.60	1.61	-	-
1024	5.97	2.19	0.98	0.66	**0.51**	2.50	1.21	1.21	-	-

Running times for a genome sequence ($\sigma = 4$)

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	4.33	2.93	8.30	**2.14**	5.51	14.49	-	-	5.19	3.59
8	**1.68**	2.64	4.21	2.27	3.58	4.38	8.09	-	2.31	1.85
16	1.71	1.57	2.66	**1.05**	1.92	2.50	2.58	4.54	1.25	1.05
32	1.41	1.47	1.62	**0.87**	1.27	1.30	1.37	1.64	0.89	0.89
64	1.21	1.02	1.10	**0.63**	1.18	0.85	0.82	1.25	-	-
128	1.09	1.33	1.13	**0.67**	1.51	0.98	1.14	1.22	-	-
256	1.37	1.44	0.59	0.51	**0.47**	0.90	0.90	0.82	-	-
512	1.20	1.56	0.50	**0.27**	0.30	0.77	0.90	0.88	-	-
1024	1.25	1.64	0.39	0.35	**0.27**	0.87	0.70	0.74	-	-

Running times for a protein sequence ($\sigma = 22$)

Conclusions

We presented two efficient variants of the Backward Oracle Matching algorithm which is considered one of the most effective algorithm for exact string matching.
The first variation, called **Extended-BOM**, introduces an efficient fast-loop over transitions of the oracle by reading two consecutive characters for each iteration.
The second variation, called **Forward-BOM**, extends the previous one by using a look-ahead character at the beginning of transitions in order to obtain larger shift advancements.
It turns out from experimental results that the new proposed variations are very fast in practice and obtain the best results in most cases, especially for long patterns and alphabets of medium dimension.