Text Searching and Processing

Maxime Crochemore

King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Strings

- **Alphabet**: A (finite) set of letters, $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots\}$
- **Strings**: A^* set of finite sequences of letters (ε denotes the empty string)
- **Length of a string** x: $|x|$ = length of the sequence
- **Notation—array representation**: $x = x[0]x[1] \ldots x[|x| - 1]$

i	0	1	2	3	4	5	6	7	8
$x[i]$	b	a	b	a	a	b	a	b	a

- **Alphabet of a string**: $\text{alph}(x)$ set of letters occurring effectively in x; each letter of $\text{alph}(x)$ appears at least once in x
- **Equality**

$$x = y \text{ iff } |x| = |y| \text{ and } x[i] = y[i] \text{ for } i = 0, 1, \ldots, |x| - 1$$

Factors

- **Concatenation or product**: xy is sequence x followed by sequence y
- **Factor**: x factor of or occurs in y if y is a product uxv for two strings u, v
  x **prefix** of y if $y = xv$; x **suffix** of y if $y = ux$

i	0	1	2	3	4	5	6	7	8
$y[i]$	`b`	`a`	`b`	`a`	`a`	`b`	`a`	`b`	`a`
left positions of `aba`		1			4		6		
right positions of `aba`				3			6		8

- **Positions**: x occurs in y at (left) position i if $y = uxv$ and $|u| = i$
  equivalently $x = y[i]y[i+1] \ldots y[i+|x|-1] = y[i \,.\,.\, i+|x|-1]$
- **Positions of the first occurrence**:

$$pos(x) = \min\{|u| : uxA^* \cap yA^* \neq \emptyset\}$$

- **Subsequence**: x subsequence of y if $y = w_0 x[0] w_1 x[1] \ldots x[|x|-1] w_{|x|}$ for $|x|+1$ strings $w_0, w_1, \ldots, w_{|x|}$
  equivalently, x can be obtained from y by deleting $|y| - |x|$ letters

Powers

- **Power**: u^k is the kth power of u, defined by
  $u^0 = \varepsilon$ and $u^e = u^{e-1}u$ for $e = 1, 2, \ldots, k$

Lemma 1

If $x^m = y^n$ for integers $m, n > 0$, then x, y are powers of the same string.

- **Primitive string**: a (nonempty) string x is primitive if it is not the power of another string — equivalently $x = u^k$ implies $k = 1$, and then $x = u$
  `abaab` is primitive, while ε and `bababa` $=$ (`ba`)^3 are not

Lemma 2 (Primitivity Lemma)

x is primitive iff it is a factor of x^2 only as a prefix and as a suffix, that is, ux prefix of x^2 implies $u = \varepsilon$ or $u = x$

`abaab` occurs at positions $0, 5$ only in `abaababaab` $=$ (`abaab`)^2
`bababa` occurs at positions $0, 2, 4, 6$ in `babababababa` $=$ (`bababa`)^2

Proofs as exercises — consequences of the Periodicity Lemma

Root and Conjugates

- **Root of** x: unique primitive u for which $x = u^k$

  Proposition 3

  There exists one and only one primitive string which $x \neq \varepsilon$ is a power of.

  `abaab` root of itself

  `ba` root of `bababa`

- **Conjugates**: x, y are conjugates if $x = uv$ and $y = vu$

  `abaab` has $5 = |$`abaab`$|$ conjugates: `abaab`, `baaba`, `aabab`, `ababa`, `babaa`

  `bababa` has $2 = |$`ba`$|$ conjugates: `bababa`, `ababab`

  Proposition 4

  x, y are conjugate if and only if their roots are conjugate.

  Proposition 5

  x, y are conjugate if and only if there exists a string z such that $xz = zy$.

Text Searching

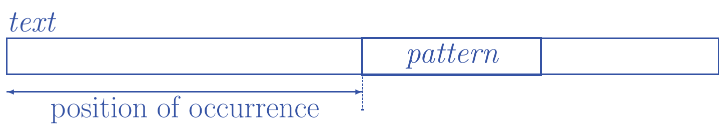
Maxime Crochemore

King's College London

Maxime.Crochemore@kcl.ac.uk

http://www.dcs.kcl.ac.uk/staff/mac/

Pattern matching



- **Problem**

  Find all the occurrences of pattern x of length m
  inside the text y of length n

- **Two types of solutions**

  – Fixed pattern — preprocessing time $O(m)$
    Use of combinatorial properties — searching time $O(n)$

  – Static text — preprocessing time $O(n)$
    Solutions based on indexes — searching time $O(m)$

Searching for a fixed string

- ⋆ **String matching**
  given a pattern x, find all its locations in any text y
- ⋆ **Pattern**: a string x of symbols, of length m

  t a t a
- ⋆ **Text**: a string y of symbols, of length n

  c a c g t a t a t a t g c g t t a t a a t
- ⋆ **Occurrences** at positions 4; 6, 15:

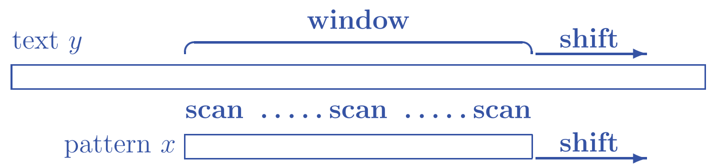
```
c a c g t a t a t a t g c g t t a t a a t
        t a t a               t a t a
            t a t a
```

- ⋆ **Basic operation**: symbol comparison $(=, \neq)$

Interest

- ⋆ **Practical**
  basic problem for
  - search for various patterns
  - lexical analysis
  - approximate string matching
  - comparisons of strings—alignments
  - ...
- ⋆ **Theoretical**
  - design of algorithms
  - analysis of algorithms—complexity
  - combinatorics on strings
  - ...

Sliding window strategy



⋆ Scan-and-shift mechanism

---

put window at the beginning of text
while window on text **do**
  scan: **if** window $=$ pattern **then** report it
  shift: shift window to the right and
  memorize some information for use during next scans and shifts

---

Naive search

c a c g t a t a t a t g c g t t a t a a
t a t a

Principles

⋆ No memorization, shift by 1 position

Complexity

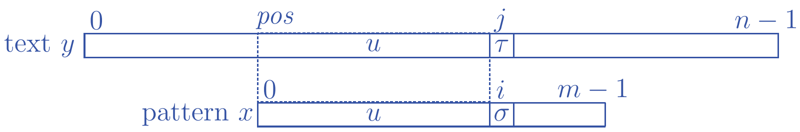
⋆ $O(m \times n)$ time, $O(1)$ extra space

Number of symbol comparisons

⋆ maximum $\approx m \times n$

⋆ expected $\approx 2 \times n$
on a two-letter alphabet,
with equiprobablity and independence conditions

Naive string-searching algorithm



NAIVE_SEARCH(string x, y; integer m, n)
 $pos \longleftarrow 0$
 while $pos \leq n - m$ **do**
  $i \longleftarrow 0$
  while $i < m$ **and** $x[i] = y[pos + i]$ **do**
   $i \longleftarrow i + 1$
  if $i = m$ **then** output('x occurs in y at position ', pos)
  $pos \longleftarrow pos + 1$

Acceleration by hashing

⋆ Hash function: $h : \mathbf{\Sigma}^m \longrightarrow \mathbf{N}$

ACCELERATED_SEARCH(string x, y; integer m, n; h)
 $h_x \longleftarrow h(x)$
 for $pos \longleftarrow 0$ **to** $n - m$ **do**
  if $h_x = h(y[pos \,..\, pos + m - 1])$ **then**
   $i \longleftarrow 0$
   while $i < m$ **and** $x[i] = y[pos + i]$ **do**
    $i \longleftarrow i + 1$
   if $i = m$ **then** output('x occurs in y at position ', pos)

⋆ Uses arithmetic operations in addition to symbol comparisons

⋆ **What hash function to speed-up the algorithm?**

⋆ **Goal:** $h(u) = h(v)$ **if it is very likely that** $u = v$

Hash function

- Hash Function: $h : \mathbf{\Sigma}^m \longrightarrow \mathbf{N}$
- Principle: do as if symbols are integers
  similar to number representation but with approximation
- Parameters: integers d (like a base) and q (for the modulo)
- Definition:

$$\begin{aligned} h_{pos} &= h(y[pos\,..\,pos+m-1]) \\ &= (y[pos]d^{m-1} + y[pos+1]d^{m-2} + \cdots + y[pos+m-1]) \ \mathbf{mod}\ q \\ &= ((\ldots(y[pos]d + y[pos+1])d + \cdots + y[pos+m-2])d \\ &\qquad + y[pos+m-1]) \ \mathbf{mod}\ q \end{aligned}$$

- Hörner's rule for the first hash value

Next hash value

- From h_{pos} to h_{pos+1}:

$$\begin{aligned} h_{pos} &= (y[pos]d^{m-1} + y[pos+1]d^{m-2} + \cdots + y[pos+m-1]) \ \mathbf{mod}\ q \\ h_{pos+1} &= (y[pos+1]d^{m-1} + y[pos+2]d^{m-2} + \cdots + y[pos+m]) \ \mathbf{mod}\ q \\ &= ((h_{pos} - y[pos]d^{m-1})d + y[pos+m]) \ \mathbf{mod}\ q \end{aligned}$$

- **It requires a fixed number of arithmetic operations**
- **. . . then executes in constant time**

Karp-Rabin string searching

⋆ Typical parameters: $d = 2^k$ and q is a prime number

KR(string x, y; integer m, n, d, q)
  $(h_x, h_y, D) \longleftarrow (0,\ 0,\ d^{m-1}$ **mod** $q)$
  for $i \longleftarrow 0$ to $m-1$ **do**
    $h_x \longleftarrow (h_x d + x[i])$ **mod** q
    $h_y \longleftarrow (h_y d + y[i])$ **mod** q
  for $pos \longleftarrow 0$ **to** $n-m$ **do**
    if $h_x = h_y$ **then**
      $i \longleftarrow 0$
      while $i < m$ **and** $x[i] = y[pos+i]$ **do**
        $i \longleftarrow i+1$
      if $i = m$ **then** output('x occurs in y at position ', pos)
    if $pos < n-m$ **then**
      $h_y \longleftarrow ((h_y - x[pos]D)d + y[pos+m])$ **mod** q

---

Complexity of the problem

pattern x of length m
text y of length n $(n > m)$

Theorem 1 *The search can be done optimally in time $O(n)$ and space $O(1)$.*

[Galil and Seiferas, 1983]

Theorem 2 *The search can be done in optimal expected time $O(\frac{\log m}{m} \times n)$.*

[Yao, 1979]

Theorem 3 *The maximal number of comparisons done during the search is $\geq n + \frac{9}{4m}(n-m)$, and can be made $\leq n + \frac{8}{3(m+1)}(n-m)$.*

[Cole *et alii*, 1995]

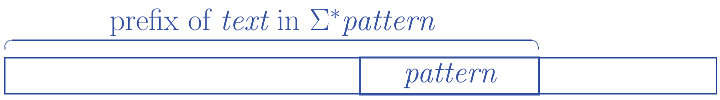
Known bounds on symbol comparisons

	Lower bounds	Upper bounds	
⋆	**access to the whole text**		
	n		[Folklore]
		$2n-1$	**[Morris and Pratt, 1970]**
	$\frac{4}{3}n$		**[Zwick and Paterson, 1992]**
⋆	**search with a sliding window of size** m		
		$\frac{3}{2}n$	**[Apostolico and Crochemore, 1989]**
	$\frac{4}{3}n$	$\frac{4}{3}n$	**[Galil and Giancarlo, 1991]**
	$\frac{4}{3}n - \frac{1}{3}m$		**[Galil and Giancarlo, 1992]**
		$n + \frac{4\log m+2}{m}(n-m)$	**[Galil and Breslauer, 1993]**
	$n + \frac{9}{4m}(n-m)$	$n + \frac{8}{3(m+1)}(n-m)$	**[Cole *et alii*, 1995]**

Known bounds on symbol comparisons (followed)

	Lower bounds	Upper bounds	
⋆	**search with a sliding window of size** 1		
		$2n-1$	**[Morris and Pratt, 1970]**
	$(2-\frac{1}{m})n$	$(2-\frac{1}{m})n$	**[Hancart, 1993]**
			[Breslauer *et alii*, 1993]
⋆	**delay = maximum number of comp's on each text symbol**		
		m	**[Morris and Pratt, 1970]**
		$\log_\Phi(m+1)$	**[Knuth, Morris and Pratt, 1977]**
		$\min\{\log_\Phi(m+1), \mathrm{card}\Sigma\}$	**[Simon, 1989]**
	$\min\{1+\log_2 m, \mathrm{card}\Sigma\}$	$\min\{1+\log_2 m, \mathrm{card}\Sigma\}$	**[Hancart, 1993]**
		$\log\,\min\{1+\log_2 m, \mathrm{card}\Sigma\}$	**[Hancart, 1996]**

Methods



- ⋆ **sequential searches** (window size = one symbol)
  - ↪ adapted to telecommunications
  - ↪ based on efficient implementations of automata

  [Knuth, Morris, Pratt, 1976], **[Simon, 1989]**,
  [Hancart, 1993], **[Breslauer, Colussi, Toniolo, 1993]**

Methods (followed)

- ⋆ **time-space optimal searches**
  - ↪ mainly of theoretical interest
  - ↪ based on combinatorial properties of strings

  [Galil, Seiferas, 1983], **[Crochemore, Perrin, 1991]**,
  [Crochemore, 1992], **[Gąsieniec, Plandowski, Rytter, 1995]**,
  [Crochemore, Gąsieniec, Rytter, 1997]

- ⋆ **practically-fast searches**
  - ↪ used in text editors, data retrieval software
  - ↪ based on combinatorics + automata (+ heuristics)

  [Boyer, Moore, 1977], **[Galil, 1979]**,
  [Apostolico, Giancarlo, 1986], **[Crochemore *et alii*, 1992]**

Sequential String Matching

Maxime Crochemore

King's College London

`Maxime.Crochemore@kcl.ac.uk`
`http://www.dcs.kcl.ac.uk/staff/mac/`

Examples

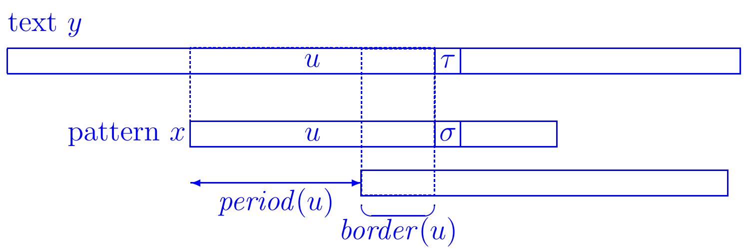
- Naive search (1)

```
a a a b a a a a b a b a a b a b . .
a a a b a a
  a a a b a a
    a a a b a a
      a a a b a a
        a a a b a a
          . . .
```

