Text Algorithms

Jaak Vilo

2015 fall
Topics

• Exact matching of one pattern(string)
• Exact matching of multiple patterns
• Suffix trie and tree indexes
  – Applications
• Suffix arrays
• Inverted index
• Approximate matching
Exact pattern matching

• $S = s_1 s_2 \ldots s_n$ (text) \hspace{1cm} |S| = n (length)

• $P = p_1 p_2 \ldots p_m$ (pattern) \hspace{1cm} |P| = m

• $\Sigma$ - alphabet \hspace{1cm} |$\Sigma$| = c

• Does $S$ contain $P$?
  – Does $S = S' \ P \ S''$ for some strings $S'$ ja $S''$?
  – Usually $m \ll n$ and $n$ can be (very) large
Find occurrences in text
Algorithms

One-pattern
• Brute force
• Knuth-Morris-Pratt
• Karp-Rabin
• Shift-OR, Shift-AND
• Boyer-Moore
• Factor searches

Multi-pattern
• Aho Corasick
• Commentz-Walter

Indexing
• Trie (and suffix trie)
• Suffix tree
Animations


- EXACT STRING MATCHING ALGORITHMS
  Animation in Java

- Christian Charras - Thierry Lecroq
  Laboratoire d'Informatique de Rouen
  Université de Rouen
  Faculté des Sciences et des Techniques
  76821 Mont-Saint-Aignan Cedex
  FRANCE

Brute force: BAB in text?

A B A C A B A B B A B B B A
B A B
Brute Force

Identify the first mismatch!

Question:

- Problems of this method? 😞
- Ideas to improve the search? 😊
Brute force

Algorithm Naive

Input: Text $S[1..n]$ and pattern $P[1..m]$

Output: All positions $i$, where $P$ occurs in $S$

```
for ( i=1 ; i <= n-m+1 ; i++ )
    for ( j=1 ; j <= m ; j++ )
        if( $S[i+j-1]$ != $P[j]$ ) break ;
    if ( j > m ) print i ;
```

attempt 1:
gcatcgacagaggtatacagtacg
GCAg....

attempt 2:
gcatcgacagaggtatacagtacg
g........

attempt 3:
gcatcgacagaggtatacagtacg
g........

attempt 4:
gcatcgacagaggtatacagtacg
g........

attempt 5:
gcatcgacagaggtatacagtacg
g........

attempt 6:
gcatcgacagaggtatacagtacg
GCAGAGAG

attempt 7:
gcatcGCAGAGAGtatacagtacg
g........
Brute force or NaiveSearch

```plaintext
1  function NaiveSearch(string s[1..n], string sub[1..m])
2    for i from 1 to n-m+1
3       for j from 1 to m
4         if s[i+j-1] ≠ sub[j]
5           jump to next iteration of outer loop
6         return i
7    return not found
```
int bf_2( char* pat, char* text, int n ) /* n = textlen */
{
    int m, i, j;
    int count = 0;
    m = strlen(pat);

    for (i=0; i + m <= n; i++) {
        for( j=0; j < m && pat[j] == text[i+j]; j++) {
            if( j == m )
                count++;
        }
    }

    return(count);
}
```c
int bf_1( char* pat, char* text )
{
    int m;
    int count = 0;
    char *tp;

    m = strlen(pat);
    tp=text ;

    for( ; *tp ; tp++ ) {
        if( strncmp( pat, tp, m ) == 0 ) {
            count++ ;
        }
    }

    return( count );
}
```
Main problem of Naive

• For the next possible location of P, check again the same positions of S
Goals

• Make sure only a constant nr of comparisons/operations is made for each position in S
  – Move (only) from left to right in S

  – How?
  – After a test of $S[i] <> P[j]$ what do we now?
Knuth-Morris-Pratt

• Make sure that no comparisons “wasted”

• After such a mismatch we already know exactly the values of green area in S!
Knuth-Morris-Pratt

• Make sure that no comparisons “wasted”

• P – longest suffix of any prefix that is also a prefix of a pattern

• Example: \( \text{ABCABD} \)
Automaton for ABCABD
Automaton for ABCABD

Fail:

```
0 1 1 1 2 3 1
```

Pattern:

```
A  B  C  A  B  D
1 2 3 4 5 6
```
KMP matching

**Input:** Text S[1..n] and pattern P[1..m]

**Output:** First occurrence of P in S (if exists)

```
i = 1; j = 1;
initfail(P) // Prepare fail links
repeat
  if j == 0 or S[i] == P[j]
    then i++, j++ // advance in text and in pattern
  else j = fail[j] // use fail link
until j > m or i > n
if j > m then report match at i-m
```
Initialization of fail links

Algorithm: KMP_Initfail
Input: Pattern P[1..m]
Output: fail[] for pattern P

i=1, j=0 , fail[1]= 0
repeat
  if j==0 or P[i] == P[j]
    then i++ , j++ , fail[i] = j
  else j = fail[j]
until i>=m
Initialization of fail links

\[ i = 1, j = 0, \text{ fail}[1] = 0 \]

repeat
  if \( j = 0 \) or \( \text{ P}[i] = \text{ P}[j] \) then
    \( i++ \), \( j++ \), \text{ fail}[i] = j \\
  else
    \( j = \text{ fail}[j] \)
until \( i >= m \)
Time complexity of KMP matching?

**Input:** Text S[1..n] and pattern P[1..m]

**Output:** First occurrence of P in S (if exists)

\[ i=1; \quad j=1; \]
\[ \text{initfail}(P) \quad // \quad \text{Prepare fail links} \]
\[ \text{repeat} \]
\[ \quad \text{if} \quad j==0 \quad \text{or} \quad S[i] == P[j] \]
\[ \quad \quad \text{then} \quad i++ \quad , \quad j++ \quad // \quad \text{advance in text and in pattern} \]
\[ \quad \quad \text{else} \quad j = \text{fail}[j] \quad // \quad \text{use fail link} \]
\[ \quad \text{until} \quad j>m \quad \text{or} \quad i>n \]
\[ \text{if} \quad j>m \quad \text{then} \quad \text{report match at} \quad i-m \]
Analysis of time complexity

- At every cycle either i and j increase by 1
- Or j decreases (j=fail[j])

- i can increase n (or m) times
- Q: How often can j decrease?
  - A: not more than nr of increases of i

- **Amortised analysis:** $O(n)$, preprocess $O(m)$
Karp-Rabin


- Compare in $O(1)$ a hash of $P$ and $S[i..i+m-1]$

- Goal: $O(n)$.

- $f( h(T[i..i+m-1]) \rightarrow h(T[i+1..i+m]) ) = O(1)$
Karp-Rabin

- Compare in $O(1)$ a hash of $P$ and $S[i..i+m-1]$

  $h(T[i+1..i+m])$

  $h(P)$

- Goal: $O(n)$.

- $f( h(T[i..i+m-1]) \rightarrow h(T[i+1..i+m]) ) = O(1)$
Hash

• “Remove” the effect of $T[i]$ and “Introduce” the effect of $T[i+m]$ – in $O(1)$

• Use base $|\Sigma|$ arithmetics and treat characters as numbers

• In case of hash match – check all m positions
• Hash collisions => Worst case $O(nm)$
Let’s use numbers

- $T = 57125677$
- $P = 125$ (and for simplicity, $h = 125$)

- $H(T[1]) = 571$
- $H(T[2]) = (571 - 5*100)*10 + 2 = 712$

- $H(T[3]) = (H(T[2]) - \text{ord}(T[1])*10^m)*10 + T[3+m-1]$
hash

• c – size of alphabet

• $H_{Si} = H( S[i..i+m-1] )$

• $H( S[i+1..i+m] ) = ( H_{Si} - \text{ord}(S[i]) \cdot c^{m-1} ) \cdot c + \text{ord}( S[i+m] )$

• Modulo arithmetic – to fit value in a word!
\[ hash(w[0..m-1]) = (w[0] \times 2^{m-1} + w[1] \times 2^{m-2} + \cdots + w[m-1] \times 2^0) \mod q \]
Karp-Rabin

Input: Text S[1..n] and pattern P[1..m]
Output: Occurrences of P in S
1. c=20; /* Size of the alphabet, say nr. of aminoacids */
2. q = 33554393 /* q is a prime */
3. $cm = c^{m-1} \mod q$
4. hp = 0 ; hs = 0
5. for i = 1 .. m do hp = ( hp*c + ord(p[i]) ) mod q // H(P)
6. for i = 1 .. m do hs = ( hp*c + ord(s[i]) ) mod q // H(S[1..m])
7. if hp == hs and P == S[1..m] report match at position
8. for i=2 .. n-m+1
9. hs = ( (hs - ord(s[i-1])*cm) * c + ord(s[i+m-1]) ) mod q
10. if hp == hs and P == S[i..i+m-1]
11. report match at position i
More ways to ensure $O(n)$?
Shift-AND / Shift-OR

• Ricardo Baeza-Yates, Gaston H. Gonnet
A new approach to text searching
[ACM Digital Library: http://doi.acm.org/10.1145/135239.135243] [DOI]

• PDF
Bit-operations

• Maintain a set of all prefixes that have so far had a perfect match

• On the next character in text update all previous pointers to a new set

• Bit vector: for every possible character
State: which prefixes match?
Move to next:
Track positions of prefix matches

Shift left $<<$

Bitwise AND

Mask on char $T[i]$
Vectors for every char in $\Sigma$

- $P=$aste

```
astebcd..z
100000...
010000...
001000...
000100...
000100...
```
• \( T = \text{lasteaed} \)

lasteaed
0 1
0 0
0 0
0 0
\[ T = \text{lasteaed} \]

\[
\begin{align*}
\text{lasteaed} & \quad 010 \\
001 & \\
000 & \\
000 &
\end{align*}
\]
- $T = \text{lasteaed}$

```plaintext
lasteaed
0 1 0 0 0 1
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 1 0
```
• $T = \text{lasteaed}$
The Shift Or algorithm uses bitwise techniques. Let $R$ be a bit array of size $m$. Vector $R_j$ is the value of the array $R$ after text character $y[j]$ has been processed (see figure 5.1). It contains informations about all matches of prefixes of $x$ that end at position $j$ in the text for $0 < i \leq m-1$:

$$R_j[i] = \begin{cases} 
0 & \text{if } x[0, i] = y[j-i, j], \\
1 & \text{otherwise.}
\end{cases}$$

![Diagram showing the meaning of vector $R_j$.](http://www-igm.univ-mlv.fr/~lecroq/string/node6.html)

**Figure 5.1:** Meaning of vector $R_j$.

The vector $R_{j+1}$ can be computed after $R_j$ as follows. For each $R_j[i]=0$:

$$R_{j+1}[i+1] = \begin{cases} 
0 & \text{if } x[i+1] = y[j+1], \\
1 & \text{otherwise,}
\end{cases}$$

and

$$R_{j+1}[0] = \begin{cases} 
0 & \text{if } x[0] = y[j+1], \\
1 & \text{otherwise.}
\end{cases}$$

If $R_{j+1}[m-1]=0$ then a complete match can be reported.
The C code

```c
int preSo(char *x, int m, unsigned int S[]) {
    unsigned int j, lim;
    int i;
    for (i = 0; i < ASIZE; ++i)
        S[i] = ~0;
    for (lim = i = 0, j = 1; i < m; ++i, j <<= 1) {
        S[x[i]] &= ~j;
        lim |= j;
    }
    lim = ~(lim>>1);
    return(lim);
}

void SO(char *x, int m, char *y, int n) {
    unsigned int lim, state;
    unsigned int S[ASIZE];
    int j;
    if (m > WORD)
        error("SO: Use pattern size <= word size");

    /* Preprocessing */
    lim = preSo(x, m, S);

    /* Searching */
    for (state = ~0, j = 0; j < n; ++j) {
        state = (state<<1) | S[y[j]];
        if (state < lim)
            OUTPUT(j - m + 1);
    }
}
```
The example

Searching phase

As $R_{12}[7]=0$ it means that an occurrence of $x$ has been found at position $12-8+1=5$. 

http://www-igm.univ-mlv.fr/~lecroq/string/node6.html
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst case</th>
<th>Ave. Case</th>
<th>Preprocess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute force</td>
<td>$O(mn)$</td>
<td>$O(n \times (1+1/</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>$O(mn)$</td>
<td>$O(n)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td></td>
<td>$O(n/m)$ ?</td>
<td></td>
</tr>
<tr>
<td>BM Horspool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factor search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift-OR</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(m</td>
</tr>
</tbody>
</table>

- **Algorithm**
- **Worst case**
- **Ave. Case**
- **Preprocess**
2.6 Other algorithms and references

Fig. 2.22. Map of experimental efficiency for different string matching algorithms.
• R. Boyer, S. Moore: A fast string searching algorithm. *CACM* 20 (1977), 762-772 [PDF]
Find occurrences in text

• Have we missed anything?
Find occurrences in text

• What have we learned if we test for a potential match from the end?
Our search algorithm may be specified as follows:

\[
\text{stringlen} \leftarrow \text{length of } string.
\]
\[
i \leftarrow \text{patlen}.
\]
\textit{top: } if \(i > \text{stringlen}\) then return false.
\[
j \leftarrow \text{patlen}.
\]
\textit{loop: } if \(j = 0\) then return \(j + 1\).
\[
\text{if } string(i) = pat(j) \text{ then}
\]
\[
j \leftarrow j - 1.
\]
\[
i \leftarrow i - 1.
\]
\[
goto \text{loop}.
\]
\[
close;
\]
\[
i \leftarrow i + \max(\text{delta}_1 (string(i)), \text{delta}_2 (j)).
\]
\[
goto \text{top}.
\]

If the above algorithm returns false, then \(pat\) does not occur in \(string\). If the algorithm returns a number, then it is the position of the left end of the first occurrence of \(pat\) in \(string\).
Find occurrences in text
Bad character heuristics
maximal shift on $S[i]$

$$\delta_1( S[i] ) = |m| \quad \text{if pattern does not contain } S[i]$$
$$\text{patlen-j} \quad \text{max } j \text{ so that } P[j] = S[i]$$

First x in pattern (from end)
void bmInitOcc() {
    char a; int j;
    for(a=0; a<alphabetsize; a++)
        occ[a]=-1;
    for (j=0; j<m; j++) {
        a=p[j];
        occ[a]=j;
    }
}
Good suffix heuristics

\[ \delta_2( S[i] ) - \text{minimal shift so that matched region is fully covered or that the suffix of match is also a prefix of } P \]
Boyer-Moore algorithm

Input: Text $S[1..n]$ and pattern $P[1..m]$
Output: Occurrences of $P$ in $S$

preprocess_BM() // delta1 and delta2
i=m
while $i \leq n$
    for( $j=m$; $j>0$ and $P[j]==S[i-m+j]$; $j--$ ) ;
    if $j==0$ report match at position $i-m+1$
    $i = i + \max( \delta_1[S[i]], \delta_2[j] )$
• http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm


• Animation: http://www-igm.univ-mlv.fr/~lecroq/string/
Simplifications of BM

• There are many variants of Boyer-Moore, and many scientific papers.
• On average the time complexity is sublinear
• Algorithm speed can be improved and yet simplify the code.
• It is useful to use the last character heuristics (Horspool (1980), Baeza-Yates(1989), Hume and Sunday(1991)).
Algorithm BMH (Boyer-Moore-Horspool)

- RN Horspool - [Practical Fast Searching in Strings](#)
  *Software - Practice and Experience, 10(6):501-506 1980*

Input: Text S[1..n] and pattern P[1..m]
Output: occurrences of P in S
1. for a in Σ do delta[a] = m
2. for j=1..m-1 do delta[P[j]] = m-j

3. i=m
4. while i <= n
5.    if S[i] == P[m]
6.       j = m-1
7.       while ( j>0 and P[j]==S[i-m+j] ) j = j-1 ;
8.       if j==0 report match at i-m+1
9.       i = i + delta[ S[i] ]
String Matching: Horspool algorithm

• How the comparison is made?

Text :
Pattern :

From right to left: suffix search

• Which is the next position of the window?

Text :
Pattern :

It depends of where appears the last letter of the text, say it ‘a’, in the pattern:

Then it is necessary a preprocess that determines the length of the shift.
Algorithm Boyer-Moore-Horspool-Hume-Sunday (BMHHS)

- Use delta in a tight loop
- If match (delta==0) then check and apply original delta d

Input: Text S[1..n] and pattern P[1..m]
Output: occurrences of P in S

1. for a in Σ do delta[a] = m
2. for j=1..m-1 do delta[P[j]] = m-j
3. d = delta[ P[ m ] ]; // memorize d on P[m]
4. delta[ P[ m ] ] = 0; // ensure delta on match of last char is 0
5. for ( i=m ; i<= n ; i = i+d )
6. repeat // skip loop
7. t=delta[ S[i] ]; i = i + t
8. until t==0
9. for( j=m-1 ; j> 0 and P[j]==S[i-m+j] ; j = j-1 ) ;
10. if j==0 report match at i-m+1

BMHHS requires that the text is padded by P: S[n+1]..S[n+m] = P
(in order for the algorithm to finish correctly – at least one occurrence!).
• **Daniel M. Sunday**: A very fast substring search algorithm [PDF] *Communications of the ACM August 1990, Volume 33 Issue 8*

• **Loop unrolling:**
  • Avoid too many loops (each loop requires tests) by just repeating code within the loop.
  • Line 7 in previous algorithm can be replaced by:

```plaintext
7.   i += delta[ S[i] ];
     i += delta[ S[i] ];
     i += (t = delta[ S[i] ]); 
```
2.1.5 Popularity

Although providing a high performance, the degree to which the Boyer-Moore approach has been put into practice may have been curbed to a certain extent by conceptual difficulties in the preprocessing, particularly with the match heuristic. Horspool (1980) observed that “many programmers may not believe that the Boyer and Moore algorithm (if they have heard of it) is a truly practical approach.” Sedgewick (1983) also noted that “both the Knuth-Morris-Pratt and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used.” In fact, Morris discovered that after a few months his initial search implementation in a text editor, being too difficult to understand by other systems programmers, had been ruined by various gratuitous ‘fixes’ (Knuth, Morris and Pratt, 1977).