- Naive search (2)

```
a b a b a a a b a a b a a b a b . .
a b a a a b a b
  a b a a a b a b
    a b a a a b a b
      a b a a a b a b
        a b a a a b a b
          . . .
```

Left-to-right scan — shift



- ⋆ Mismatch situation: $\sigma \neq \tau$
- ⋆ $period(u) = |u| - |border(u)|$
- ⋆ Optimal shift length $= period(u\tau)$
- ⋆ Valid if $u = x$

Periods and borders

- ⋆ Non-empty string u, integer p, $0 < p \leq |u|$
- ⋆ p is a period of u if any of these equivalent conditions is satisfied:

  [1] $u[i] = u[i+p]$, for $0 \leq i < |u| - p$

  [2] u is a prefix of some y^k, $k > 0$, $|y| = p$

  [3] $u = yw = wz$, for some strings y, z, w with $|y| = |z| = p$
  String w is called **a border** of u
- ⋆ **The** period of u, $period(u)$, is its smallest period (can be $|u|$)
- ⋆ **The** border of u, $border(u)$, is its longest border (can be empty)
- ⋆ Periods and borders of `abacabacaba`

4	abacaba
8	aba
10	a
11	empty string

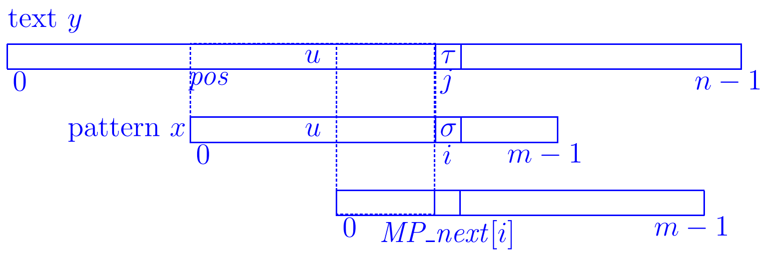
Sequential search

- ⋆ Simple online search
- ⋆ Length of shift = period
- ⋆ Memorization of borders

---

while window on text **do**
    $u \longleftarrow$ longest common prefix of window and pattern
    if $u =$ pattern **then** report a match
    shift window $\mathit{period}(u)$ places to right
    memorize $\mathit{border}(u)$

MP algorithm



MP(string x, y; integer m, n)
    $i \longleftarrow 0$; $j \longleftarrow 0$
    while $j < n$ **do**
        while $(i = m)$ **or** $(i \geq 0$ **and** $x[i] \neq y[j])$ **do**
            $i \longleftarrow MP\_next[i]$
        $i \longleftarrow i + 1$; $j \longleftarrow j + 1$
        if $i = m$ **then** output('x occurs in y at position', $j - i$)

Example of MP run

- MP_next table

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	c	a	b	a	c	a	b	
$MP\_next[i]$	-1	0	0	1	0	1	2	3	4	5	6

- Run of MP algorithm

```
y a b a b a c a b a d a b . .
x a b a c a b a c a b
      a b a c a b a c a b
              a b a c a b a c a b
                  a b a c a b a c a b
                    a b a c a b a c a b
                      a b a c a b a c a b
```

- If end of y, MP algorithm gives the longest overlap between y and x.

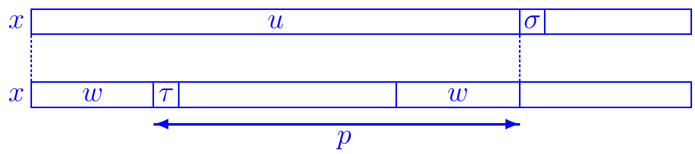
Computing borders of prefixes

- A border of a border of u is a border of u
  A border of u is either $border(u)$ or a border of it
- $\text{Border}[i] = |border(x[0\,..\,i-1])|$
- j runs through decreasing lengths of borders

COMPUTE_BORDERS(string x; integer m)
  Border[0] $\longleftarrow -1$
  for $i \longleftarrow 0$ **to** $m-1$ **do**
    $j \longleftarrow$ Border[i]
    while $j \geq 0$ **and** $x[i] \neq x[j]$ **do**
      $j \longleftarrow$ Border[j]
    Border[$i+1$] $\longleftarrow j+1$
  return Border

- $MP\_next[i] = \text{Border}[i]$ for $i = 0, \ldots, m$

Improvement



⋆ Interrupted periods — strict borders

⋆ Changes only the preprocessing of MP algorithm

---

while window on text **do**
  $u \longleftarrow$ longest common prefix of window and pattern
  if $u =$ pattern **then** report a match
  shift window $interrupt\_period(u)$ places to the right
  memorize $strict\_border(u)$

Interrupted periods and strict borders

⋆ Fixed string x, non-empty prefix u of x

⋆ w is a strict border of u if both:

  – w is a border of u

  – $w\tau$ is a prefix of x, but $u\tau$ is not

⋆ p is an interrupted period of u if $p = |u| - |w|$ for some strict border $|w|$ of u

⋆ Prefix `abacabacaba` of `abacabacabacc`

Interrupted periods and strict borders of `abacabacaba`

10	a
11	empty string

KMP preprocessing

- $k = MP\_next[i]$
- $KMP\_next[i] = \begin{cases} k, & \text{if } x[i] \neq x[k] \text{ or if } i = m, \\ KMP\_next[k], & \text{if } x[i] = x[k]. \end{cases}$

COMPUTE_KMP_NEXT(string x; integer m);
  $KMP\_next[0] \longleftarrow -1$; $k \longleftarrow 0$
  for $i \longleftarrow 1$ **to** $m-1$ **do** {here: $k = MP\_next[i]$}
    if $x[i] = x[k]$ **then**
      $KMP\_next[i] \longleftarrow KMP\_next[k]$
    else $KMP\_next[i] \longleftarrow k$
      do $k \longleftarrow KMP\_next[k]$
      while $k \geq 0$ **and** $x[i] \neq x[k]$
    $k \longleftarrow k + 1$
  $KMP\_next[m] \longleftarrow k$
  return KMP_next

Example of KMP run

- KMP_next table

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	c	a	b	a	c	a	b	
$MP\_next[i]$	-1	0	0	1	0	1	2	3	4	5	6
$KMP\_next[i]$	-1	0	-1	1	-1	0	-1	1	-1	0	6

- Run of KMP algorithm

```
y a b a b a c a b a d a b . .
x a b a c a b a c a b
      a b a c a b a c a b
                  a b a c a b a c a b
                    a b a c a b a c a b
                      a b a c a b a c a b
```

- If end of y, KMP algorithm gives the longest overlap between y and x.

Running times of MP and KMP

Theorem 1 *On a text of length n, MP and KMP string-searching algorithm run in time $O(n)$.*
They make less than $2n$ symbol comparisons.

Proof Positive comparisons increase the value of j
Negative comparisons increase the value of $j - i$ (shift)

* Delay = maximum number of comparisons on a text symbol

Theorem 2 *Pattern of length m. The delay for MP algorithm is no more than m. The delay for KMP algorithm is no more than $\log_\Phi(m+1)$, where Φ is the golden ratio, $(1+\sqrt{5})/2$.*

Proof For KMP, use the periodicity theorem

* A worst-case pattern of length 19: `abaababaabaababaaba`

Periodicities

Theorem 3 *If p and q are periods of a word x and satisfy $p + q - GCD(p, q) \leq |x|$ then $GCD(p, q)$ is a period of x.*

[Fine, Wilf, 1965]

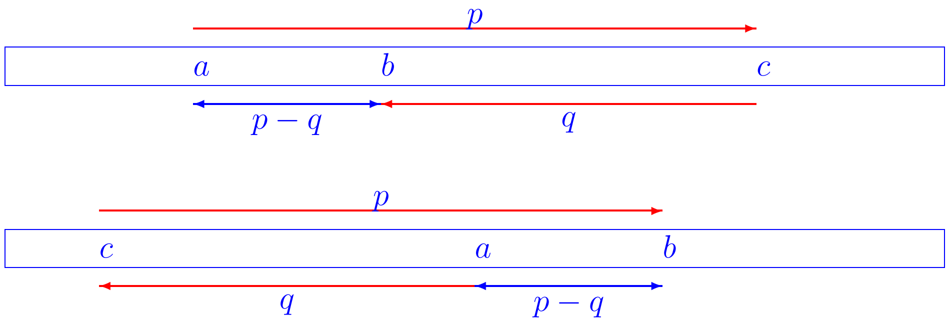
Used in the analysis of KMP algorithm and in the analysis of many other pattern matching algorithms.

Theorem 4 *(Weak version) If p and q are periods of a word x and satisfy $p + q \leq |x|$ then $GCD(p, q)$ is a period of x.*

Proof If p and q are periods of x, $p > q$, then $p - q$ is also a period of x. Rest of the proof analogous to correctness of Euclid's gcd algorithm.

Proof of the weak statement

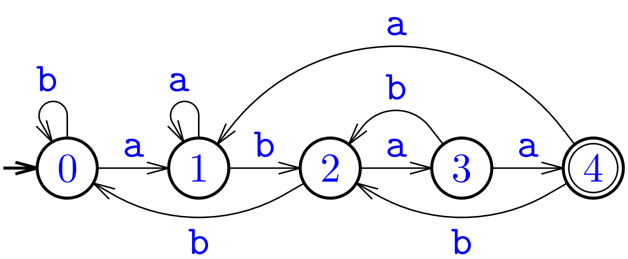
- ⋆ p and q periods of x with $p+q \le |x|$ and $p > q$
- ⋆ $p-q$ period of x because:



- ⋆ rest of the proof like Euclid's induction

Searching with an automaton

- ⋆ Uses the string-matching automaton $SMA(x)$:
  smallest deterministic automaton accepting $\Sigma^* x$
- ⋆ Example $x =$ `abaa`



- ⋆ Search for `abaa` in:

		b	a	b	b	a	a	b	a	a	b	a	a	b	b	a	···
state	0	0	1	2	0	1	1	2	3	4	2	3	4	2	0	1	···

Searching algorithm

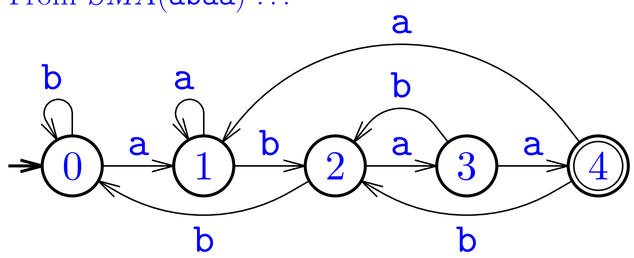
- Simple online parsing of the text with
  the string-matching automaton $SMA(x)$

SEARCH(string x, y; integer m, n)
    $(Q, \Sigma, initial, \{terminal\}, \delta)$ is the automaton $SMA(x)$
    $q \longleftarrow$ *initial state*
    if $q =$ *terminal* **then** report an occurrence of x in y
    while not end of y **do**
        $\sigma \longleftarrow$ next symbol of y
        $q \longleftarrow \delta(q, \sigma)$
        if $q =$ *terminal* **then** report an occurrence of x in y

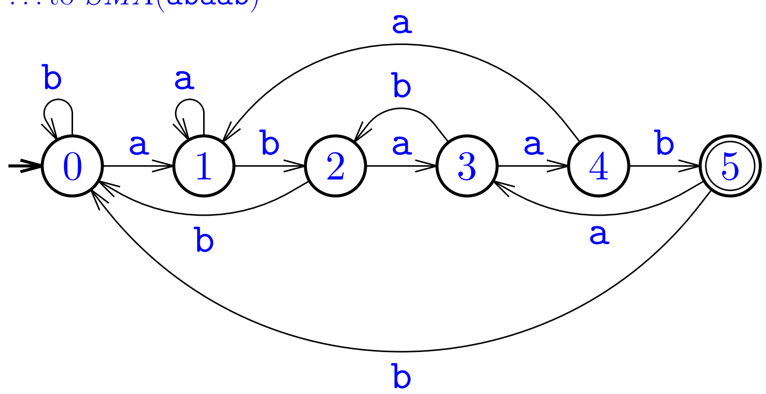
---

Construction of $SMA(x)$

- **Unwinding arcs**
- From $SMA($`abaa`$)$ ...