Hume and Sunday (1991) add that “partially because the best algorithms presented in the literature are difficult to understand and to implement, knowledge of fast and practical algorithms is not commonplace.” Arguing for a more consistent approach to algorithm development, and against “widespread chaotic algorithm presentation,” Woude (1989) asserts that the Knuth-Morris-Pratt preprocessing should be no more difficult to understand than the actual search procedure itself. There have also been some attempts of late to popularise the Boyer-Moore approach, particularly in its simplified form (e.g. Menico, 1989). As a step towards elucidating this field, Hume and Sunday (1991) have presented a means of classifying the various string-matching algorithms based on the Knuth-Morris-Pratt and Boyer-Moore approaches. A taxonomy of string-matching algorithms, including these two aforementioned approaches, has also been put forward by Watson and Zwaan (1992).
Forward-Fast-Search: Another Fast Variant of the Boyer-Moore String Matching Algorithm

• The Prague Stringology Conference '03
• Domenico Cantone and Simone Faro
• Abstract: We present a variation of the Fast-Search string matching algorithm, a recent member of the large family of Boyer-Moore-like algorithms, and we compare it with some of the most effective string matching algorithms, such as Horspool, Quick Search, Tuned Boyer-Moore, Reverse Factor, Berry-Ravindran, and Fast-Search itself. All algorithms are compared in terms of run-time efficiency, number of text character inspections, and number of character comparisons. It turns out that our new proposed variant, though not linear, achieves very good results especially in the case of very short patterns or small alphabets.

• http://cs.felk.cvut.cz/psc/event/2003/p2.html
• PS.gz (local copy)
Factor based approach

• Optimal average-case algorithms
  – Assuming independent characters, same probability

• Factor – a substring of a pattern
  – Any substring
  – (how many?)
Factor based approach

simple. It is shown in Figure 2.13. Suppose that we have read backward a factor $u$ of the pattern, and that we failed on the next letter $\sigma$. This means that the string $\sigma u$ is no longer a factor of $p$, so no occurrence of $p$ can contain $\sigma u$, and we can safely shift the window to after $\sigma$.

Fig. 2.13. Basic idea for shifting the window with the factor search approach. If we failed to recognize a factor of the pattern on $\sigma$, then $\sigma u$ is not a factor of the pattern and the window can be safely shifted after $\sigma$. 
Factor searches

Do not compare characters, but find the longest match to any subregion of the pattern.
Examples

• Backward DAWG Matching (BDM)
  – Crochemore et al 1994

• Backward Nondeterministic DAWG Matching (BNDM)
  – Navarro, Raffinot 2000

• Backward Oracle Matching (BOM)
  – Allauzen, Crochemore, Raffinot 2001
Backward DAWG Matching BDM

Suffix automaton recognises all factors (and suffixes) in $O(n)$

Fig. 2.14. Basic search of the BDM algorithm with the suffix automaton. The variable $last$ stores the beginning position of the longest suffix of the part read that is also a prefix of the pattern.

Do not compare characters, but find the longest match to any subregion of the pattern.
BNDM – simulate using bitparallelism

Bits – show where the factors have occurred so far

String matching

with 0 and 1 as with Shift-And. The number 1, representing an active
state at position $j$ of $p$, means that the factor $p_j \ldots p_{j+|u|-1}$ is equal to $u$.
Figure 2.15 shows this relationship. If the pattern is of size less that $w$, then
the set fits in a computer word $D = d_m \ldots d_1$.

![Diagram of bit-parallel factor search]

Fig. 2.15. Bit-parallel factor search. The table $D$ keeps a list of the positions in $p$
where the factor $u$ begins.
BNDM matches an NDA

NDA on the suffixes of ‘announce’
Deterministic version of the same Backward Factor Oracle

Fig. 2.20. Factor oracle for the reverse string of “announce”.
2.6 Other algorithms and references

BNDM – Backward Non-Deterministic DAWG Matching
BOM - Backward Oracle matching

Fig. 2.22. Map of experimental efficiency for different string matching algorithms.
String Matching of one pattern

CTACTACTACGTCTATACTGATCGTAGC
TACTACGGTATGACTAA

1. Prefix search
2. Suffix search
3. Factor search
Multiple patterns
Why?

- Multiple patterns
  - Highlight multiple different search words on the page
  - Virus detection – filter for virus signatures
  - Spam filters
  - Scanner in compiler needs to search for multiple keywords
  - Filter out stop words or disallowed words
  - Intrusion detection software
  - Next-generation sequencing produces huge amounts (many millions) of short reads (20-100 bp) that need to be mapped to genome!
- ...
Algorithms

• Aho-Corasick (search for multiple words)
  – Generalization of Knuth-Morris-Pratt

• Commentz-Walter
  – Generalization of Boyer-Moore & AC

• Wu and Manber
  – improvement over C-W

• Additional methods, tricks and techniques
Aho-Corasick (AC)

- Alfred V. Aho and Margaret J. Corasick (Bell Labs, Murray Hill, NJ)
  Efficient string matching. An aid to bibliographic search.
  Communications of the ACM, Volume 18, Issue 6, p333-340 (June 1975)
- ACM:DOI PDF
- ABSTRACT This paper describes a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text. The algorithm consists of constructing a finite state pattern matching machine from the keywords and then using the pattern matching machine to process the text string in a single pass. Construction of the pattern matching machine takes time proportional to the sum of the lengths of the keywords. The number of state transitions made by the pattern matching machine in processing the text string is independent of the number of keywords. The algorithm has been used to improve the speed of a library bibliographic search program by a factor of 5 to 10.

References:

• Generalization of KMP for many patterns
• Text S like before.
• Set of patterns $\mathbf{P} = \{ P_1, \ldots, P_k \}$
• Total length $|\mathbf{P}| = m = \sum_{i=1..k} m_i$
• Problem: find all occurrences of any of the $P_i \in \mathbf{P}$ from S
Idea

1. Create an **automaton** from all patterns

2. Match the automaton

- Use the PATRICIA trie for creating the main structure of the automaton
PATRICIA trie


**Abstract** PATRICIA is an algorithm which provides a flexible means of storing, indexing, and retrieving information in a large file, which is economical of index space and of reindexing time. It does not require rearrangement of text or index as new material is added. It requires a minimum restriction of format of text and of keys; it is extremely flexible in the variety of keys it will respond to. It retrieves information in response to keys furnished by the user with a quantity of computation which has a bound which depends linearly on the length of keys and the number of their proper occurrences and is otherwise independent of the size of the library. It has been implemented in several variations as FORTRAN programs for the CDC-3600, utilizing disk file storage of text. It has been applied to several large information-retrieval problems and will be applied to others.

- [ACM:DOI PDF](#)
Trie

From Wikipedia, the free encyclopedia

In computer science, a trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

The term trie comes from "retrieval." Following the etymology, the inventor, Edward Fredkin, pronounces it [tri] ("tree") [1]. However, it is usually pronounced [traɪ] ("try").[2]
• **Word trie** - a good data structure to represent a set of words (e.g. a dictionary).

• **trie** (data structure)

• **Definition:** A tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.

• See also digital tree, digital search tree, directed acyclic word graph, compact DAWG, Patricia tree, suffix tree.

• **Note:** The name comes from retrieval and is pronounced, "tree."

• To test for a word p, only $O(|p|)$ time is used no matter how many words are in the dictionary...
Trie for P={he, she, his, hers}
Trie for P={he, she, his, hers}
How to search for words like he, sheila, hi. Do these occur in the trie?
1. Create an automaton $M_P$ for a set of strings $P$.
2. Finite state machine: read a character from text, and change the state of the automaton based on the state transitions...
3. Main links: $\text{goto}[j,c]$ - read a character $c$ from text and go from a state $j$ to state $\text{goto}[j,c]$.
4. If there are no $\text{goto}[j,c]$ links on character $c$ from state $j$, use $\text{fail}[j]$.
5. Report the output. Report all words that have been found in state $j$. 
goto[1,i] = 6.

Output table
state output[j]
2 he
5 she, he
7 his
9 hers
AC - matching

**Input:** Text $S[1..n]$ and an AC automaton $M$ for pattern set $P$

**Output:** Occurrences of patterns from $P$ in $S$ (last position)

1. $\text{state} = 0$
2. **for** $i = 1..n$ **do**

3. **while** ($\text{goto}[$state,$S[i]] = \emptyset$) **and** ($\text{fail}[$state$] \neq $state$)$ **do**
4. $\text{state} = \text{fail}[$state$]$
5. $\text{state} = \text{goto}[$state,$S[i]]$
6. **if** (output[state] not empty) **then** report matches output[state] at position $i$
Algorithm Aho-Corasick preprocessing I (TRIE)

Input: \( P = \{ P_1, \ldots, P_k \} \)
Output: \( \text{goto[]} \) and \( \text{parDal} \) output[]
Assume: \( \text{output}(s) \) is empty when a state \( s \) is created;
\( \text{goto}[s,a] \) is not defined.

procedure enter\((a_1, \ldots, a_m)\) /* \( P_i = a_1, \ldots, a_m \) */ begin
1. \( s=0; j=1; \)
2. while \( \text{goto}[s,a_j] \neq \emptyset \) do // follow existing path
   3. \( s = \text{goto}[s,a_j]; \)
   4. \( j = j+1; \)
5. for \( p=j \) to \( m \) do // add new path (states)
   6. \( \text{news} = \text{news}+1; \)
   7. \( \text{goto}[s,a_p] = \text{news}; \)
   8. \( s = \text{news}; \)
   9. \( \text{output}[s] = a_1, \ldots, a_m \)
end

begin
10. \( \text{news} = 0 \)
11. for \( i=1 \) to \( k \) do enter\(( P_i )\)
12. for \( a \in \Sigma \) do
13. \( \text{if} \ \text{goto}[0,a] = \emptyset \ \text{then} \ \text{goto}[0,a] = 0; \)
end
Preprocessing II for AC (FAIL)

queue = ∅
for a ∈ Σ do
  if goto[0,a] ≠ 0 then
    enqueue( queue, goto[0,a] )
    fail[ goto[0,a] ] = 0

while queue ≠ ∅
  r = take( queue )
  for a ∈ Σ do
    if goto[r,a] ≠ ∅ then s = goto[ r, a ]
      enqueue( queue, s ) // breadth first search
      state = fail[r]
      while goto[state,a] = ∅ do state = fail[state]
      fail[s] = goto[state,a]
      output[s] = output[s] + output[ fail[s] ]
Correctness

• Let string t "point" from initial state to state j.