- ... to $SMA($`abaab`$)$



Construction algorithm

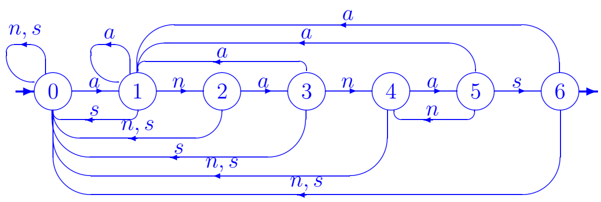
⋆ Unwind the appropriate arc

---

automaton SMA(string x)
    let *initial* be a new state
    $Q \longleftarrow \{initial\}$
    $terminal \longleftarrow initial$
    for all σ **in** Σ **do** $\delta(initial, \sigma) \longleftarrow initial$
    while not end of x **do**
        $\tau \longleftarrow$ next symbol of x
        $r \longleftarrow \delta(terminal, \tau)$
        add new state s to Q
        $\delta(terminal, \tau) \longleftarrow s$
        for all σ in Σ **do** $\delta(s, \sigma) \longleftarrow \delta(r, \sigma)$
        $terminal \longleftarrow s$
    return $(Q, \Sigma, initial, \{terminal\}, \delta)$

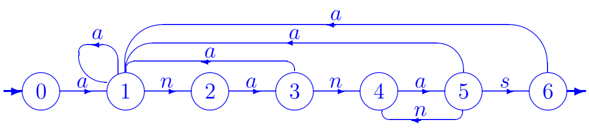
Significant arcs

⋆ Complete $SMA($`ananas`$)$



⋆ **Forward arcs**: spell the pattern

⋆ **Backward arcs**: arcs going backwards without reaching the initial state



Lemma 1 *$SMA(x)$ contains at most $|x|$ backward arcs.*

⋆ Consequence: the implementation of $SMA(x)$ can be done in $O(|x|)$ time and space, independently of the alphabet size

Backward arcs in SMA

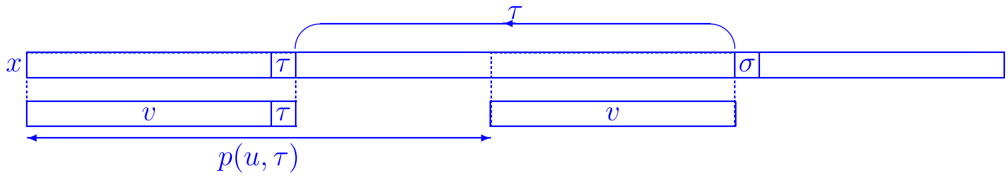
- States of $SMA(x)$ are identified with prefixes of x

  A backward arc is of the form $(u, \tau, v\tau)$ (u, v prefixes of x, τ symbol) with

  – $v\tau$ longest suffix of $u\tau$ that is a prefix of x, and $u \neq v$

  Note: $u\tau$ is not a prefix of x

  Let $p(u, \tau) = |u| - |v|$; it is a period of u because v is a border of u



- **Backward arcs to periods: p is injective**

  Each period p, $1 \leq p \leq |x|$, corresponds to at most one backward arc, thus there are at most $|x|$ such arcs

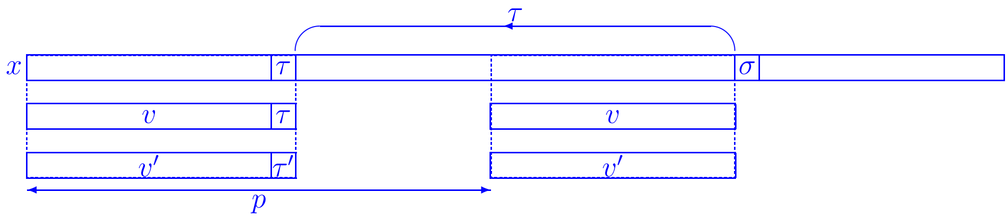
- **A worst case**: $SMA(ab^{m-1})$ has m backward arcs ($a \neq b$)

Backward arcs (followed)

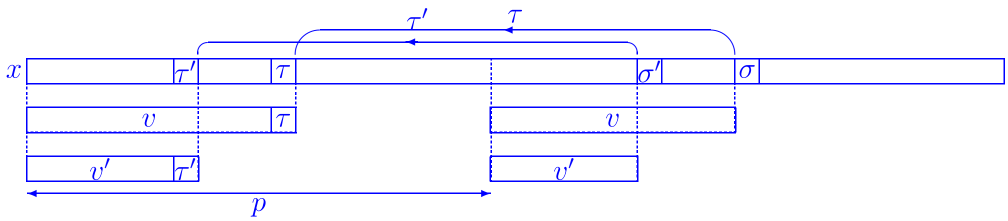
- **Proof that p is injective**

  Two backward arcs $(u, \tau, v\tau)$, $(u', \tau', v'\tau')$

  Assume $p(u, \tau) = p(u', \tau') = p$; we prove $u = u'$ and $\tau = \tau'$.



- **If** $v = v'$ then $u = u'$ and also $\tau = \tau'$



- **If** v' a proper prefix of v then $v'\tau'$ and $v'\sigma'$ are prefixes of v

  thus $\tau' = \sigma'$ a contradiction

Complexity of searching with SMA

- Pattern x of length m, text y of length n
- **With complete SMA implemented by transition matrix**

Preprocessing on pattern x	time	$O(m \times \text{card}\,\Sigma)$
	space	$O(m \times \text{card}\,\Sigma)$
Search on text y	time	$O(n)$
	space	$O(m \times \text{card}\,\Sigma)$
Delay	time	constant

- **With SMA implemented by lists of forward and backward arcs**

Preprocessing on pattern x	time	$O(m)$
	space	$O(m)$
Search on text y	time	$O(n)$
	space	$O(m)$
Delay	comparisons	$\min\{\text{card}\,\Sigma, \log_2 m\}$

- Improves on KMP algorithm

Dictionary Matching

Maxime Crochemore

King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Matching several strings

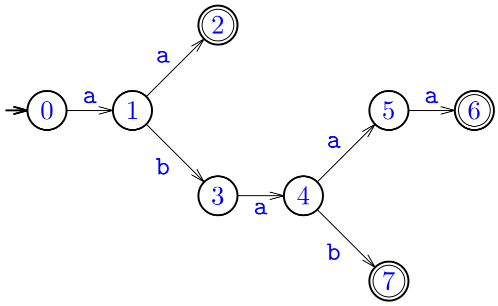
- **Dictionary**: set of finite strings, $X = \{x_0, x_1, \ldots, x_{k-1}\}$ (empty string not in X)
- **Matching**: locate occurrences of the strings in any text y

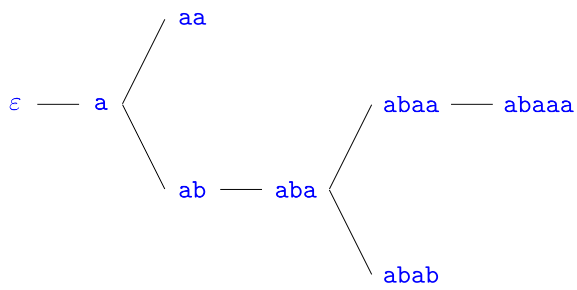
Note that each position on y can be the position of several strings, yielding a possible quadratic output (e.g. $X = \{\mathtt{a}, \mathtt{aa}, \ldots, \mathtt{a}^k\}$ and $y = \mathtt{a}^n$)

- **Output**: list of positions on y that are end-positions of some string in X
- **Standard method**: based on the Dictionary Matching Automaton (Aho-Corasick automaton), an extension of the String Matching Automaton
- **Other method**: extension of the right-to-left window scanning strategy (Boyer-Moore technique)

Trie of the dictionary

- **Trie**: $\mathcal{T}(X)$, digital tree whose branches are labelled by strings of X
  As an automaton, the language it accepts is X
- **Example** : $\mathcal{T}(\{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\})$

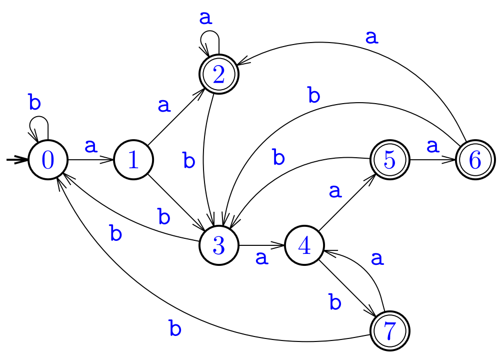




- Nodes are all prefixes of strings in X
- $\mathcal{T}(X)$ is the basis of the Dictionary Matching Automaton

Dictionary Matching Automaton

- **Dictionary Matching Automaton**, $\mathcal{D}(X)$
  it accepts the language A^*X and is defined by:
  - set of states is $\mathrm{Pref}(X)$; initial state is the empty string
  - set of terminal states is $\mathrm{Pref}(X) \cap A^*X$
  - arcs are of the form $(u, a, h(ua))$
    where $h(ua) =$ longest suffix of ua that belongs to $\mathrm{Pref}(X)$
- **Example** : $\mathcal{D}(\{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\})$ with alphabet $A = \{\texttt{a}, \texttt{b}\}$



Searching with DMA

⋆ **Searching**

DET-SEARCH-BY-FAILURE(X, y)
- $M \leftarrow$ DMA-BY-FAILURE(X)
- $r \leftarrow$ *initial*[M]
- **for** each letter a of y, sequentially **do**
  - $r \leftarrow$ TARGET(,)(r, a)
  - OUTPUT-IF(*terminal*[r])

⋆ **Implementation** of $\mathcal{D}(\{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\})$ with a transition table

Next state table	0	1	2	3	4	5	6	7
a	0	2	2	4	5	6	2	4
b	1	3	3	0	7	3	3	0

⋆ **Searching time**: $O(|y|)$

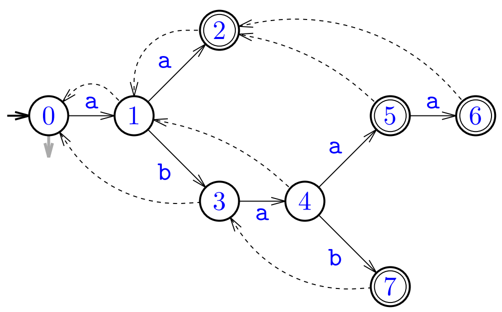
j		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
$y[j]$		c	d	a	b	b	a	b	a	a	b	a	b	a	b	b	a	a	...
state	0	0	0	1	3	0	1	3	4	**5**	3	4	**7**	4	**7**	0	1	**2**	...

Implementation by failure function

⋆ **Aim**: reduction of the implementation to $O(\Sigma\{|x| : x \in X\})$ independent of the alphabet

⋆ **Failure function** (u nonempty string)
$f(u)$ = the longest proper suffix of u that belongs to Pref(X)

⋆ **Implementation** of $\mathcal{D}(\{\texttt{aa}, \texttt{abaaa}, \texttt{abab}\})$ with successor lists and f



state	successors
0	(a,1)
1	(a,2), (b,3)
2	
3	(a,4)
4	(a,5), (b,7)
5	(a,6)
6	
7	

⋆ **Searching**

j		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
$y[j]$		c	d	a	b	b	a	b	a	a	b	a	b	a	b	b	a	a	...
state	0	0	0	1	3	0,0	1	3	4	**5**	2,1,3	4	**7**	3,4	**7**	3,0,0	1	**2**	...

Searching with failure function

⋆ **Searching**

DET-SEARCH-BY-FAILURE(X, y)

$M \leftarrow$ DMA-BY-FAILURE(X)
$r \leftarrow$ *initial*$[M]$
for each letter a of y, sequentially **do**
  $r \leftarrow$ TARGET-BY-FAILURE(r, a)
  OUTPUT-IF(*terminal*$[r]$)

⋆ **Target**: next state function using successors lists

TARGET-BY-FAILURE(p, a)

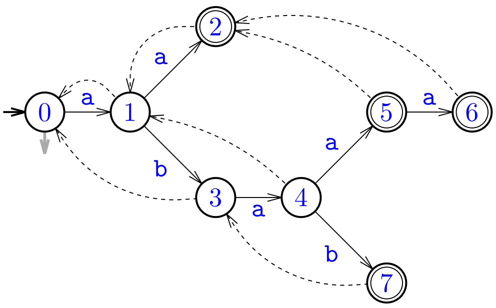
while $p \neq$ NIL and TARGET$(p, a) =$ NIL **do**
  $p \leftarrow$ *fail*$[p]$
if $p =$ NIL **then**
  return *initial*$[M]$
else return TARGET(p, a)

⋆ **Running time**: $O(|y| \times \log \operatorname{card} A)$

Computing the failure links

⋆ $f(ua) =$ TARGET-BY-FAILURE$(f(u), a)$

⋆ **Computation via a width-first traversal of the trie**



⋆ *fail*[0], *fail*[1], *fail*[2], *fail*[3], and *fail*[4] already computed

⋆ **Computing *fail*[5] from** 4: $\delta(4, \mathtt{a}) = 5$, *fail*$[4] = 1$, and $\delta(1, \mathtt{a}) = 2$ gives *fail*$[5] = 2$

Note: since state 2 is terminal, state 5 becomes terminal

⋆ **Computing *fail*[6] from** 5: $\delta(5, \mathtt{a}) = 6$, *fail*$[5] = 2$, and $\delta(2, \mathtt{a})$ undefined but TARGET-BY-FAILURE$(2, \mathtt{a}) = 2$ gives *fail*$[6] = 2$

Computing the failure links—algorithm

⋆ **Failure links** *fail*[] implements the failure function f

DMA-BY-FAILURE(X)

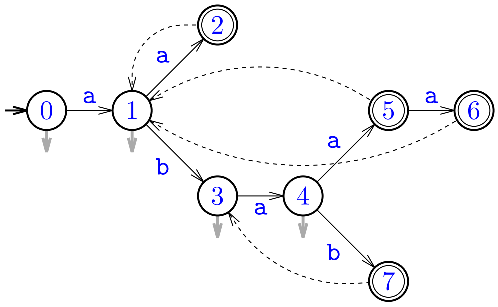
```
M ← Trie(X)
fail[initial[M]] ← NIL
F ← Empty-Queue()
Enqueue(F, initial[M])
while non Queue-is-empty(F) do
    t ← Dequeued(F)
    for each pair (a, p) ∈ Succ[t] do
        r ← Target-by-failure(fail[t], a)
        fail[p] ← r
        if terminal[r] then
            terminal[p] ← TRUE
        Enqueue(F, p)
return M
```

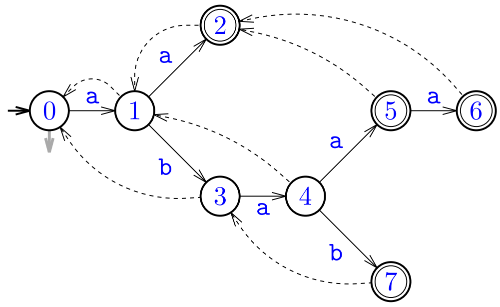

Optimized failure links

⋆ **Next set** of state $u \in \mathrm{Pref}(X)$:
$Next(u) = \{a : a \in A, ua \in \mathrm{Pref}(X)\}$

⋆ **New link** defined by function f': for u nonempty $f'(u) = f^k(u)$
where $k = \min\{\ell : Next(f^\ell(u)) \not\subseteq Next(u)\}$

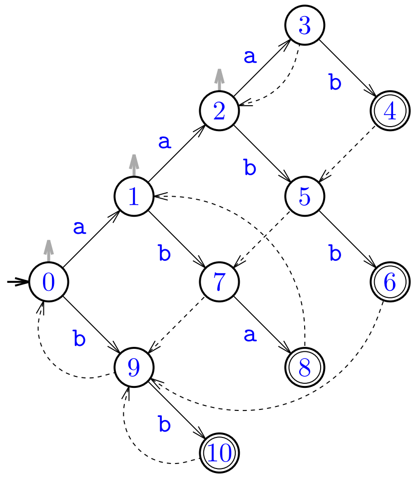
⋆ **Optimized link** **Non optimized**





Delay

- **Delay**: maximum time spent on a letter of y
  It is $\max\{|x| : x \in X\}$ even with the optimized links
- **Example**: $X = \{\texttt{aaab}, \texttt{aabb}, \texttt{aba}, \texttt{bb}\}$