• Must show that fail[j] points to longest suffix that is also a prefix of some word in P.

• Look at the article...
AC matching time complexity

• **Theorem** For matching the $M_p$ on text $S$, $|S|=n$, less than $2n$ transitions within $M$ are made.

• **Proof** Compare to KMP.

• There is at most $n$ goto steps.

• Cannot be more than $n$ Fail-steps.

• In total -- there can be less than $2n$ transitions in $M$. 
Individual node (goto)

- Full table
- List
- Binary search tree(?)
- Some other index?
AC thoughts

• Scales for many strings simultaneously.
• For very many patterns – search time (of grep) improves(??)
  – See Wu-Manber article
• When k grows, then more fail[] transitions are made (why?)
• But always less than n.
• If all goto[j,a] are indexed in an array, then the size is $|M_p| \times |\Sigma|$, and the running time of AC is $O(n)$.
• When k and c are big, one can use lists or trees for storing transition functions.

• Then, $O(n \log(\min(k,c))$ ).
Advanced AC

• Precalculate the next state transition correctly for every possible character in alphabet

• Can be good for short patterns
Problems of AC?

- Need to rebuild on adding / removing patterns
- Details of branching on each node(?)
Commentz-Walter

• Generalization of Boyer-Moore for multiple sequence search

• **Beate Commentz-Walter**  
  *A String Matching Algorithm Fast on the Average*  
  Proceedings of the 6th Colloquium, on Automata, Languages and Programming. Lecture Notes In Computer Science; Vol. 71, **1979**. pp. 118 - 132, Springer-Verlag


C-W description

- Aho and Corasick [AC75] presented a linear-time algorithm for this problem, based on an automata approach. This algorithm serves as the basis for the UNIX tool fgrep. A linear-time algorithm is optimal in the worst case, but as the regular string-searching algorithm by Boyer and Moore [BM77] demonstrated, it is possible to actually skip a large portion of the text while searching, leading to faster than linear algorithms in the average case.
Commentz-Walter [CW79]

- Commentz-Walter [CW79] presented an algorithm for the multi-pattern matching problem that combines the Boyer-Moore technique with the Aho-Corasick algorithm. The Commentz-Walter algorithm is substantially faster than the Aho-Corasick algorithm in practice. Hume [Hu91] designed a tool called gre based on this algorithm, and version 2.0 of fgrep by the GNU project [Ha93] is using it.

- Baeza-Yates [Ba89] also gave an algorithm that combines the Boyer-Moore-Horspool algorithm [Ho80] (which is a slight variation of the classical Boyer-Moore algorithm) with the Aho-Corasick algorithm.
Idea of C-W

- Build a **backward** trie of all keywords
- Match from the end until mismatch...
- Determine the shift based on the combination of heuristics
Horspool for many patterns

Search for ATGTATG,TATG,ATAAT,ATGTG

1. Build the trie of the inverted patterns

2. $l_{\text{min}}=4$

3. *Table of shifts*

4. *Start the search*

---

Slides courtesy of: Xavier Meseguer PeyPOCH (http://wwwlsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

A 1
C 4 (lmin)
G 2
T 1

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

A 1
C 4 (lmin)
G 2
T 1

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA…

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>4 (lmin)</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>

Slides courtesy of: Xavier Messegue Peypoch (http://www.lsi.upc.es/~alggen)
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

A 1
C 4 (lmin)
G 2
T 1
Horspool for many patterns

Search for ATGTATG, TATG, ATAAT, ATGTG

The text ACATGCTATGTGACA...

Short Shifts!

Slides courtesy of: Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
What are the possible limitations for C-W?

• Many patterns, small alphabet – minimal skips

• What can be done differently?
Wu-Manber

- Citeseer: [http://citeseer.ist.psu.edu/wu94fast.html](http://citeseer.ist.psu.edu/wu94fast.html) [Postscript]
- We present a different approach that also uses the ideas of Boyer and Moore. Our algorithm is quite simple, and the main engine of it is given later in the paper. An earlier version of this algorithm was part of the second version of agrep [WM92a, WM92b], although the algorithm has not been discussed in [WM92b] and only briefly in [WM92a]. The current version is used in glimpse [MW94]. The design of the algorithm concentrates on typical searches rather than on worst-case behavior. This allows us to make some engineering decisions that we believe are crucial to making the algorithm significantly faster than other algorithms in practice.
Key idea

• Main problem with Boyer-Moore and many patterns is that, the more there are patterns, the shorter become the possible shifts...

• Wu and Manber: check several characters simultaneously, i.e. increase the alphabet.
• Instead of looking at characters from the text one by one, we consider them in blocks of size $B$.
• $\log_c 2M$; in practice, we use either $B = 2$ or $B = 3$.
• The SHIFT table plays the same role as in the regular Boyer-Moore algorithm, except that it determines the shift based on the last $B$ characters rather than just one character.
Horspool to Wu-Manber

How do we can increase the length of the shifts?

With a table shift of l-mers with the patterns ATGTATG, TATG, ATAAAT, ATGTG

<table>
<thead>
<tr>
<th>1 símbol</th>
<th>2 símbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>4 (lmin)</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>

Slides courtesy of Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
Wu-Manber algorithm

Search for ATGTATG, TATG, ATAAT, ATGTG

into the text: ACATGCTATGTGACATAATA

Experimental length: $\log_{|\Sigma|} 2^{\log_{\text{min}} r}$

Slides courtesy of: Xavier Meseguer Peypoch (http://www.lsi.upc.es/~alggen)
Backward Oracle

• Set Backwards oracle SBDM, SBOM

• Pages 68-72
3.4.3 Set Backward Oracle Matching algorithm

The Set Backward Oracle Matching algorithm (SBOM) [AR99] uses a factor oracle of the set of strings. The factor oracle of \( P \) recognizes at least all the factors of the strings in \( P \). The search algorithm is similar to SDBM. We slide a window of size \( \ell_{\text{min}} \) along the text, reading backward a suffix of the window in the factor oracle. If we fail on a letter \( \sigma \), we can safely shift the window past \( \sigma \). If not, we reach the beginning of the window and verify a subset of \( P \) against the text.

3.4.3.1 Factor oracle of a set of strings

The factor oracle construction on a set of strings resembles the Aho-Corasick automaton construction. The only difference appears when going down the supply path looking for an outgoing transition labeled by \( \sigma \). In the Aho-Corasick automaton construction, if this transition does not exist, we just jump to the next state on the supply path (Section 3.2.2). In the factor oracle construction, we create in addition a transition labeled by \( \sigma \) from each state on the supply path to the state where the original transition leads.
String matching of many patterns

Wu-Manber

SBOM

Lmin

(5 patterns)

(10 patterns)

(100 patterns)

Slides courtesy of Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
String matching of many patterns

Wu-Manber

SBOM

| Σ |
---|---|
8 |
4 |
2 |

Lmin

5              10              15              20            25             30            35           40            45

5              10              15              20            25             30            35           40            45

8 |
4 |
2 |

(5 patterns)

(10 patterns)

(1000 patterns)

(100 patterns)

Slides courtesy of Xavier Messeguer Peypoch (http://www.lsi.upc.es/~alggen)
5 strings

Flexible Pattern Matching in Strings  By Gonzalo Navarro, Mathieu Raffinot

string matching algorithms, showing for all of them the zone in which they are most efficient in practice. The text of 10 megabytes is randomly built, as are the patterns. The experiments were performed on a $w = 32$ bits Ultra Sparc 1 running SunOs 5.6. The sets contain 5, 10, 100, and 1000 strings of the same length, varying from 5 to 100 in steps of 5. We tested all the algorithms presented. The Wu-Manber algorithm used in these experiments is the implementation found in Agrep. Its performance may vary, depending on the hash functions and the sizes of the tables used.

![Graph showing string matching algorithms efficiency](image)

Fig. 3.24. Map of the most efficient algorithms when searching for 5 strings.