- $L(m) = \{\texttt{a}^{m-1}\texttt{b}\} \cup \{\texttt{a}^{2k-1}\texttt{ba} : 1 \leq k < \lceil m/2 \rceil\} \cup \{\texttt{a}^{2k}\texttt{bb} : 0 \leq k < \lfloor m/2 \rfloor\}$

Practically-efficient String Matching

Maxime Crochemore

King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Examples

- Naive search with backward scan (1)

```
a a c a a a a a b a b a a b a b . .
a b a a a a
  a b a a a a
    a b a a a a
      a b a a a a
        a b a a a a
          . . .
```

- Naive search with backward scan (2)

```
a b a b a a a b a b b a a b a b . .
a b a a a b a b
  a b a a a b a b
    a b a a a b a b
      a b a a a b a b
        . . .
```

Right-to-left scan — shift

`text`	···	c	g	c	t	c	**g**	**c**	g	c	t	**a**	t	c	g	···
pattern		c	g	c	t	a	**g**	**c**								
						c	**g**	**c**	t	a	g	c				
								c	g	c	t	**a**	g	c		

- Match shift: good-suffix rule (function d)
- Occurrence shift heuristics: bad-character rule
- Extra rules if memorization

---

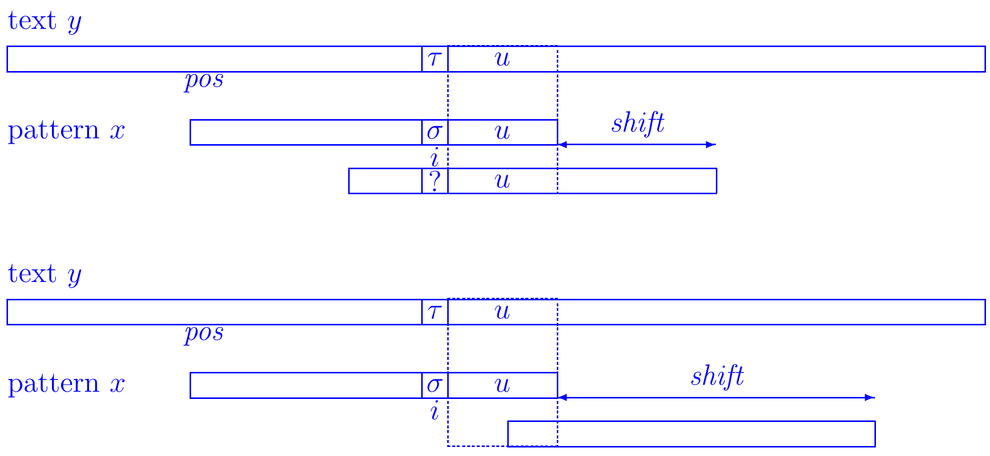
while window on text **do**

  $u \longleftarrow$ longest common suffix of window and pattern

  if $u =$ pattern **then** report a match

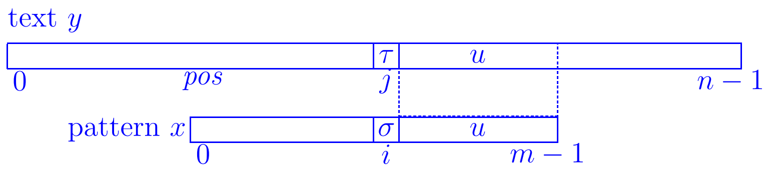
  shift window $d(u)$ places to the right

Match shift



- **Precomputation**
  of rightmost occurrences of u's: $O(m)$
- Second shift length = a period of x
- Table D implements the good-suffix rule:
  $shift = d(u) = D[i]$

BM algorithm

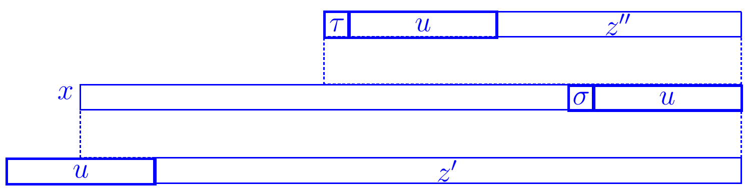


- ⋆ No memorization of previous matches

BM(string x, y; integer m, n)
    $pos \longleftarrow 0$
    while $pos \leq n - m$ **do**
        $i \longleftarrow m - 1$
        while $i \geq 0$ **and** $x[i] = y[pos + i]$ **do** $i \longleftarrow i - 1$
        if $i = -1$ **then**
            output('x occurs in y at position ', pos)
            $pos \longleftarrow period(x)$
        else
            $pos \longleftarrow pos + D[i]$

Suffix displacement

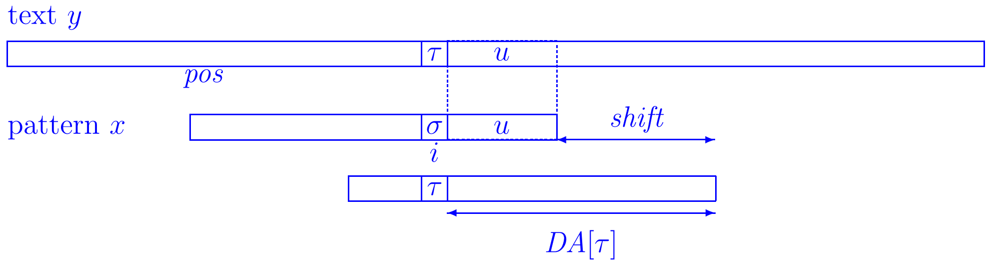
- ⋆ Displacement function d:
  $d(u) = \min\{|z| > 0 \mid (x \text{ suffix of } uz)$ or
  $(\tau uz \text{ suffix of } x \text{ and } \tau u \text{ not suffix of } x, \text{ for } \tau \in \Sigma)\}$
- ⋆ Displacement table D:
  $D[i] = d(x[i+1 \,..\, m-1]), \text{ for } i = 0, .., m-1$



- ⋆ Note 1: u is a (strict) border of uz''
- ⋆ Note 2: $|z'|$ is a period of uz' (thus, $|z'|$ is a period of x)

Lemma 1 *Table D can be computed in linear time.* [see Page 22]

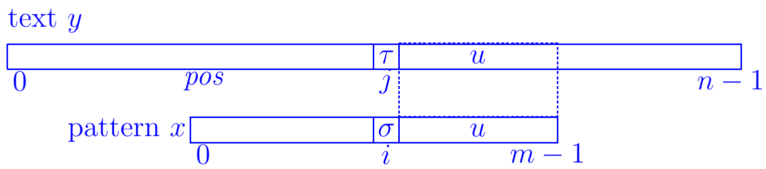
Occurrence shift



- ⋆ Table DA implements the bad-character rule:
  $DA[\sigma] = \min\{|z| > 0 \mid \sigma z \text{ suffix of } x\} \cup \{|x|\}$
- ⋆ $shift = DA[\tau] - |u| = DA[\tau] - m + i + 1$

COMPUTE_DA(string x; integer m)
    for all σ **in** Σ **do**
        $DA[\sigma] = m$
    for $i \longleftarrow 0$ **to** $m-2$ **do**
        $DA[x[i]] = m - i - 1$
    return DA

BM with occurrence shift



- ⋆ Use of DA in addition to D

BM(string x, y; integer m, n);
    $pos \longleftarrow 0$
    while $pos \leq n - m$ **do**
        $i \longleftarrow m - 1$
        while $i \geq 0$ **and** $x[i] = y[pos + i]$ **do**
            $i \longleftarrow i - 1$
        if $i = -1$ **then**
            output('x occurs in y at position ', pos)
            $pos \longleftarrow period(x)$
        else
            $pos \longleftarrow pos + \max\{D[i], DA[y[pos + i]] - m + i + 1\}$

Complexity of BM

- **Preprocessing phase**

match shift	$O(m)$
occurrence shift	$O(m + \mathrm{card}\,\Sigma)$

- **Search phase** (finding all occurrences)

running time	$O(n \times m)$
minimum number of comparisons	n/m
maximum number of comparisons	$n \times m$

- **Extra space**

for shift functions	$O(m + \mathrm{card}\,\Sigma)$
can be reduced to	$O(m)$

Symbol comparisons in variants of BM

- For finding the first occurrence

[Knuth, Morris, Pratt, 1977]	$\leq 7 \times n$
[Guibas, Odlysko, 1980]	$\leq 4 \times n$
[Cole, 1990]	$\leq 3 \times n$

Theorem 1 *If period$(x) > m/2$, BM searching algorithm performs at most $3n - n/m$ symbol comparisons. The bound is tight.*

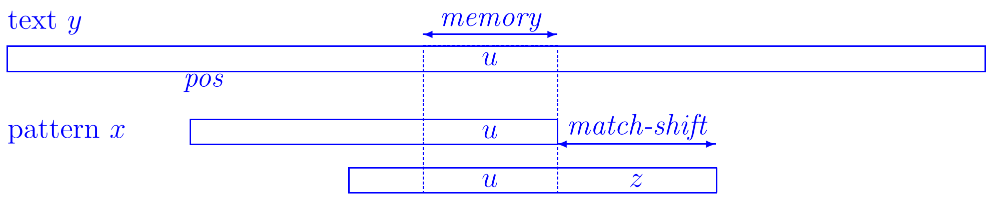
Proof difficult

Symbol comparisons in variants of BM (followed)

- For finding all occurrences

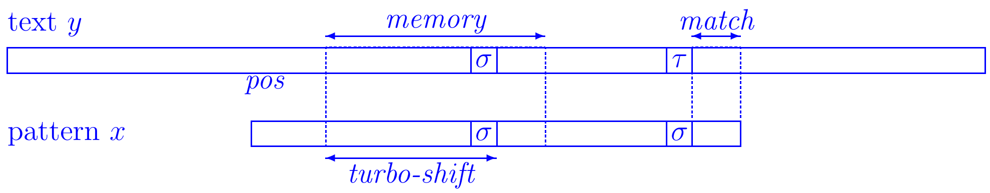
[Galil, 1979]	$O(n)$
$\longrightarrow$ prefix memorization	$O(1)$ extra space
[Crochemore *et alii*, 1991]	$\leq 2 \times n$
$\longrightarrow$ last-suffix memorization	$O(1)$ extra space
$\longrightarrow$ **Turbo-BM**	
[Apostolico, Giancarlo, 1986]	$\leq 1.5 \times n$
$\longrightarrow$ all-suffix memorization	$O(m)$ extra space

Turbo-BM method



- **Features**
  - **Stores** the last match in case of match shift (*memory*)
  - **Jumps** on *memory*
  - Uses **turbo-shifts**
- **Preprocessing**
  same as BM algorithm
- **Search**
  $O(1)$ extra space to store *memory*: (length, right position)
- Note: *match-shift* is a period of uz since u is a border of it

Turbo-shift



- ⋆ *turbo-shift* $= |memory| - |match|$
- ⋆ **Use in Turbo-BM**:

  shift $\leftarrow \max\{$*match-shift, occ-shift, turbo-shift*$\}$;

  if (*shift* = *match-shift*) **then**

  set memory;

  else

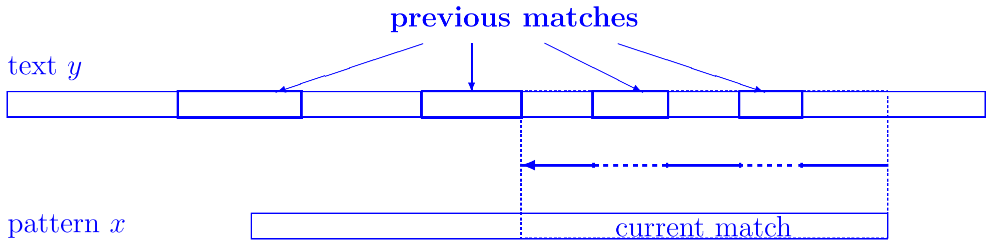
  $\{$ *shift* $\leftarrow \max\{$*shift, match*$+1\}$; no memory; $\}$

Tight number of comparisons for Turbo-BM

Theorem 2 *The Turbo-BM searching algorithm runs in time $O(n)$. It makes no more than $2n$ symbol comparisons.*

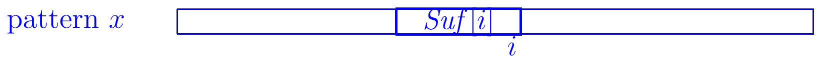
Proof difficult

AG method



- ⋆ **Features**
  - **Stores** all previous matches (suffixes of pattern)
  - Uses the table *Suf*

    $Suf[i]$ = longest suffix of x ending at i in x



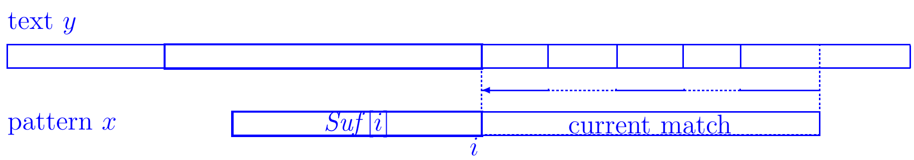
- ⋆ **Extra preprocessing**

  computing *Suf* on the pattern: $O(m)$ time and space
- ⋆ **Search**

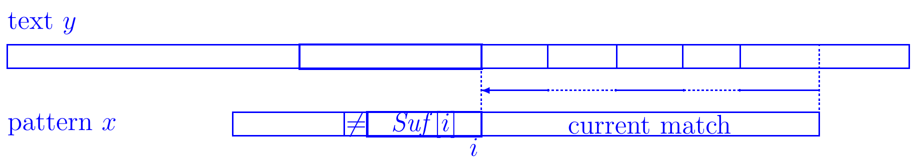
  $O(m)$ extra space to store the matches

Rules for shifts in AG

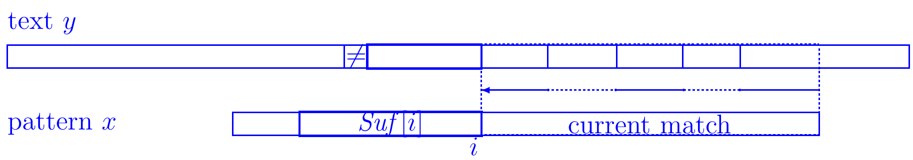
- **match**



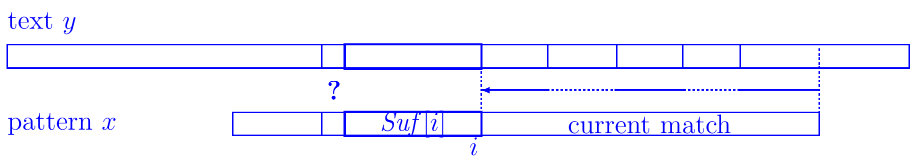
- **mismatch**



- **mismatch**



- **jump**



Tight number of comparisons for AG

Lemma 2 *The AG algorithm makes at most $\frac{n}{2}$ comparisons on text characters previously compared.*

Theorem 3 *The AG searching algorithm runs in time $O(n)$. It makes no more than $1.5n$ symbol comparisons.*

Proof rather difficult

Worst-case example

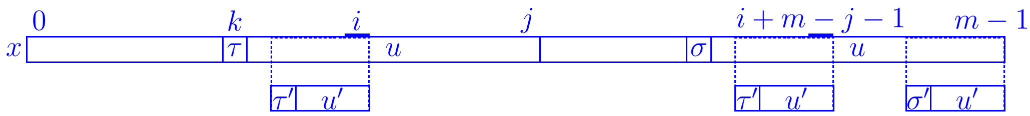
```
a a b a a a b a a b a a a b a a b a a a b ...
a a b a a a b
        a a b a a a b
          a a b a a a b
            a a b a a a b
              a a b a a a b
                      a a b a a a b
                        a a b a a a b
                          a a b a a a b
                            a a b a a a b
```

pattern $= a^{m-1}ba^mb$, text $= (a^{m-1}ba^mb)^e$

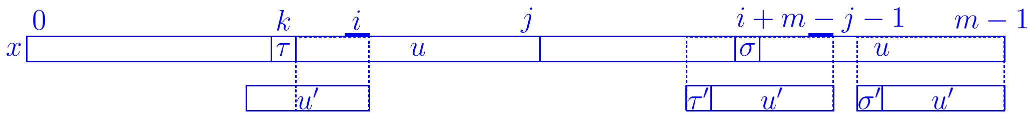
number of comparisons $= 2m+1+(3m+1)e = \frac{3m+1}{2m+1}n - m \approx 1.5n$

Computing the table of suffixes

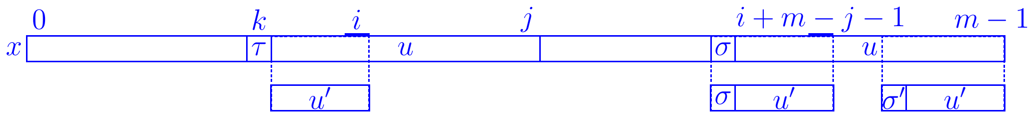
★ i given: $Suf[j]$ known for $i < j \leq m-1$
$k = \min\{j - Suf[j] \mid i < j < m\}$ (leftmost considered position)
$\sigma \neq \tau$; $\sigma' \neq \tau'$; $Suf[i+m-j-1] = |u'|$



★ **If** $Suf[i+m-j-1] < i-k$: $Suf[i] = Suf[i+m-j-1]$



★ **If** $Suf[i+m-j-1] > i-k$: $Suf[i] = i-k$



★ **If** $Suf[i+m-j-1] = i-k$: find $Suf[i]$ by scanning **from position** k
Yields a new k and a new j $(= i)$

Extra preprocessing for AG

★ Linear-time computation of table Suf

COMPUTE_SUF(string x; integer m)
  $Suf[m-1] \longleftarrow m$; $k \longleftarrow m-1$
  for $i \longleftarrow m-2$ **downto** 0 **do**
    if $i > k$ **and** $Suf[i+m-j-1] \neq i-k$ **then**
      $Suf[i] \longleftarrow \min\{Suf[i+m-j-1], i-k\}$
    else
      $j \longleftarrow i$; $k \longleftarrow \min\{i,k\}$;
      while $k \geq 0$ **and** $x[k] = x[k+m-j-1]$ **do**
        $k \longleftarrow k-1$;
      $Suf[i] \longleftarrow j-k$
  return Suf

From *Suf* to D

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	a	a	b	a	b	a	b	a
$Suf[i]$	1	0	3	1	1	0	3	0	5	0	11

★ **Initializing D using the periods of x only**

$Suf[2] = 3 \Longrightarrow$ period 8 ; $Suf[0] = 1 \Longrightarrow$ period 10 ; period 11

8	8	8	8	8	8	8	8	10	10	11

★ **Accounting for occurrences inside x of its suffixes**

$Suf[3] = 1 \Longrightarrow D[9] \leq 7$; $Suf[8] = 5 \Longrightarrow D[5] \leq 2$

	8	8	8	8	8	$\cancel{8}$	8	$\cancel{8}$	10	$\cancel{10}$	$\cancel{11}$
						2		$\cancel{8}$		$\cancel{10}$	$\cancel{9}$
								4		$\cancel{7}$	$\cancel{5}$
										6	$\cancel{3}$
											1
$D[i]$	8	8	8	8	8	2	8	4	10	6	1

Computing the displacement table D with the table *Suf*

★ Linear-time computation of table D

COMPUTE_D(string x; integer m; table D)
    $j \longleftarrow 0$;
    for $i \longleftarrow m-2$ **downto** -1 **do**
        if $i = -1$ **or** $Suf[i] = i+1$ **then**
            while $j < m-i-1$ **do**
                $D[j] \longleftarrow m-i-1$;
                $j \longleftarrow j+1$;
    for $i \longleftarrow 0$ **to** $m-2$ **do**
        $D[m - Suf[i] - 1] \longleftarrow m-i-1$;
    return D

Searching a list of strings

Maxime Crochemore

King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Searching problem

⋆ **Input**

– a list L of n strings of Σ^* stored in increasing lexicographic order in a table: $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$

– a string $x \in \Sigma^*$ of length m.

⋆ **Simple problem**

find

– either i, $-1 < i < n$, with $x = L_i$ if x occurs in L,

– or d and f, $-1 \leq d < f \leq n$, that satisfy $d + 1 = f$ and $L_d < x < L_f$ otherwise.

⋆ **Interval**

find d et f, $-1 \leq d < f \leq n$, with:
$d < i < f$ if and only if x prefix of L_i.

Example

- **List** L

  $L_0 = \texttt{a a a b a a}$

  $L_1 = \texttt{a a a b b}$

  $L_2 = \texttt{a a b b b b}$

  $L_3 = \texttt{a b}$

  $L_4 = \texttt{b a a a}$

  $L_5 = \texttt{b b}$

- **Search**

  $x = \texttt{a a a b b} \longrightarrow 1$

  $x = \texttt{a a b a} \longrightarrow (1, 2)$

- **Interval**

  $x = \texttt{a a} \longrightarrow (-1, 3)$

Searching algorithm

SIMPLE-SEARCH(L, n, x, m)

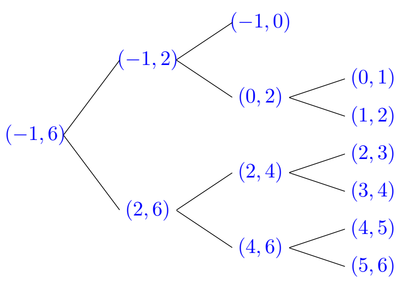
$d \leftarrow -1$
$f \leftarrow n$
while $d + 1 < f$ **do** $\quad\triangleright$ Invariant: $L_d < x < L_f$
  $i \leftarrow \lfloor (d + f)/2 \rfloor$
  $\ell \leftarrow |lcp(x, L_i)|$
  if $\ell = m$ and $\ell = |L_i|$ **then**
    return i
  elseif $(\ell = |L_i|)$ or $(\ell \neq m$ and $L_i[\ell] < x[\ell])$ **then**
    $d \leftarrow i$
  else $f \leftarrow i$
return (d, f)

Complexity

- **Running time**
  $O(m \times \log n)$
- **Worst case**
  - list $L = (\mathtt{a}^{m-1}\mathtt{b}, \mathtt{a}^{m-1}\mathtt{c}, \mathtt{a}^{m-1}\mathtt{d}, \ldots)$
  - string $x = \mathtt{a}^m$
- **Additional space**
  constant

Binary search tree

- **Nodes**
  $n+1$ external nodes $(-1,0), (0,1), (1,2), \ldots, (n-1,n)$
  n internal nodes in the form (d,f) with $d+1 < f$
  children of (d,f): $(d, \lfloor (d+f)/2 \rfloor)$ and $(\lfloor (d+f)/2 \rfloor, f)$
  root: $(-1,n)$
- **Size**
  $2n+1$ nodes for a list of n strings
- **Example for** $n = 6$



Search using LCP's

- **Aim**
  reduce the running time to $O(m + \log n)$
- **LCP**, *longest common prefix*
  $lcp(L_d, L_f)$ known for any pair (d, f) considered in the binary search
- **Additional space**: $O(n)$ integers for the $2n+1$ LCP's associated with nodes of the binary search tree
- **Algorithm** based on properties arising in three cases (plus symmetric cases)
- **Variables**
  $ld = |lcp(x, L_d)|$, $lf = |lcp(x, L_f)|$, $i = \lfloor (d+f)/2 \rfloor$
  maintained during execution

Case one

- **Hypotheses**
  $L_d < x < L_f$ and $ld \leq |lcp(L_i, L_f)| < lf$
- **Example**

L_d	a a a c a	x	a a b b b a a
	a a a c b a		
L_i	a a b b a b a		
	a a b b a b b		
L_f	a a b b b a b	x	a a b b b a a

- **Conclusion**
  $L_i < x < L_f$ and $|lcp(x, L_i)| = |lcp(L_i, L_f)|$

Case two

- **Hypotheses**

  $L_d < x < L_f$ and $ld \leq lf < |lcp(L_i, L_f)|$

- **Example**

L_d	a a a c a	x	a a b a c b
	a a a c b a		
L_i	a a b b a b a		
	a a b b a b b		
L_f	a a b b b a b	x	a a b a c b

- **Conclusion**

  $L_d < x < L_i$ and $|lcp(x, L_i)| = |lcp(x, L_f)|$

Case three

- **Hypotheses**

  $L_d < x < L_f$ and $ld \leq lf = |lcp(L_i, L_f)|$

- **Example**

L_d	a a a c a	x	a a b b a b
	a a a c b a		
L_i	a a b b a b a		
	a a b b a b b		
L_f	a a b b b a b	x	a a b b a b

- **Conclusion**

  compare x and L_i from position lf

Improved searching algorithm

Search(L, n, Lcp, x, m)

$(d, ld) \leftarrow (-1, 0)$
$(f, lf) \leftarrow (n, 0)$
while $d + 1 < f$ **do** ▷ Invariant : $L_d < x < L_f$
    $i \leftarrow \lfloor (d + f)/2 \rfloor$
    if $ld \leq Lcp(i, f) < lf$ **then**
        $(d, ld) \leftarrow (i, Lcp(i, f))$
    elseif $ld \leq lf < Lcp(i, f)$ **then**
        $f \leftarrow i$
    elseif $lf \leq Lcp(d, i) < ld$ **then**
        $(f, lf) \leftarrow (i, Lcp(d, i))$
    elseif $lf < ld < Lcp(d, i)$ **then**
        $d \leftarrow i$
    else $\ell \leftarrow \max\{ld, lf\}$
        $\ell \leftarrow \ell + |lcp(x[\ell \,.\,.\, m-1], L_i[\ell \,.\,.\, |L_i| - 1])|$
        if $\ell = m$ and $\ell = |L_i|$ **then**
            return i
        elseif $(\ell = |L_i|)$ or $(\ell \neq m$ and $L_i[\ell] < x[\ell])$ **then**
            $(d, ld) \leftarrow (i, \ell)$
        else $(f, lf) \leftarrow (i, \ell)$
return (d, f)

Complexity

Proposition 1 *Algorithm* Search *finds a string x of length m in a sorted list of n strings in time $O(m + \log n)$.*
It makes no more than $m + \lceil \log(n + 1) \rceil$ comparisons of letters.
It requires $O(n)$ extra space.

Sketch of the proof

Number of letter comparisons:

- ⋆ each positive comparison strictly increases ℓ, yielding no more than m such comparisons.
- ⋆ each negative comparison leads to divide by two the value of $f - d$, producing no more than $\lceil \log(n + 1) \rceil$ such comparisons.

Lcp can be implemented to run in constant time after preprocessing.

Interval

```
▷ next line replace line 15 of Search
if ℓ = m then
      ▷ next lines replace line 16 of Search
      e ← i
      while d + 1 < e do
            j ← ⌊(d + e)/2⌋
            if Lcp(j, e) < m then
                  d ← j
            else e ← j
      if Lcp(d, e) ≥ m then
            d ← max{d − 1, −1}
      e ← i
      while e + 1 < f do
            j ← ⌊(e + f)/2⌋
            if Lcp(e, j) < m then
                  f ← j
            else e ← j
      if Lcp(e, f) ≥ m then
            f ← min{f + 1, n}
      return (d, f)
```


Preprocessing the list

Let $||L|| = \Sigma_{i=0}^{n-1} |L_i|$.

- **Sorting**
  repetitive application of bin sorting: time $O(||L||)$
- **Computing LCP's** of L_{f-1} and L_f, $0 \leq f \leq n$
  straight algorithm: time $O(||L||)$
- **Computing other LCP's**
  based on next lemma

Lemma 1 *Let $L_0 \leq L_1 \leq \ldots \leq L_{n-1}$. Let d, i and f, $-1 < d < i < f < n$. Then $|lcp(L_d, L_f)| = \min\{|lcp(L_d, L_i)|, |lcp(L_i, L_f)|\}$.*

Proposition 2 *Preprocessing L, sorting and computing LCP's, takes $O(||L||)$ time.*

Suffix Arrays

Maxime Crochemore

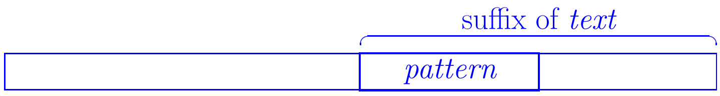
King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Indexes

- ⋆ **Pattern matching** in static texts
- ⋆ **Basic operations**
  - existence of patterns in the text
  - number of occurrences of patterns
  - list of positions of occurrences
- ⋆ **Other applications**
  - finding repetitions in texts
  - finding regularities in texts
  - approximate matchings
  - ...