The maps are shown in Figures 3.24 to 3.27. The most efficient algorithms are just Wu-Manber, the advanced Aho-Corasick, and SBOM. As the set grows in size, SBOM becomes more and more attractive. The advanced Aho-Corasick also improves in comparison with the others for short strings, since it reads the text only once.
3.6 Other algorithms and references

Fig. 3.25. Map of the most efficient algorithms when searching for 10 strings.
Fig. 3.26. Map of the most efficient algorithms when searching for 100 strings.
Fig. 3.27. Map of the most efficient algorithms when searching for 1000 strings.

Complexity [Sri86] by using additional memory. These algorithms are, however, not efficient in practice.

On matching a set of strings on unbounded alphabets The problem of matching a set of strings of the same length $m$ on an unbounded alphabet has been investigated [Bre95]. The resulting algorithm runs in $O((\log(|P|)/m + 1) \times n)$ comparisons after an $O(|P| \times m \times \log |A|)$ preprocessing time, where $A$ is the alphabet on which the set $P$ is built.
Factor Oracle
Factor Oracle: safe shift
Factor Oracle:

Shift to match prefix of P2?
3.4 Factor based approach

Example using DNA. We search for the set of strings $P = \{\text{ATATATA}, \text{TATAT}, \text{ACGAT}\}$ in the text $\text{AGATACGATATAC}$. The factor oracle for the reverse set of $P_{\text{tmin}} = \{\text{TATAT, ATAT, ACG}\}$ is shown in Figure 3.23.

Fig. 3.23. Factor oracle for the reverse set of $P_{\text{tmin}} = \{\text{TATAT, ATAT, ACG}\}$. Double-circled states are terminal.
Construction of factor Oracle

The resulting factor automaton recognizes at least all factors of $P$ [AR99]. The construction algorithm is worst-case time $O(|P|)$. Its pseudo-code is given in Figure 3.20.

```plaintext
Build_Oracle_Multiple(P = \{p^1, p^2, \ldots, p^r\})
1. OR_trie ← Trie(P)
   \delta_{OR} is its transition function
2. Mark the states that correspond to an entire string $p^i$ as terminal
3. I ← root of OR_trie
4. S_{OR}(I) ← \emptyset
5. For Current in transversal order Do
   6. Parent ← parent in OR_trie of Current
   7. \sigma ← label of the transition from Parent to Current
   8. Down ← S_{OR}(Parent)
   9. While Down \neq \emptyset and \delta_{OR}(Down, \sigma) = \emptyset Do
      10. \delta_{OR}(Down, \sigma) ← Current
      11. Down ← S_{OR}(Down)
   12. End of while
   13. If Down \neq \emptyset Then
      14. S_{OR}(Current) ← \delta_{OR}(Down, \sigma)
   15. Else S_{OR}(Current) ← I
   16. End of if
   17. End of for
```

Fig. 3.20. Construction of the factor oracle for a set $P = \{p^1, p^2, \ldots, p^r\}$. The state $Current$ goes through the trie $OR_trie$ built on $P$ in transversal order. The state $Down$ goes down the supply links from the parent of $Current$ looking for an outgoing transition labeled with the same character as between $Current$ and its parent, creating it if it does not exist.
Factor oracle


- [http://www-igm.univ-mlv.fr/~allauzen/work/sofsem.ps](http://www-igm.univ-mlv.fr/~allauzen/work/sofsem.ps)
2 Factor oracle

2.1 Construction algorithm

\begin{center}
\textbf{Build\_Oracle}(p = p_1p_2 \ldots p_m)
1. For \( i \) from 0 to \( m \)
2. Create a new state \( i \)
3. For \( i \) from 0 to \( m - 1 \)
4. Build a new transition from \( i \) to \( i + 1 \) by \( p_{i+1} \)
5. For \( i \) from 0 to \( m - 1 \)
6. Let \( u \) be a minimal length word in state \( i \)
7. For all \( \sigma \in \Sigma, \sigma \neq p_{i+1} \)
8. If \( u\sigma \in \text{Fact}(p_i-|u|+1 \ldots p_m) \)
9. Build a new transition from \( i \) to \( i + \text{poccur}(u\sigma,p_i-|u|+1 \ldots p_m) \) by \( \sigma \)
\end{center}

\textbf{Figure 1.} High-level construction algorithm of the Oracle
Definition 1  The factor oracle of a word $p = p_1p_2 \ldots p_m$ is the automaton built by the algorithm Build_Oracle (figure 1) on the word $p$, where all the states are terminal. It is denoted by $\text{Oracle}(p)$.

The factor oracle of the word $p = abbaab$ is given as an example figure 2. On this example, it can be noticed that the word $aba$ is recognized whereas it is not a factor of $p$.

![Diagram](image)

Figure2. Factor oracle of $abbaab$. The word $aba$ is recognized whereas it is not a factor.

Note: all the transitions that reach state $i$ of $\text{Oracle}(p)$ are labeled by $p_i$.

Lemma 1  Let $u \in \Sigma^*$ be a minimal length word among the words recognized in state $i$ of $\text{Oracle}(p)$. Then, $u \in \text{Fact}(p)$ and $i = \text{pocc}ur(u,p)$.
So far

- Generalised KMP -> AhoCorasick
- Generalised Horspool -> CommentzWalter, WuManber
- BDM, BOM
  -> Set Backward Oracle Matching...

- Other generalisations?
Multiple Shift-AND

- $P = \{P_1, P_2, P_3, P_4\}$. Generalize Shift-AND

- **Bits** =

- **Start** =

- **Match** =
Text Algorithms (6EAP)

Full text indexing

Jaak Vilo

2010 fall
Problem

• Given P and S – find all exact or approximate occurrences of P in S

• You are allowed to preprocess S (and P, of course)

• Goal: to speed up the searches
E.g. Dictionary problem

• Does $P$ belong to a dictionary $D=\{d_1, ..., d_n\}$
  – Build a binary search tree of $D$
  – B-Tree of $D$
  – Hashing
  – Sorting + Binary search

• Build a keyword trie: search in $O(|P|)$
  – Assuming alphabet has up to a constant size $c$
  – See Aho-Corasick algorithm, Trie construction
Sorted array and binary search
Sorted array and binary search

O( |P| \log n )
Trie for $D=\{ \text{he}, \text{hers}, \text{his}, \text{she} \}$

$O(|P|)$
S != set of words

• S of length n

• How to index?

• Index from every position of a text

• Prefix of every possible suffix is important
Trie(babaab)

- babaab
- abaab
- baab
- aab
- ab
- b
Suffix tree

- **Definition:** A compact representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents.

- **Definition (suffix tree).** A suffix tree $T$ for a string $S$ (with $n = |S|$) is a rooted, labeled tree with a leaf for each non-empty suffix of $S$. Furthermore, a suffix tree satisfies the following properties:
  - Each internal node, other than the root, has at least two children;
  - Each edge leaving a particular node is labeled with a non-empty substring of $S$ of which the first symbol is unique among all first symbols of the edge labels of the edges leaving this particular node;
  - For any leaf in the tree, the concatenation of the edge labels on the path from the root to this leaf exactly spells out a non-empty suffix of $S$.

Literature on suffix trees

• **STXXL**: Standard Template Library for Extra Large Data Sets.
  

• **ANSI C implementation of a Suffix Tree**
  

These are old links – check for newer…
The suffix tree Tree(T) of T

- data structure **suffix tree**, Tree(T), is **compacted trie** that represents all the suffixes of string T
- linear size: $|\text{Tree}(T)| = O(|T|)$
- can be constructed in linear time $O(|T|)$
- has *myriad virtues* (A. Apostolico)
- is well-known: 366 000 Google hits

Suffix trie and suffix tree

abaab
baab
aab
ab
b

\textit{Trie}(abaab) \quad \textit{Tree}(abaab)

Suffix tree and suffix array techniques for pattern analysis in strings

Esko Ukkonen
Univ Helsinki
Erice School 30 Oct 2005

Algorithms for combinatorial string matching?

- deep beauty?  +-  
- shallow beauty?  +  
- applications?  ++  
- intensive algorithmic miniatures  
- sources of new problems: text processing, DNA, music,...