Implementation of indexes



Implementation with efficient data structures

- ⋆ **Suffix Trees**
  digital trees, PATRICIA tree (compact trees)
- ⋆ **Suffix Automata or DAWG's**
  minimal automata, compact automata

Implementation with efficient algorithm

- ⋆ **Suffix Arrays**
  binary search in the ordered list of suffixes

Efficient constructions

- ⋆ **Position tree, suffix tree**
  [Weiner 1973], [McCreight, 1976], [Ukkonen, 1992]
  [Farach, 1997]
- ⋆ **Suffix DAWG, suffix automaton, factor automaton**
  [Blumer *et al.*, **1983], [Crochemore, 1984]**
- ⋆ **Suffix array, PAT array**
  [Manber, Myers, 1990], [Gonnet, 1986]
  [Kärkkäinen, Sanders, 2003]
  [Kim et al., 2003], [Ko, Aluru, 2003]
- ⋆ **Some other implementations of suffix trees**
  [Andersson, Nilsson, 1993], [Irving, 1995]
  [Kärkkäinen, 1995], [Munro et al., 1999]
- ⋆ **For external memory** (*SB-trees*)
  [Ferragina, Grossi, 1995]
- ⋆ **Compact suffix automaton**
  [Crochemore, Vérin, 1997], [Inenaga et al., 2001]

Suffixes

Text $y \in \Sigma^*$

- ⋆ $\mathit{Suff}(y)$ = set of suffixes of y,
- ⋆ card $\mathit{Suff}(y) = |y| + 1$
- ⋆ Suff(`ababbb`)

i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

						position	
a	b	a	b	b	b	0	
	b	a	b	b	b	1	
		a	b	b	b	2	
			b	b	b	3	
				b	b	4	
					b	5	
					ε	6	(empty string)

Suffix array

- ⋆ **Text** $y \in \Sigma^*$ of length n
- ⋆ **Permutation of suffixes positions**
  SUF: $\{0, 1, \ldots n-1\} \rightarrow \{0, 1, \ldots n-1\}$ such that
  $$y[SUF[0] \,..\, n-1] < y[SUF[1] \,..\, n-1] < \ldots < y[SUF[n-1] \,..\, n-1]$$
- ⋆ **LCP's of suffixes**
  $LCP[i] = |lcp(y[SUF[i-1] \,..\, n-1], y[SUF[i] \,..\, n-1])|$

i	$SUF[i]$	$LCP[i]$	
0	10	0	a
1	0	1	a a b a a b a a b b a
2	3	6	a a b a a b b a
3	6	3	a a b b a
4	1	1	a b a a b a a b b a
5	4	5	a b a a b b a
6	7	2	a b b a
7	9	0	b a
8	2	2	b a a b a a b b a
9	5	4	b a a b b a
10	8	1	b b a

Operations on indexes

Text y of length n

- **Index implemented by suffix array of y**
  memory space $O(n)$
- **String matching**
  searching y for x of length m: time $O(m + \log n)$
  number of occurrences of x in y: same complexity
- **All occurrences**
  finding all occurrences of x in y: time $O(m + \log n + |output|)$
- **Repetitions**
  computing a longest factor of y occurring at least k times: time $O(n)$
- **Marker**
  computing a shortest factor of y occurring exactly once: time $O(n)$

Computing a suffix array

- **Sorting suffixes**
  previous solution runs in time $O(n^2)$ because $||Suff(y)|| = O(n^2)$
  $O(n \log n)$-time algorithm based on the **doubling technique**
  $O(n)$-time algorithm on integer alphabet
  [Manber and Myers, 1993], **[Kärkkäinen, Sanders, 2003]**
- **Computing LCP's of suffixes**
  previous solution runs in time $O(n^2)$ because card $Suff(y) = O(n^2)$
  $O(n)$-time simple algorithm using SUF and its inverse r
  [Kasai et al., 2001]
- see also **[Kim et al., 2003]**, **[Ko, Aluru, 2003]**

Ranks of suffixes

Text y of length n, string $u \in \Sigma^*$, integer $k > 0$

⋆ **First**

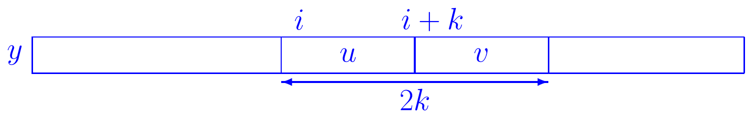
$$First_k(u) = \begin{cases} u & \text{if } |u| \le k \\ u[0\,.\,.\,k-1] & \text{otherwise} \end{cases}$$

⋆ **Rank**

i position of suffix $y[i\,.\,.\,n-1]$

$R_k[i]$ = rank of $First_k(y[i\,.\,.\,n-1])$ inside the ordered list of all $First_k(u)$, u non-empty suffix of y

Lemma 1 (doubling) *$R_{2k}[i]$ is the rank of $(R_k[i], R_k[i+k])$ in the ordered list of these pairs.*



Doubling technique

Input

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a

Output

$k=1$	$\{0,1,3,4,6,7,10\}$	$\{2,5,8,9\}$									
$k=2$	$\{10\}$	$\{0,3,6\}$	$\{1,4,7\}$	$\{2,5,9\}$	$\{8\}$						
$k=4$	$\{10\}$	$\{0,3\}$	$\{6\}$	$\{1,4\}$	$\{7\}$	$\{9\}$	$\{2,5\}$	$\{8\}$			
$k=8$	$\{10\}$	$\{0\}$	$\{3\}$	$\{6\}$	$\{1\}$	$\{4\}$	$\{7\}$	$\{9\}$	$\{2\}$	$\{5\}$	$\{8\}$

One step, doubling to get R_4 from R_2

i	0	1	2	3	4	5	6	7	8	9	10
$R_2[i]$	1	2	3	1	2	3	1	2	4	3	0
pair	(1,3)	(2,1)	(3,2)	(1,3)	(2,1)	(3,2)	(1,4)	(2,3)	(4,0)	(3,-1)	(0,-1)
$R_4[i]$	1	3	6	1	3	6	2	4	7	5	0

Sorting suffixes

```
Suffix-sort(y, n)
for r ← 0 to n − 1 do
    SUF[r] ← r
k ← 1
for i ← 0 to n − 1 do
    Rk[i] ← rank of y[i] in the ordered list of letters in alph(y)
SUF ← Sort(SUF, n, Rk, 0)
i ← card alph(y)
while i < n do
    SUF ← Sort(SUF, n, Rk, k)
    SUF ← Sort(SUF, n, Rk, 0)
    i ← 0
    R2k[SUF[0]] ← i
    for r ← 1 to n − 1 do
        if Rk[SUF[r]] ≠ Rk[SUF[r − 1]] or Rk[SUF[r] + k] ≠ Rk[SUF[r − 1] + k] then
            i ← i + 1
        R2k[SUF[r]] ← i
    (k, Rk) ← (2k, R2k)
return SUF
```


Complexity of the sorting

- **Sort**
  implemented by bucket sort (radix sort)
  each call runs in $O(n)$ time
- **One step**
  runs in $O(n)$ time

Proposition 1 *Algorithm* Suffix-sort *applied to text y of length n runs in $O(n \log n)$ time and $O(n)$ space.*

Sorting Suffixes on a Bounded Integer Alphabet

- ⋆ **Skew algorithm**

  [1] bucket sort positions i according to $First_3(y[i \mathinner{.\,.} n-1])$, for $i = 3q$ or $i = 3q+1$ (include $i = n$ if n multiple of 3) $t[i]$: rank of i in the sorted list

  [2] recursively sort the suffixes of the 2/3-shorter word
  $t[0]t[3]\cdots t[3q]\cdots t[1]t[4]\cdots t[3q+1]\cdots$
  $s[i]$: rank of suffix i in the sorted list ($i = 3q$ or $i = 3q+1$ position on y)

  [3] sort suffixes $y[j \mathinner{.\,.} n-1]$ for j of the form $3q+2$ (i.e., bucket sort pairs $(y[j], s[j+1])$)

  [4] merge lists obtained at steps 2 and 3
  Note: comparing suffixes i (first list) and j (second list) remains to compare:
  $(y[i], s[i+1])$ and $(y[j], s[j+1])$ if $i = 3q$
  $(y[i]y[i+1], s[i+2])$ and $(y[j]y[j+1], s[j+2])$ if $i = 3q+1$

- ⋆ **Running time**: $T(n) = T(2n/3) + O(n)$ then $T(n) = O(n)$
  [Kärkkäinen, Sanders, 2003]

Example 1

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a

Rank t	
0	a
1	a a b
2	a b a
3	a b b
4	b a

Rank s	i	$Suff$(11142230)
0	10	0
1	0	1 1 1 4 2 2 3 0
2	3	1 1 4 2 2 3 0
3	6	1 4 2 2 3 0
4	1	2 2 3 0
5	4	2 3 0
6	7	3 0
7	9	4 2 2 3 0

Rank	j	$(y[j], s[j+1])$
0	2	(b, 2)
1	5	(b, 3)
2	8	(b, 7)

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a
$r[i]$	1	4	8	2	5	9	3	6	10	7	0
$SUF[i]$	10	0	3	6	1	4	7	9	2	5	8

Example 2 — WRONG

i	0	1	2	3	4	5	6	7	8
$y[i]$	a	b	a	a	a	a	a	a	a

Rank t	
0	a a
1	a a a
2	a b a
3	b a a

Rank s	i	*Suff*(211310)
0	7	0
1	4	1 0
2	3	1 1 3 1 0
3	6	1 3 1 0
4	0	2 1 1 3 1 0
5	1	3 1 0

- ⋆ **WRONG: suffix at position 6 does not have the right rank**
- ⋆ **Solution**: when n is a multiple of 3, consider position n during steps 1 to 3 as if $y[n]$ is a symbol smaller than any other symbol

Example 2 — Correct

i	0	1	2	3	4	5	6	7	8	**9**
$y[i]$	a	b	a	a	a	a	a	a	a	

Rank t	
0	ε
1	a a
2	a a a
3	a b a
4	b a a

Rank s	i	*Suff*(3220421)
0	9	0 4 2 1
1	7	1
2	6	2 0 4 2 1
3	4	2 1
4	3	2 2 0 4 2 1
5	0	3 2 2 0 4 2 1
6	1	4 2 1

Rank	j	$(y[j], s[j+1])$
0	8	(a, 0)
1	5	(a, 2)
2	2	(a, 4)

i	0	1	2	3	4	5	6	7	8
$y[i]$	a	b	a	a	a	a	a	a	a
$r[i]$	7	8	6	5	4	3	2	1	0
$SUF[i]$	8	7	6	5	4	3	2	0	1

Computing LCP's of suffixes

⋆ **LCP's of suffixes**

$LCP[i] = |lcp(y[SUF[i-1]\,..\,n-1], y[SUF[i]\,..\,n-1])|$, for $0 \le i \le n$

i	0	1	2	3	4	5	6	7	8	9	10	11
$y[i]$	a	a	b	a	a	b	a	a	b	b	a	
$SUF[i]$	10	0	3	6	1	4	7	9	2	5	8	
$LCP[i]$	0	1	6	3	1	5	2	0	2	4	1	0

j	$r[j]$		j	$r[j]$	
0	1	a a b a a b a a b b a	1	4	a b a a b a a b b a
3	2	a a b a a b b a	4	5	a b a a b b a

Lemma 2 *Let $j \in (1, 2, \ldots, n-1)$ with $r[j] > 0$.*
Then $LCP[r[j-1]] - 1 \le LCP[r[j]]$.

Example

i	$SUF[i]$	$LCP[i]$	
0	10	0	a
1	0	1	a a b a a b a a b b a
2	3	6	a a b a a b b a
3	6	3	a a b b a
4	1	1	a b a a b a a b b a
5	4	5	a b a a b b a
6	7	2	a b b a
7	9	0	b a
8	2	2	b a a b a a b b a
9	5	4	b a a b b a
10	8	1	b b a

i / $r[j]$	$SUF[i]$ / j	$LCP[i]$ / $LCP[r[j]]$	
1	0	1	a a b a a b a a b b a
4	1	1	a b a a b a a b b a
8	2	2	b a a b a a b b a
2	3	6	a a b a a b b a
5	4	5	a b a a b b a
9	5	4	b a a b b a
3	6	3	a a b b a
6	7	2	a b b a
10	8	1	b b a
7	9	0	b a
0	10	0	a

LCP algorithm

- **Rank** r: $r[j]$ = rank of suffix at position j $\quad (r = SUF^{-1})$
- **Permutation** SUF: $SUF[k]$ = position of suffix of rank k

```
Lcp(y, n, SUF, r)
ℓ ← 0
for j ← 0 to n − 1 do
    ℓ ← max{0, ℓ − 1}
    if r[j] > 0 then
        i ← SUF[r[j] − 1]
        while y[i + ℓ] = y[j + ℓ] do
            ℓ ← ℓ + 1
        LCP[r[j]] ← ℓ
LCP[0] ← 0
LCP[n] ← 0
return LCP
```

- **Running time**: $O(n)$

Complexity of LCP computation

- **Overall LCP computation**
  in time $O(n)$ for suffixes in lexicographic order, *i.e.* for the $n+1$ external nodes of the binary tree
  in time $O(n)$ for the other n nodes of the binary tree by Lemma 1

Proposition 2 *The computation of LCP's of pairs of suffixes used in the binary search can be done in time $O(n)$ with $O(n)$ memory space after suffixes are sorted.*

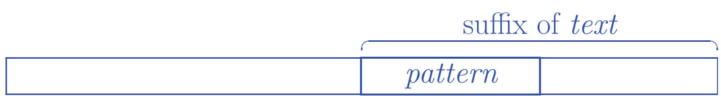
Suffix Trees

MAXIME CROCHEMORE

King's College London

Maxime.Crochemore@kcl.ac.uk
http://www.dcs.kcl.ac.uk/staff/mac/

Implementation of indexes



Implementation with efficient data structures

⋆ **Suffix Trees**
digital trees, PATRICIA tree (compact trees)

⋆ **Suffix Automata or DAWG's**
minimal automata, compact automata

Implementation with efficient algorithm

⋆ **Suffix Arrays**
binary search in the ordered list of suffixes

Suffixes

Text $y \in \Sigma^*$

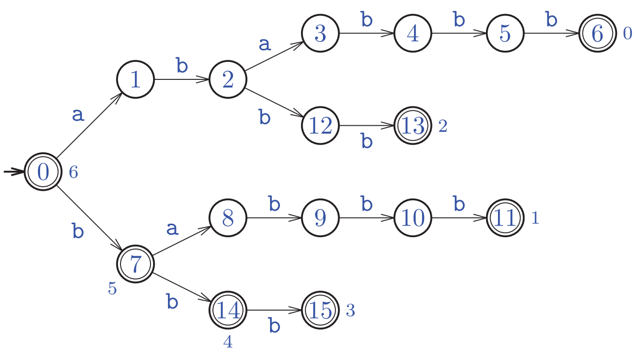
- ⋆ $\mathit{Suff}(y)$ = set of suffixes of y,
- ⋆ card $\mathit{Suff}(y) = |y| + 1$
- ⋆ $\mathit{Suff}(\texttt{ababbb})$

i	0	1	2	3	4	5
$y[i]$	`a`	`b`	`a`	`b`	`b`	`b`

suffix	position	
`ababbb`	0	
`babbb`	1	
`abbb`	2	
`bbb`	3	
`bb`	4	
`b`	5	
ε	6	(empty string)