High-throughput genome-scale sequence analysis and mapping using compressed data structures

Veli Mäkinen

Department of Computer Science

University of Helsinki
ttttttttttttttgagacgggagtctcgctctgtcgcccaaggtgagtgcagtggcgggatctcgggtcactgcaagctccgctccgcgggttcacgccattctcctgcctcagccctcccaagtagctgggactacaggcgccgccactacgcctccggctaatttttttgtatttttagtagagagacgggggttttcaccgttttagccggtatgtgtctcgatctcctgacctcgtgtatccgccccgcctcggcctccaaagtgcgctgggattacaggctg

Analysis of a string of symbols

• $T = \texttt{hatti vatti}$ ‘text’
• $P = \texttt{att}$ ‘pattern’

• Find the occurrences of $P$ in $T$:
  $\texttt{hatti vatti}$

• Pattern synthesis: $\#(t) = 4 \quad \#(atti) = 2$
  $\#(t^{***}t) = 2$

Solution: backtracking with suffix tree

...ACACATTATCACAGGCATCGGCATTAGCGATCGAGTCG.....
Pattern finding & synthesis problems

- $T = t_1 t_2 \ldots t_n$, $P = p_1 p_2 \ldots p_n$, strings of symbols in finite alphabet

- **Indexing problem**: Preprocess $T$ (build an index structure) such that the occurrences of different patterns $P$ can be found fast
  - static text, any given pattern $P$

- **Pattern synthesis problem**: Learn from $T$ new patterns that occur surprisingly often

- **What is a pattern?** Exact substring, approximate substring, with generalized symbols, with gaps, ...
1. **Suffix tree**

2. Suffix array

3. Some applications

4. Finding motifs

The suffix tree Tree(T) of T

• data structure **suffix tree**, Tree(T), is **compacted trie** that represents all the suffixes of string T
• linear size: \(|\text{Tree}(T)| = O(|T|)\)
• can be constructed in linear time \(O(|T|)\)
• has **myriad virtues** (A. Apostolico)
• is well-known: 366 000 Google hits

Suffix trie and suffix tree

abaab
baab
aab
ab
b

Trie(abaab)

Tree(abaab)
Trie(T) can be large

- $|\text{Trie}(T)| = O(|T|^2)$
- bad example: $T = a^n b^n$
- Trie(T) can be seen as a DFA: language accepted = the suffixes of T
- minimize the DFA => directed cyclic word graph (’DAWG’)
Tree(T) is of linear size

- only the internal branching nodes and the leaves represented explicitly
- edges labeled by substrings of T
- \( v = \text{node}(\alpha) \) if the path from root to \( v \) spells \( \alpha \)
- one-to-one correspondence of leaves and suffixes
- \(|T|\) leaves, hence < \(|T|\) internal nodes
- \(|\text{Tree}(T)| = O(|T| + \text{size(edge labels)})\)
Tree(hattivatti)
Tree(hattivatti)

substring labels of edges represented as pairs of pointers
Tree(T) is full text index

P occurs in T \Leftrightarrow P is a prefix of some suffix of T
\Leftrightarrow Path for P exists in Tree(T)

All occurrences of P in time $O(|P| + \#occ)$
Find \textbf{att} from Tree(hattivatti)
Linear time construction of Tree(T)

Weiner (1973), 'algorithm of the year'

McCreight (1976)

'on-line' algorithm (Ukkonen 1992)
On-line construction of Trie(T)

• $T = t_1t_2 \ldots t_n$

• $P_i = t_1t_2 \ldots t_i$  i:th prefix of $T$

• on-line idea: update $Trie(P_i)$ to $Trie(P_{i+1})$

• => very simple construction
Trie(abaab)

Trie(a)       Trie(ab)       Trie(aba)

aba
ba
a
a
ε

chain of links connects the end points of current suffixes
Trie(abaab)

Trie(abaa)
Trie(abaab)

Add next symbol = b
Trie(abaab)

Add next symbol = b

From here on b-arc already exists
Trie(abaab)
What happens in $Trie(P_i) \Rightarrow Trie(P_{i+1})$?

**Before**

From here on the $a_i$-arc exists already => stop updating here

**After**

New suffix links

New nodes
What happens in $Trie(P_i) \Rightarrow Trie(P_{i+1})$?

- time: $O(\text{size of } Trie(T))$
- suffix links:
  $$\text{slink}(\text{node}(a\alpha)) = \text{node}(\alpha)$$
On-line procedure for suffix trie

1. Create $\text{Trie}(t_1)$: nodes root and $v$, an arc $\text{son}(\text{root}, t_i) = v$, and suffix links $\text{slink}(v) := \text{root}$ and $\text{slink}(\text{root}) := \text{root}$

2. for $i := 2$ to $n$ do begin

3. $v_{i-1} := \text{leaf of } \text{Trie}(t_1...t_{i-1}) \text{ for string } t_1...t_{i-1} \text{ (i.e., the deepest leaf)}$

4. $v := v_{i-1}; \ v' := 0$

5. while node $v$ has no outgoing arc for $t_i$ do begin

6. Create a new node $v''$ and an arc $\text{son}(v, t_i) = v''$

7. if $v' \neq 0$ then $\text{slink}(v) := v''$

8. $v := \text{slink}(v); \ v' := v''$ end

9. for the node $v''$ such that $v'' = \text{son}(v, t_i)$ do

   if $v'' = v'$ then $\text{slink}(v') := \text{root}$ else $\text{slink}(v') := v''$

end
Suffix trees on-line

• ‘compacted version’ of the on-line trie construction: simulate the construction on the linear size tree instead of the trie => time $O(|T|)$

• all trie nodes are conceptually still needed => *implicit* and *real* nodes
Implicit and real nodes

• Pair \((v, \alpha)\) is an *implicit node* in Tree(T) if \(v\) is a node of Tree and \(\alpha\) is a (proper) prefix of the label of some arc from \(v\). If \(\alpha\) is the empty string then \((v, \alpha)\) is a *'real' node* (= \(v\)).

• Let \(v = node(\alpha')\) in Tree(T). Then implicit node \((v, \alpha)\) represents \(node(\alpha'\alpha)\) of Trie(T)
Implicit node

\[ (v, \alpha) \]

\[ \alpha \]

\[ \alpha' \]
Suffix links and open arcs

\[
\text{slink}(v) \rightarrow v \\
\text{label } [i,\ast] \text{ instead of } [i,j] \text{ if } w \text{ is a leaf and } j \text{ is the scanned position of } T
\]
Big picture

suffix link path traversed: \textit{total work }O(n)

new arcs and nodes created: \textit{total work }O(\text{size(Tree(T))})
On-line procedure for suffix tree

Input: string $T = t_1t_2 \ldots t_n$

Output: $Tree(T)$

Notation: $son(v, \alpha) = w$ iff there is an arc from $v$ to $w$ with label $\alpha$

$son(v, \varepsilon) = v$

Function $Canonize(v, \alpha)$:

while $son(v, \alpha') \neq 0$ where $\alpha = \alpha' \alpha''$, $|\alpha'| > 0$ do

$v := son(v, \alpha')$; $\alpha := \alpha''$

return $(v, \alpha)$
Suffix-tree on-line: main procedure

Create Tree(t_1);  slink(root) := root
(v, α) := (root, ε)   /* (v, α) is the start node */
for i := 2 to n+1 do
  ν' := 0
  while there is no arc from ν with label prefix αt_i do
    if α ≠ ε then   /* divide the arc w = son(ν, αη) into two */
      son(ν, α) := ν'';  son(ν'', t_i) := ν''';  son(ν'', η) := w
    else
      son(ν, t_i) := ν'''';  ν''' := ν
    if ν' ≠ 0 then slink(ν') := ν''
    ν' := ν'';  ν := slink(ν);  (ν, α) := Canonize(ν, α)
  if ν' ≠ 0 then slink(ν') := ν
(v, α) := Canonize(ν, αt_i)   /* (v, α) = start node of the next round */
The actual time and space

• \(|\text{Tree}(T)|\) is about 20\(|T|\) in practice
• brute-force construction is \(O(|T|\log|T|)\) for random strings as the average depth of internal nodes is \(O(\log|T|)\)
• difference between linear and brute-force constructions not necessarily large (Giegerich & Kurtz)
• truncated suffix trees: \(k\) symbols long prefix of each suffix represented (Na et al. 2003)
• alphabet independent linear time (Farach 1997)
abc
abcabxabcd
Applications of Suffix Trees


• **APL1:** Exact String Matching Search for P from text S. Solution 1: build STree(S) - one achieves the same O(n+m) as Knuth-Morris-Pratt, for example!

• Search from the suffix tree is O(|P|)

• **APL2:** Exact set matching Search for a set of patterns P
Back to backtracking

ACA, 1 mismatch

Same idea can be used to many other forms of approximate search, like Smith-Waterman, position-restricted scoring matrices, regular expression search, etc.
Applications of Suffix Trees

- **APL3**: substring problem for a database of patterns
  Given a set of strings $S=S_1, \ldots, S_n$ --- a database Find all $S_i$ that have $P$ as a substring

- Generalized suffix tree contains all suffixes of all $S_i$

- Query in time $O(|P|)$, and can identify the LONGEST common prefix of $P$ in all $S_i$
Applications of Suffix Trees

• **APL4**: Longest common substring of two strings
• Find the longest common substring of S and T.
• Overall there are potentially $O(n^2)$ such substrings, if $n$ is the length of a shorter of S and T
• Donald Knuth once (1970) **conjectured** that linear-time algorithm is **impossible**.
• Solution: construct the STree(S+T) and find the node deepest in the tree that has suffixes from both S and T in subtree leaves.
• Ex: $S=superiorcalifornialives$ $T=sealiver$ have both a substring xxxxxxx.
Simple analysis task: LCSS

• Let $\text{LCSSA}(A,B)$ denote the longest common substring two sequences $A$ and $B$. E.g.:
  – $\text{LCSS}(\text{AGATCTATCT},\text{CGCCTCTATG})=\text{TCTAT}$.