Trie of suffixes

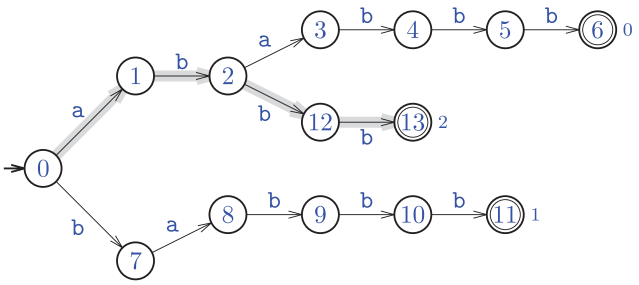
- ⋆ $\mathcal{T}(y)$ = digital tree which branches are labeled by suffixes of y
  = tree-like deterministic automaton accepting $\mathit{Suff}(y)$
- ⋆ **Nodes**
  identified with factors (subwords) of y
- ⋆ **Terminal nodes**
  identified with suffixes of y, output = position of the suffix
- ⋆ **Suffix trie of** `ababbb`



Forks

Insertion of $u = y[i \,.\,.\, n-1]$ in the structure accepting longer suffixes

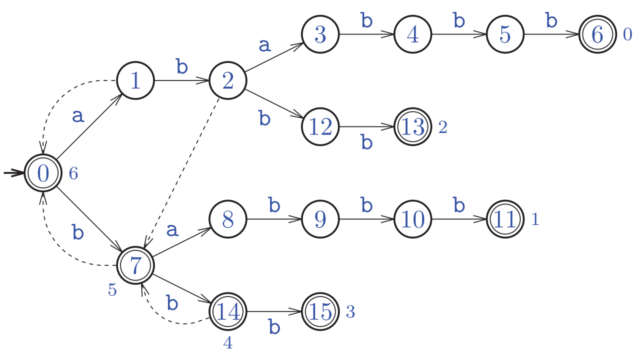
- ⋆ **Head** of u: longest prefix $y[i \,.\,.\, k-1]$ of u occurring before i
- ⋆ **Tail** of u: rest $y[k \,.\,.\, n-1]$ of suffix u
- ⋆ $y =$ `ababbb`; head of `abbb` is `ab`; tail of `abbb` is `bb`



- ⋆ **Fork**
  any node that has outdegree 2 at least,
  or that both has outdegre 1 and is terminal
- ⋆ **Note**: the node associated with the head of u is a fork
  initial node is a fork iff y non empty

Suffix link

- ⋆ **Function** s_y, *suffix link*
  if node p identified with factor av, $a \in \Sigma$, $v \in \Sigma^*$
  $s_y(p) = q$, node identified with v



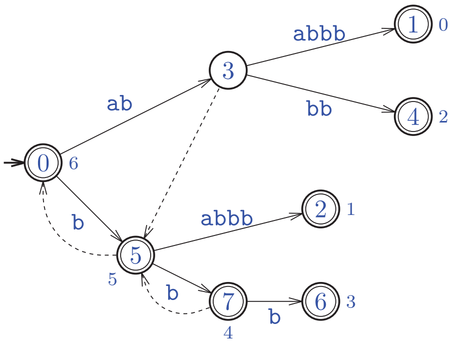
- ⋆ **Use**
  creates shortcuts used to accelerate heads computations
- ⋆ **Useful for forks only**
  undefined on initial node
- ⋆ **Note**: if p is a fork, so is $s_y(p)$

Suffix Tree

Text $y \in \Sigma^*$ of length n

$\mathcal{S}(y)$ suffix tree of y: compact trie accepting *Suff*(y)

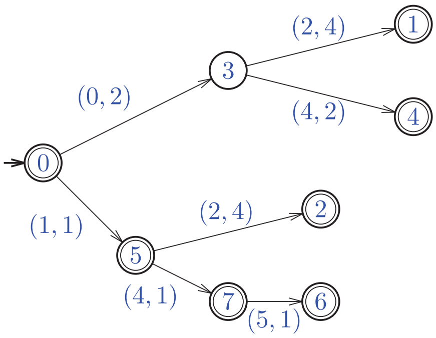
- ⋆ **Definition**
  tree obtained from the suffix trie of y by deleting all nodes having outdegree 1 that are not terminal
- ⋆ **Edges labeled by factors** of y instead of letters



- ⋆ **Number of nodes**: no more than $2n$ (if $n > 0$)
  because all internal nodes have two children at least
  and there are at most n external nodes

Labels of edges

- ⋆ **Labels represented by pairs** $(pos, Length)$



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

- ⋆ Requires to have y in main memory
- ⋆ **Size of** $\mathcal{S}(y)$: $O(n)$

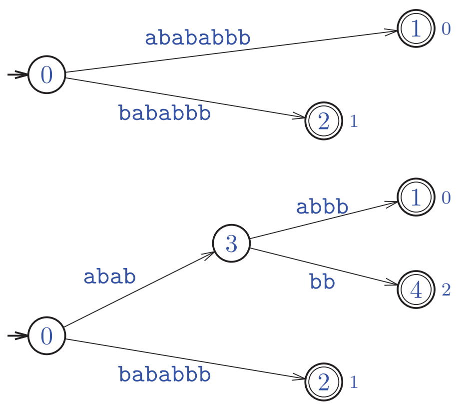
Scheme of suffix tree construction

Suffix-tree(y)
$T \leftarrow$ New-tree()
for $i \leftarrow 0$ **to** $n-1$ **do**
  find *fork* of head of $y[i\,..\,n-1]$ using
   Fast-Find from node $s[parent]$ if needed
   and then Slow-Find
  $k \leftarrow$ position of tail of $y[i\,..\,n-1]$
  if $k < n$ **then**
    $q \leftarrow$ New-state()
    $Adj[fork] \leftarrow Adj[fork] \cup \{((k, n-k), q)\}$
    $output[q] \leftarrow i$
  else $output[fork] \leftarrow i$
$output[initial] \leftarrow n$
return T

- ⋆ Adjacency-list representation of labeled arcs

Straight insertion

- ⋆ Insertion of suffix `ababbb` is done by letter comparisons from the initial node (current node)



i	0	1	2	3	4	5	6	7
$y[i]$	a	b	a	b	a	b	b	b

- ⋆ It leads to create node 3 which suffix link is still undefined,
- ⋆ and node 4 associated with suffix `ababbb` at position 2
- ⋆ Head is `abab`, tail is `bb`

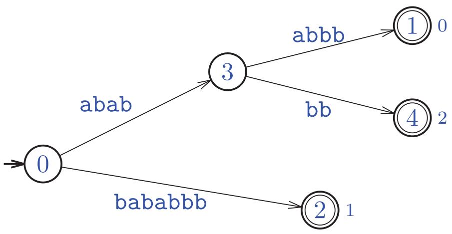
Slow find

SLOW-FIND(p, k)

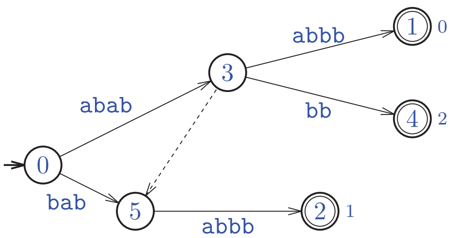
while $k < n$ and TARGET$(p, y[k]) \neq$ NIL **do**
  $q \leftarrow$ TARGET$(p, y[k])$
  $(j, \ell) \leftarrow label(p, q)$
  $i \leftarrow j$
  do $i \leftarrow i + 1$
    $k \leftarrow k + 1$
  while $i < j + \ell$ and $k < n$ and $y[i] = y[k]$
  if $i < j + \ell$ **then**
    $Adj[p] \leftarrow Adj[p] \setminus \{((j, \ell), q)\}$
    $r \leftarrow$ NEW-STATE$()$
    $Adj[p] \leftarrow Adj[p] \cup \{((j, i - j), r)\}$
    $Adj[r] \leftarrow Adj[r] \cup \{((j + i - j, \ell - i + j), q)\}$
    return (r, k)
  $p \leftarrow q$
return (p, k)

New suffix link

- Computing $s[3] = s_y(3)$ remains to find the node associated with `bab`



i	0	1	2	3	4	5	6	7
$y[i]$	a	b	a	b	a	b	b	b



- Arc $(0, (1, 7), 2)$ is split into $(0, (1, 3), 5)$ and $(5, (4, 4), 2)$
- Execution in constant time (here)
- In general, iteration in time proportional to the number of nodes along the path (and not proportional to the length of the string)

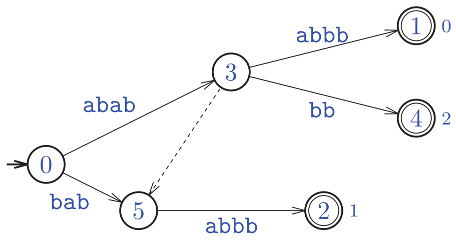
Fast find

$\textsc{Fast-Find}(r, j, k)$

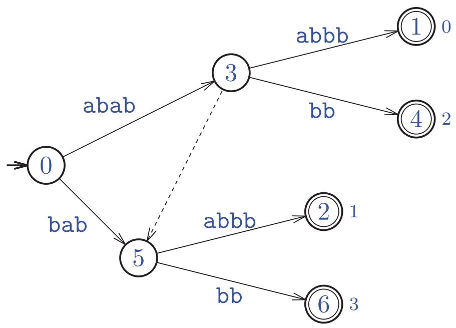
$\triangleright$ computes $\textsc{Target}(r, y[j \,..\, k-1])$
if $j \geq k$ **then**
    return r
else $q \leftarrow \textsc{Target}(r, y[j])$
    $(j', \ell) \leftarrow \mathit{label}(r, q)$
    if $j + \ell \leq k$ **then**
        return $\textsc{Fast-Find}(q, j + \ell, k)$
    else $\mathit{Adj}[r] \leftarrow \mathit{Adj}[r] \setminus \{((j', \ell), q)\}$
        $p \leftarrow \textsc{New-state}()$
        $\mathit{Adj}[r] \leftarrow \mathit{Adj}[r] \cup \{((j, k - j), p)\}$
        $\mathit{Adj}[p] \leftarrow \mathit{Adj}[p] \cup \{((j' + k - j, \ell - k + j), q)\}$
        return p

Next insertion

- End of insertion of suffix `babbb`



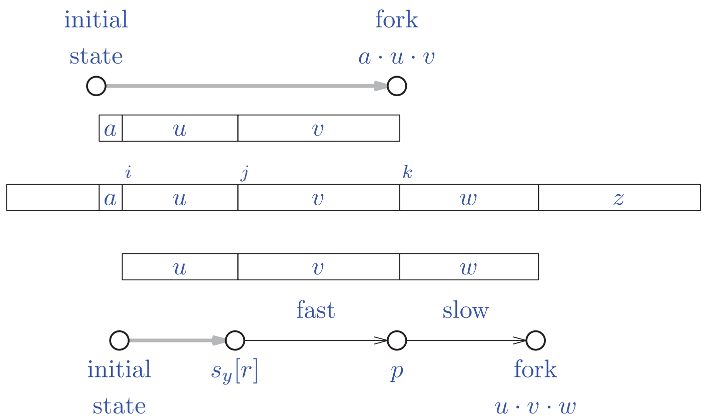
i	0	1	2	3	4	5	6	7
$y[i]$	`a`	`b`	`a`	`b`	`a`	`b`	`b`	`b`



- Execution in constant time
- Head is `bab`, tail is `bb`

Scheme for insertion

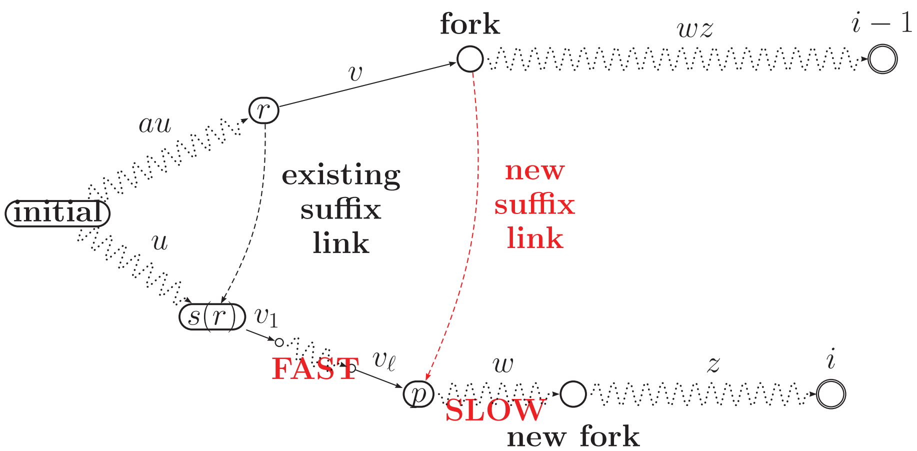
- Scheme for the insertion of suffix $y[i \,.\,.\, n-1] = u \cdot v \cdot w \cdot z$



- It first computes $p = \text{TARGET}(s[r], v)$ with FAST-FIND (if necessary)
- then the fork of the current suffix with SLOW-FIND

Scheme for insertion (continued)

- General scheme for inserting the next suffix in the data structure when the suffix target of the current fork is not defined

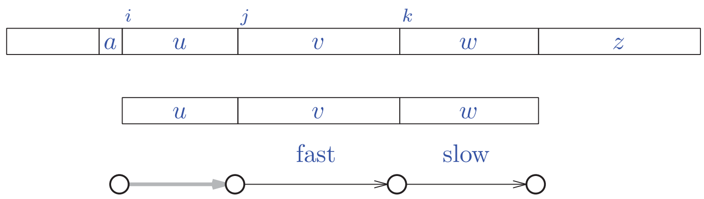


Complete algorithm

```
SUFFIX-TREE(y)
T ← NEW-TREE()
s[initial[T]] ← initial[T]
(fork, k) ← (initial[T], 0)
for i ← 0 to n − 1 do
    k ← max{k, i}
    if s[fork] = NIL then
        r ← parent of fork
        (j, ℓ) ← label(r, fork)
        if r = initial[T] then
            ℓ ← ℓ − 1
        s[fork] ← FAST-FIND(s[r], k − ℓ, k)
    (fork, k) ← SLOW-FIND(s[fork], k)
    if k < n then
        q ← NEW-STATE()
        Adj[fork] ← Adj[fork] ∪ {((k, n − k), q)}
        output[q] ← i
    else output[fork] ← i
output[initial] ← n
return T
```