• A good solution is to build suffix tree for the shorter sequence and make a *descending suffix walk* with the other sequence.
Suffix link
Descending suffix walk

Read B left-to-right, always going down in the tree when possible. If the next symbol of B does not match any edge label on current position, take suffix link, and try again. (Suffix link in the root to itself emits a symbol). The node $v$ encountered with largest string depth is the solution.
Another common tool: Generalized suffix tree

node info:
subtree size 47813871
sequence count 87

ACCTTA...ACCT#CACATT..CAT#TGTCGT...GTA#TCACCACC...C$
Generalized suffix tree application

node info:
subtree size 4398
blue sequences 12/15
red sequences 2/62

...ACC..#...ACC...#...ACC...ACC..ACC..ACC..#...ACC...#...
...#....#...#...#...ACC...#...#...#...#...#...#..#..ACC..ACC...#......#...

subtree size 4398
blue sequences 12/15
red sequences 2/62

...ACC..#...ACC...#...ACC...ACC..ACC..ACC..#...ACC...#...
...#....#...#...#...ACC...#...#...#...#...#...#..#..ACC..ACC...#......#...
Case study continued

motif?
	suffix tree of genome

5 blue
1 red

time? genome

regions with ChIP-seq matches
Properties of suffix tree

• Suffix tree has \( n \) leaves and at most \( n-1 \) internal nodes, where \( n \) is the total length of all sequences indexed.

• Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).

• Can be constructed in linear time.
Properties of suffix tree... in practice

• Huge overhead due to pointer structure:
  – Standard implementation of suffix tree for human genome requires over 200 GB memory!
  – A careful implementation (using $\log n$ -bit fields for each value and array layout for the tree) still requires over 40 GB.
  – Human genome itself takes less than 1 GB using 2-bits per bp.
Applications of Suffix Trees

- **APL4**: Longest common substring of two strings
- Find the longest common substring of S and T.
- Overall there are potentially $O(n^2)$ such substrings, if $n$ is the length of a shorter of S and T
- Donald Knuth once (1970) conjectured that linear-time algorithm is impossible.
- Solution: construct the STree(S+T) and find the node deepest in the tree that has suffixes from both S and T in subtree leaves.
- Ex: $S = \text{superiorcalifornialives}$ $T = \text{sealiver}$ have both a substring *alive*. 
Applications of Suffix Trees

- **APL5**: Recognizing DNA contamination Related to DNA sequencing, search for longest strings (longer than threshold) that are present in the DB of sequences of other genomes.

- **APL6**: Common substrings of more than two strings Generalization of APL4, can be done in linear (in total length of all strings) time
Applications of Suffix Trees

• **APL7**: Building a directed graph for exact matching: **Suffix graph** - directed acyclic word graph (DAWG), a **smallest finite state automaton** recognizing all suffixes of a string S. This automaton can recognize membership, but not tell which suffix was matched.

• Construction: merge isomorphic subtrees.

• Isomorphic in Suffix Tree when exists suffix link path, and subtrees have equal nr. of leaves.
Applications of Suffix Trees

• APL8: A reverse role for suffix trees, and major space reduction Index the pattern, not tree...
• Matching statistics.
• APL10: All-pairs suffix-prefix matching For all pairs $S_i, S_j$, find the longest matching suffix-prefix pair. Used in shortest common superstring generation (e.g. DNA sequence assembly), EST alignment etc.
Applications of Suffix Trees

• APL11: Finding all maximal repetitive structures in linear time
• APL12: Circular string linearization e.g. circular chemical molecules in the database, one wants to linearize them in a canonical way...
• APL13: Suffix arrays - more space reduction will touch that separately
Applications of Suffix Trees

• APL14: Suffix trees in genome-scale projects
• APL15: A Boyer-Moore approach to exact set matching
• APL16: Ziv-Lempel data compression
• APL17: Minimum length encoding of DNA
Applications of Suffix Trees

- Additional applications Mostly exercises...
- Extra feature: **CONSTANT time lowest common ancestor retrieval (LCA)**
  
  Andmestruktuur mis võimaldab leida konstantse ajaga alumist ühist vanemat (see vastab pikimale ühisele prefixile!) on võimalik koostada lineaarse ajaga.

- APL: Longest common extension: a bridge to inexact matching
- APL: **Finding all maximal palindromes in linear time**
  
  Palindrome reads from central position the same to left and right. E.g.: kirik, saippuakivikauppias.

- Build the suffix tree of S and inverted S (aabcbad => aabcbad#dabcbaa ) and using the LCA one can ask for any position pair (i, 2i-1), the longest common prefix in constant time.
- The whole problem can be solved in O(n).
Applications of Suffix Trees

• APL: Exact matching with wild cards
• APL: The k-mismatch problem
• Approximate palindromes and repeats
• Faster methods for tandem repeats
• A linear-time solution to the multiple common substring problem
• And many-many more ...
1. Suffix tree
2. Suffix array
3. Some applications
4. Finding motifs
• Sort all suffixes. Allows to perform binary search!
### Suffix array: example

<table>
<thead>
<tr>
<th></th>
<th>suffix array</th>
<th>lexicographic order of the suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hattivatti</td>
<td>ε</td>
</tr>
<tr>
<td>2</td>
<td>attivatti</td>
<td>atti</td>
</tr>
<tr>
<td>3</td>
<td>ttivatti</td>
<td>attivatti</td>
</tr>
<tr>
<td>4</td>
<td>tivatti</td>
<td>hattivatti</td>
</tr>
<tr>
<td>5</td>
<td>ivatti</td>
<td>i</td>
</tr>
<tr>
<td>6</td>
<td>vatti</td>
<td>ivatti</td>
</tr>
<tr>
<td>7</td>
<td>atti</td>
<td>i</td>
</tr>
<tr>
<td>8</td>
<td>tti</td>
<td>tivatti</td>
</tr>
<tr>
<td>9</td>
<td>ti</td>
<td>tti</td>
</tr>
<tr>
<td>10</td>
<td>i</td>
<td>ttivatti</td>
</tr>
<tr>
<td>11</td>
<td>ε</td>
<td>vatti</td>
</tr>
</tbody>
</table>

- **suffix array** = lexicographic order of the suffixes
Suffix array construction: sort!

- suffix array = lexicographic order of the suffixes
Suffix array

• **suffix array** $SA(T) = \text{an array giving the lexicographic order of the suffixes of } T$
• space requirement: $5|T|$
• practitioners like suffix arrays (simplicity, space efficiency)
• theoreticians like suffix trees (explicit structure)
Reducing space: suffix array

\[
\begin{align*}
C & \rightarrow 4 \\
A & \rightarrow 2 \\
T & \rightarrow 1 \\
A & \rightarrow 5 \\
C & \rightarrow 6 \\
T & \rightarrow 3
\end{align*}
\]

\[
\begin{align*}
\text{C A T A C T} & = [3,3] \\
\text{A T} & = [3,6] \\
\text{A C T} & = [2,6] \\
\text{T} & = [6,6] \\
\text{A C T} & = [4,6]
\end{align*}
\]
Suffix array

• Many algorithms on suffix tree can be simulated using \textit{suffix array}...
  – ... and couple of additional arrays...
  – ... forming so-called \textit{enhanced suffix array}...
  – ... leading to the similar space requirement as careful implementation of suffix tree

• Not a satisfactory solution to the space issue.
Pattern search from suffix array

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Length</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>hattivatti</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>attivatti</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>ttivatti</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>tivatti</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ivatti</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>vatti</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>atti</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>tti</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>ti</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>ε</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

binary search
What we learn today?

• We learn that it is possible to replace suffix trees with *compressed suffix trees* that take 8.8 GB for the human genome.

• We learn that *backtracking* can be done using *compressed suffix arrays* requiring only 2.1 GB for the human genome.

• We learn that *discovering* interesting motif seeds from the human genome takes 40 hours and requires 9.3 GB space.
Recent suffix array constructions

- Manber & Myers (1990): $O(|T| \log |T|)$
- Linear time via suffix tree
- January / June 2003: direct linear time construction of suffix array
  - Kim, Sim, Park, Park (CPM03)
  - Kärkkäinen & Sanders (ICALP03)
  - Ko & Aluru (CPM03)
Kärkkäinen-Sanders algorithm

1. Construct the suffix array of the suffixes starting at positions $i \mod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.

2. Construct the suffix array of the remaining suffixes using the result of the first step.

3. Merge the two suffix arrays into one.
Notation

• string $T = T[0,n) = t_0 t_1 \ldots t_{n-1}$
• suffix $S_i = T[i,0) = t_i t_{i+1} \ldots t_{n-1}$
• for $C \subset [0,n]$: $S_C = \{S_i \mid i \in C\}$

• *suffix array* $SA[0,n]$ of $T$ is a permutation of $[0,n]$ satisfying $S_{SA[0]} < S_{SA[1]} < \ldots < S_{SA[n]}$
Running example

0 1 2 3 4 5 6 7 8 9 10 11

• $T[0,n) = y a b b a d a b b a d o 0 0 ...$

• $SA = (12,1,6,4,9,3,8,2,7,5,10,11,0)$
Step 0: Construct a sample

- for $k = 0, 1, 2$
  $$B_k = \{i \in [0,n] \mid i \text{ mod } 3 = k\}$$
- $C = B_1 \cup B_2$ \textit{sample positions}
- $S_C$ \textit{sample suffixes}

- Example: $B_1 = \{1,4,7,10\}$, $B_2 = \{2,5,8,11\}$, $C = \{1,4,7,10,2,5,8,11\}$
Step 1: Sort sample suffixes

- for \( k = 1, 2 \), construct

\[
R_k = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{\text{maxBk}} t_{\text{maxBk+1}} t_{\text{maxBk+2}}]
\]

\[R = R_1 \ ^\land \ R_2 \ \text{concatenation of } R_1 \text{ and } R_2\]

Suffixes of \( R \) correspond to \( S_C \): suffix \([t_i t_{i+1} t_{i+2}]\ldots\) corresponds to \( S_i \); correspondence is order preserving.

Sort the suffixes of \( R \): radix sort the characters and rename with ranks to obtain \( R' \). If all characters different, their order directly gives the order of suffixes. Otherwise, sort the suffixes of \( R' \) using Kärkkäinen-Sanders. Note: \(|R'| = 2n/3\).
Step 1 (cont.)

• once the sample suffixes are sorted, assign a rank to each: \( \text{rank}(S_i) = \) the rank of \( S_i \) in \( S_C \); \( \text{rank}(S_{n+1}) = \text{rank}(S_{n+2}) = 0 \)

• Example:

\[
R = [\text{abb}][\text{ada}][\text{bba}][\text{do0}][\text{bba}][\text{dab}][\text{bad}][\text{o00}]
\]

\[
R' = (1,2,4,6,4,5,3,7)
\]

\[
SA_{R'} = (8,0,1,6,4,2,5,3,7)
\]

\[
\text{rank}(S_i) \quad 1 \text{ 4} - 2 \text{ 6} - 5 \text{ 3} - 7 \text{ 8} - 0 \text{ 0}
\]
Step 2: Sort nonsample suffixes

• for each non-sample $S_i \in S_{B0}$ (note that $\text{rank}(S_{i+1})$ is always defined for $i \in B0$):
  $S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$

• radix sort the pairs $(t_i, \text{rank}(S_{i+1}))$.

• Example: $S_{12} < S_6 < S_9 < S_3 < S_0$ because $(0,0) < (a,5) < (a,7) < (b,2) < (y,1)$
Step 3: Merge

- merge the two sorted sets of suffixes using a standard comparison-based merging:
- to compare $S_i \in S_C$ with $S_j \in S_{B0}$, distinguish two cases:
  - $i \in B1$: $S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$
  - $i \in B2$: $S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2}))$

- note that the ranks are defined in all cases!
- $S_1 < S_6$ as $(a,4) < (a,5)$ and $S_3 < S_8$ as $(b,a,6) < (b,a,7)$
Running time $O(n)$

- excluding the recursive call, everything can be done in linear time
- the recursion is on a string of length $2n/3$
- thus the time is given by recurrence
  \[ T(n) = T(2n/3) + O(n) \]
- hence $T(n) = O(n)$
Implementation

• about 50 lines of C++
• code available e.g. via Juha Kärkkäinen’s home page
LCP table

• Longest Common Prefix of successive elements of suffix array:
• $LCP[i] = \text{length of the longest common prefix of suffixes } S_{SA[i]} \text{ and } S_{SA[i+1]}$
• build inverse array $SA^{-1}$ from $SA$ in linear time
• then LCP table from $SA^{-1}$ in linear time (Kasai et al, CPM2001)
- **Oxford English Disctionary**  http://www.oed.com/
- **Example - Word of the Day, Fourth**  
  http://www.oed.com/cgi/display/wotd
- **PAT index** - by **Gaston Gonnet** (ta on samuti Maple tarkvara üks loojatest ning hiljem molekulaarbioloogia tarkvarapaketi väljatöötajaid)
- **PAT index** is essentially a suffix array. To save space, indexed only from first character of every word
- **XML-tagging** (or SGML, at that time!) also indexed
- To mark certain fields of XML, the bit vectors were used.
- **Main concern** - improve the speed of search on the CD - minimize random accesses.
- For slow medium even 15-20 accesses is too slow...
Suffix tree vs suffix array

- suffix tree $\iff$ suffix array + LCP table
1. Suffix tree
2. Suffix array
3. Some applications
4. Finding motifs
Substring motifs of string $T$

- string $T = t_1 ... t_n$ in alphabet $A$.
- Problem: what are the frequently occurring (ungapped) substrings of $T$? Longest substring that occurs at least $q$ times?
- Thm: Suffix tree $Tree(T)$ gives complete occurrence counts of all substring motifs of $T$ in $O(n)$ time (although $T$ may have $O(n^2)$ substrings!)

Counting the substring motifs

• internal nodes of Tree(T) $\leftrightarrow$ repeating substrings of $T$

• number of leaves of the subtree of a node for string $P = \text{number of occurrences of } P \text{ in } T$
Substring motifs of hattivatti

Counts for the $O(n)$ maximal motifs shown
Finding repeats in DNA

- human chromosome 3
- the first 48,999,930 bases
- 31 min cpu time (8 processors, 4 GB)

- Human genome: $3 \times 10^9$ bases
- Tree(HumanGenome) feasible
Longest repeat?

**Occurrences at:** 28395980, 28401554

**Length:** 2559

ttagggtacatgtgcacacagtgtgattcatagatgtgctgcaccattaactctgcatattagctgttagtgtatatatctccgaattgctgtatagcgcagcaatatttttcctgctatagtgcagtgatatctccgaatgctatccctcccccctccccccccccccccccaaacagttagttgctgaggtctgtgtgtttctctgtttctctggtgtttttggctgtttttttgctgccataaaacattcagtgcttttatagcagcatattataatctctttgggtatataccagtaatgggatggctgggtaaatggtatttctagttctagatccctgaggaat cacccacacagtccacacacggagtgtgcaacttagtttagctccacagctcaaatctcttcttctcattgttcaattcccacctatgagtgagaa catgcggtgtttggttttttgtccttgcgaaagtttgctgagaatgatggtttccagcttcatccatatccctacaccccttcaccagctcgttttttgctgacacttttaattaagatgacatgc gagaaggattctgtgttctcattctccgtccccactttgtgatgggtttttttttcttgtaaatttgttggagttcattgtagattctgggtattagccctttgtcagatgaggtggaaaaattttctcccattctgtaggtgcctgttcactctgatggtggtttcttctgctgtgcagagaagctcttttagtttaattagctcccatttgtcaattttgctgcatagctttgttgcagaaggagatttttagcatgaagagttcttgaattttgtcaaaggccttttctgcatcttttgagataatcatgtggtttctgtctttggttctgtttatatgctggagtaattttgattttcgtatgttgaaaccagccttgcatcccagggatgaagcccacttgatcatggtggataagctttttgatgtgctgctggattcggtttgccagtattttattgaggattttctgcatcgatgttcatcaaggatattggtctaaaattctctttttttgttgtgtctctgtcaggctttggtatcaggatgagctggcctcataaaatga
Ten occurrences?

```
tttttttttttttgagacggagtctcgctctgtcgcccaiggctggagtgcagtggcgggat
tctcggcctcactgcaagctccgcctcccgggttcacgccattctcctgcctcagcctcc
ccaaagtagctgggactacaggcgcgcctacgcgcctgggtaattttttgtatttttagt
agagacggggtttcaccgttttagccgggatggtctcgatctcctgacctcgtgatccg
cccgcctcggcctcccaagtgctgggattacaggcgt
```

**Length**: 277

**Occurrences at**: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125

In the reversed complement at: 17858493, 41463059, 42431718, 42580925
Using suffix trees: plagiarism

• find longest common substring of strings X and Y
• build Tree(X$Y) and find the deepest node which has a leaf pointing to X and another pointing to Y
Using suffix trees: approximate matching

- edit distance: insertions, deletions, changes

- STOCKHOLM vs TUKHOLMA
String distance/similarity functions

STOCKHOLM vs TUKHOLMA

STOCKHOLM_  _TU_  

KHOLMA

=> 2 deletions, 1 insertion, 1 change
Approximate string matching

- **A:** STOCKHOLM
  - / 
  - / 
  - TU K H O L M A

- **B:**

- minimum number of ’mutation’ steps:
  - a -> b  a -> e  e -> b ...

- \(d_{ID}(A,B) = 5\)  \(d_{Levenshtein}(A,B)= 4\)

- mutation costs => probabilistic modeling
- evaluation by dynamic programming => alignment
Dynamic programming

\[ d_{i,j} = \min \begin{cases} 
\text{if } a_i = b_j \text{ then } d_{i-1,j-1} & \text{else } \infty, \\
 d_{i-1,j} + 1, \\
 d_{i,j-1} + 1
\end{cases} \]

= distance between i-prefix of A and j-prefix of B
(substitution excluded)

mxn table \( d \)
\[ d_{i,j} = \min(\text{if } a_i = b_j \text{ then } d_{i-1,j-1} \text{ else } \infty, \]
\[ d_{i-1,j} + 1, \]
\[ d_{i,j-1} + 1) \]

<table>
<thead>
<tr>
<th>A \backslash B</th>
<th>s</th>
<th>t</th>
<th>o</th>
<th>c</th>
<th>k</th>
<th>h</th>
<th>o</th>
<th>l</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>h</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>o</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>m</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

optimal alignment by trace-back

\[ d_{ID}(A,B) \]
Search problem

• find approximate occurrences of pattern P in text T: substrings P’ of T such that d(P,P’) small
• dyn progr with small modification: O(mn)
• lots of (practical) improvement tricks
Index for approximate searching?

- dynamic programming: $P \times \text{Tree}(T)$ with backtracking