Running time

- ⋆ Scheme for insertion



- ⋆ Main iteration increments i, which never decreases
- ⋆ Iteration in FAST-FIND increments j, which never decreases
- ⋆ Iteration in SLOW-FIND increments k, which never decreases
- ⋆ Basic operations run in constant time or in time $O(\log \operatorname{card} \Sigma)$

Theorem 1 *Execution of* SUFFIX-TREE$(y) = \mathcal{S}(y)$ *takes* $O(|y| \times \log \operatorname{card} \Sigma)$ *time in the comparison model.*

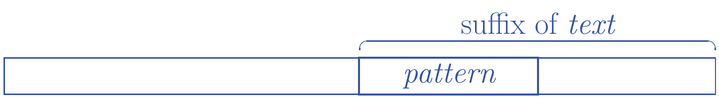
Suffix Automata

Maxime Crochemore

King's College London

`Maxime.Crochemore@kcl.ac.uk`
`http://www.dcs.kcl.ac.uk/staff/mac/`

Implementation of indexes



Implementation with efficient data structures

- ⋆ **Suffix Trees**
  digital trees, PATRICIA tree (compact trees)
- ⋆ **Suffix Automata or DAWG's**
  minimal automata, compact automata

Implementation with efficient algorithm

- ⋆ **Suffix Arrays**
  binary search in the ordered list of suffixes

Suffixes

Text $y \in \Sigma^*$

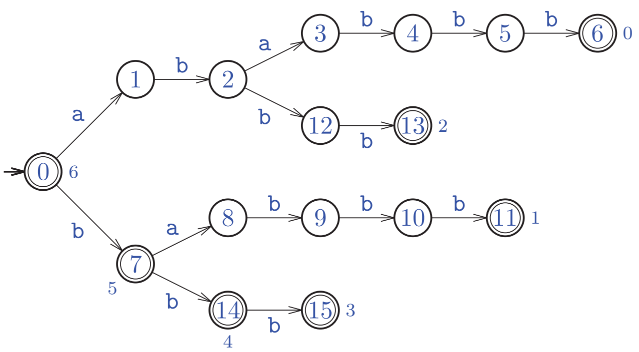
- $\mathit{Suff}(y)$ = set of suffixes of y,
- card $\mathit{Suff}(y) = |y| + 1$
- $\mathit{Suff}(\texttt{ababbb})$

i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

							position	
	a	b	a	b	b	b	0	
		b	a	b	b	b	1	
			a	b	b	b	2	
				b	b	b	3	
					b	b	4	
						b	5	
						ε	6	(empty string)

Trie of suffixes

- $\mathcal{T}(y)$ = digital tree which branches are labeled by suffixes of y
  = tree-like deterministic automaton accepting $\mathit{Suff}(y)$
- **Nodes**
  identified with factors (subwords) of y
- **Terminal nodes**
  identified with suffixes of y, output = position of the suffix
- **Suffix trie of** `ababbb`

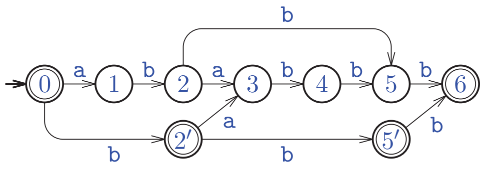


Suffix Automaton

Text $y \in \Sigma^*$ of length n

$\mathcal{A}(y)$ = minimal deterministic automaton accepting $\mathit{Suff}(y)$

⋆ **Minimization** of the trie of suffixes



⋆ States are classes of factors (subwords) of y

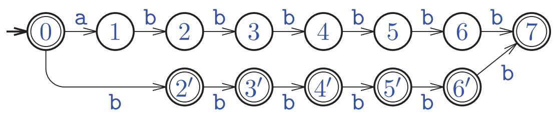
⋆ **Size**:

$n + 1 \leq \#states \leq 2n - 1$

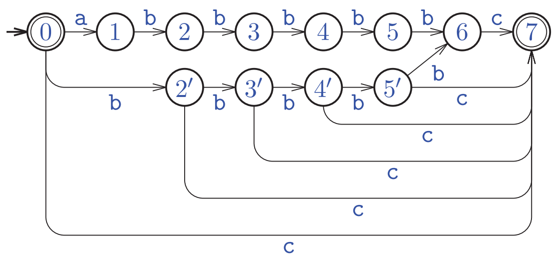
$n \leq \#arcs \leq 3n - 4$

Maximal size

⋆ **Maximal number of states**

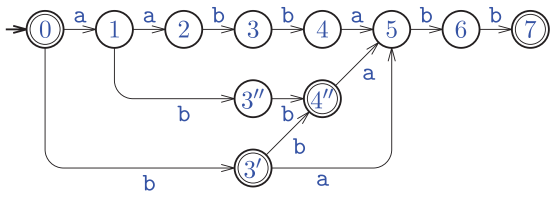


⋆ **Maximal number of arcs**



Suffix link

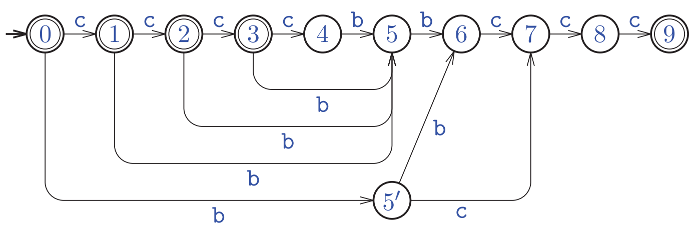
- **Function** f_y, *suffix link*
  let $p = \text{TARGET}(\mathit{initial}[\mathcal{A}], v)$, $v \in \Sigma^+$
  $f_y(p) = \text{TARGET}(\mathit{initial}[\mathcal{A}], u)$, where u is the longest suffix of v occurring in a different right context

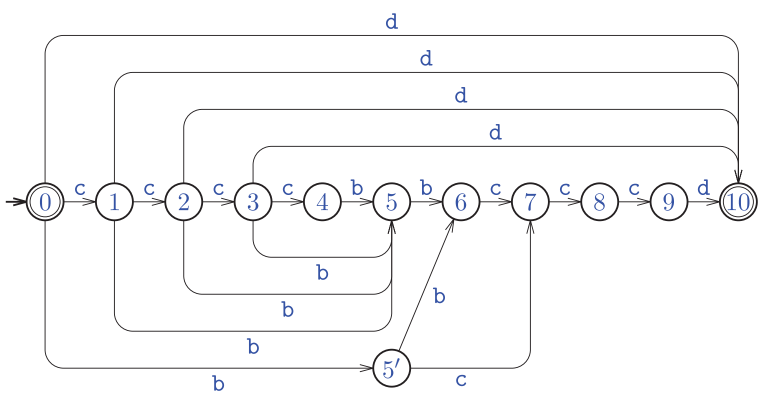


- $f[1] = 0$, $f[2] = 1$, $f[3] = 3''$, $f[3''] = 3'$, $f[3'] = 0$,
  $f[4] = 4''$, $f[4''] = 3'$, $f[5] = 1$, $f[6] = 3''$, $f[7] = 4''$.
- **Suffix path**
  example for state 7: $\langle 7, 4'', 3', 0\rangle$, sequence of terminal states
- **Use**
  same but more efficient than suffix link in suffix trees

Construction—one step (1)

- **From** $\mathcal{A}$(ccccbbccc) **to** $\mathcal{A}$(ccccbbcccd)

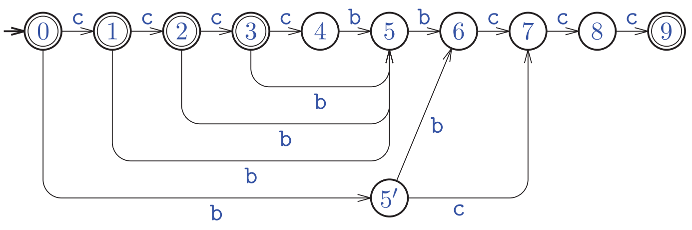


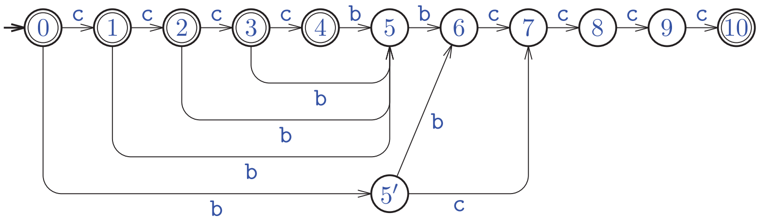


- New arcs from states of the suffix path $\langle 9, 3, 2, 1, 0\rangle$.

Construction—one step (2)

★ **From $\mathcal{A}$(ccccbbccc) to $\mathcal{A}$(ccccbbcccc)**

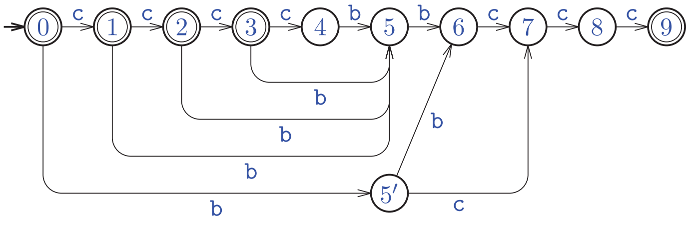


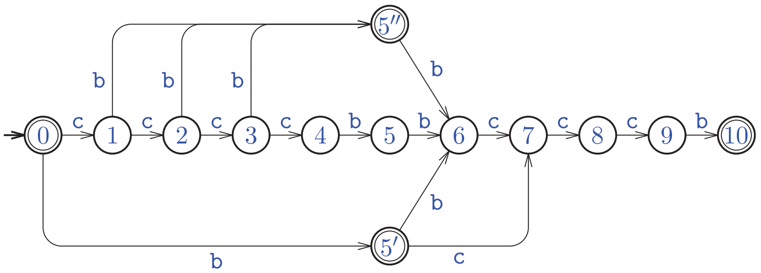


★ Link $3 = f[9]$ and solid arc $(3, \texttt{c}, 4)$ (not a shortcut)
then, $f[10] = \textsc{Target}(3, \texttt{c}) = 4$ that becomes a terminal state

Construction—one step (3)

★ **From $\mathcal{A}$(ccccbbccc) to $\mathcal{A}$(ccccbbcccb)**





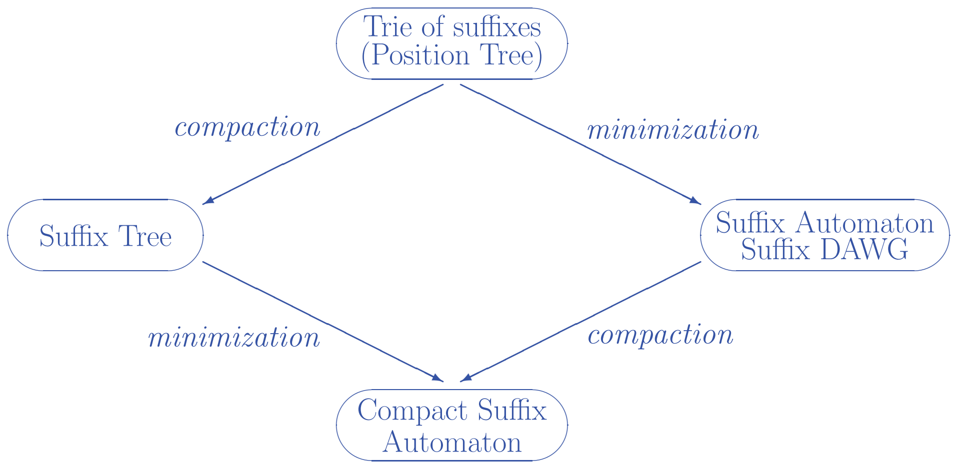
★ Link $3 = f[9]$, non-solid arc $(3, \texttt{b}, 5)$, cccb suffix but ccccb not
state 5 is cloned into $5'' = f[10] = f[5]$, $f[5''] = 5'$
arcs $(3, \texttt{b}, 5)$, $(2, \texttt{b}, 5)$ et $(1, \texttt{b}, 5)$ are redirected onto $5''$

Operations on indexes

Text y of length n

- **Index implemented by suffix tree or suffix automaton of y**
  memory space $O(n)$, construction time $O(n \times \log \operatorname{card} \Sigma)$
- **String matching**
  searching y for x of length m: time $O(m \times \log \operatorname{card} \Sigma)$
  number of occurrences of x in y: same complexity after $O(n)$ preprocessing
- **All occurrences**
  finding all occurrences of x in y: time $O(m \times \log \operatorname{card} \Sigma) + |output|)$
- **Repetitions**
  computing a longest factor of y occurring at least k times: time $O(n)$
- **Marker**
  computing a shortest factor of y occurring exactly once: time $O(n)$

Saving space

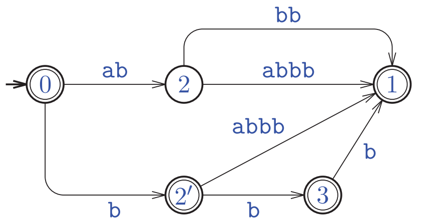


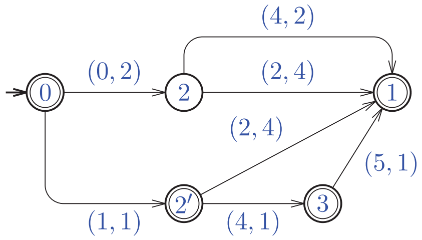
Compact Suffix Automaton

Text $y \in \Sigma^*$ of length n

$\mathcal{A}^c(y)$ = compact minimal automaton accepting $Suff(y)$

⋆ **Compaction** of $\mathcal{A}(y)$, or **minimization** of $\mathcal{S}(y)$





i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

⋆ **Linear size**: $O(n)$

Direct construction of CSA

⋆ Similar to both
 – Suffix Tree construction
 – Suffix Automaton construction

⋆ Sequential addition of suffixes in the structure from the longest to the shortest

⋆ Used features:
 – "slow-find" and "fast-find" procedures
 – suffix links
 – solid and non-solid arcs
 – state splitting
 – re-directions of arcs

⋆ **Complexity**: $O(n \log \operatorname{card} \Sigma)$ time, $O(n)$ space
50% **saved** on space of Suffix Automaton