Text Algorithms

Jaak Vilo
2014 fall

Exact pattern matching

- $S = s_1 s_2 ... s_n$ (text) $|S| = n$ (length)
- $P = p_1 p_2 ... p_m$ (pattern) $|P| = m$
- $\Sigma$ - alphabet $|\Sigma| = c$
- Does $S$ contain $P$?
  - Does $S = S' P S''$ for some strings $S'$ and $S''$?
  - 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
- e-mails: [Christian.Charras, Thierry.Lecroq]@laposte.net

Brute force: BAB in text?

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

---

**Brute Force**

**Algorithm Naive**

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

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

```plaintext
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;
```

---

**Question:**

- Problems of this method? ☺
- Ideas to improve the search? ☺

---

**Brute force or NaiveSearch**

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 $s[i+j-1] \neq sub[j]$
5. jump to next iteration of outer loop
6. return $i$
7. return not found

---

**C code**

```c
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);
}
```

---

**Main problem of Naive**

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

- Make sure only a constant number of comparisons/operations is made for each position in $S$
  - Move (only) from left to right in $S$

  - How?
  - After a test of $S[i] \neq P[j]$ 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”

Automaton for ABCABD

- $P$ – longest suffix of any prefix that is also a prefix of a pattern
- Example: $ABCABD$

Automaton for ABCABD

KMP matching

Input: Text $S[1..n]$ and pattern $P[1..m]$
Output: First occurrence of $P$ in $S$ (if exists)

\[
i=1; \quad j=1;
\]

initfail($P$) // Prepare fail links

repeat

  if $j=0$ or $S[i] \neq P[j]$
    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

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
• O: How often can j decrease?
  — A: not more than nr of increases of i
• Amortised analysis: O(n), preprocess O(m)

Karp-Rabin
R.Karp and M. Rabin: Efficient randomized pattern-matching algorithms.

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

  • Goal: O(n).
  • f( h(T[i..i+m-1]) -> 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 $\Rightarrow$ Worst case $O(nm)$

Let’s use numbers

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

• $H(T[1]) = 571$
• $H(T[2]) = (571-5*100)*10 + 2 = 712$
• $H(T[3]) = (H(T[2]) - \text{ord}(T[1])*10m)*10 + T[3+m-1]$

hash

• $c$ – size of alphabet

• $HS_i = H( S[i..i+m-1] )$

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

• Modulo arithmetic – to fit value in a word!

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. $q = 33554432$ /* $q$ is a prime */
3. $cm = c^{m-1}$ mod $q$
4. $hp = 0$; $hs = 0$
5. for $i = 1$ .. $m$ do $hp = (hp*c + \text{ord}(p[i]) )$ mod $q$ // $H(P)$
6. for $i = 1$ .. $m$ do $hs = (hp*c + \text{ord}(s[i]) )$ mod $q$ // $H(S[1..m])$
7. if $hp == hs$ and $P == S[1..m]$ report match at position

More ways to ensure $O( n )$?
Shift-AND / Shift-OR
- Ricardo Baeza-Yates, Gaston H. Gonnet
  A new approach to text searching
  *Communications of the ACM* October 1992,
  Volume 35 Issue 10
  [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

Vectors for every char in $\Sigma$
- P=aste
  
  a s t e b c d .. z
  1 0 0 0 0 ...
  0 1 0 0 0 ...
  0 0 1 0 0 ...
  0 0 0 1 0 ...
Summary

<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>$O(n^2) \times (1+1/</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>$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)$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>$O(n/m) ?$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BM Horspool</td>
<td>$O(n/m)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factor search</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift-Or</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(m</td>
</tr>
</tbody>
</table>

A Fast String Searching Algorithm

Robert S. Boyer
Stanford Research Institute
J Strother Moore
Xerox Palo Alto Research Center

---

Find occurrences in text

```
P
S
```

• What have we learned if we test for a potential match from the end?

---

Find occurrences in text

```
P
S
```

---

Find occurrences in text

```
P
S
```

---

Find occurrences in text

```
P
S
```

---

Find occurrences in text

```
P
S
```

---

Find occurrences in text

```
P
S
```

• Have we missed anything?

---

Bad character heuristics

maximal shift on S[i]

```
P
S
S
```

\[
\text{delta}_j(S[i]) = |m| \quad \text{if pattern does not contain } S[i]
\]

\[
\text{pattern[j] = max} \quad \text{so that } P[j] = S[i]
\]

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

<table>
<thead>
<tr>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>μ</td>
</tr>
<tr>
<td>μ</td>
<td>μ</td>
</tr>
<tr>
<td>μ</td>
<td>μ</td>
</tr>
</tbody>
</table>

delta(S[i]) = 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

1. i=m
   while i <= 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( delta1[ S[i] ], delta2[ j ] )

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], Baesa-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] ]

• http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
• http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
• Animation: http://www.igm.univ-mlv.fr/~lecroq/string/
• http://www.inf.fh-flensburg.de/lang/algorithmen/pattern/bmen.htm
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?
  - Pattern:
  - Text:
  - It depends on 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.

- 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:

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

Algorithm Boyer-Moore-Horspool-Hume-Sunday (BMHHS)

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

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]]; // maximize d on P[m]
4. delta[P[m]] = 0; // ensure delta on match of last char is 0
5. for (i=m, a=a; i>0 && P[i]==S[i+m-a] ; i = i-a) do:
6. repeat // skip loop
7. d = delta[S[i]] ; i = i + d
8. until i=m+1
9. for (j=m-1 ; j>0 && P[j]==S[i-m+j] ; j = j-1) do:
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).

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.

```
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**

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 $x$. 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 $x$.

![Figure 2.13: Basic idea for shifting the window with the factor search approach. If we failed to recognize a factor of the pattern or $x$, then $\sigma u$ is no a factor of the pattern and the window can be safely shifted after $x$.](image)

**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

**Factor searches**

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

![Factor searches](image)

**Backward DAWG Matching BDM**

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

![Backward DAWG Matching BDM](image)

**BNDM — simulate using bitparallelism**

Bits — show where the factors have occurred so far

![BNDM — simulate using bitparallelism](image)

**BNDM matches an NDA**

NDA on the suffixes of ‘announce’

![BNDM matches an NDA](image)
Deterministic version of the same Backward Factor Oracle

String Matching of one pattern

<table>
<thead>
<tr>
<th>CTACTACTACGTCTATACTGATCGTAGC</th>
<th>TACTACGGTATGACTAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Prefix search</td>
<td>2. Suffix search</td>
</tr>
</tbody>
</table>

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

**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 like before.
- Set of patterns $P = \{ P_1, \ldots, P_k \}$
- Total length $|P| = m = \sum_{i=1}^{k} m_i$
- Problem: find all occurrences of any of the $P_i \in P$ 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 versions 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

**Trie**

- 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 $P=\{\text{he, she, his, hers}\}$

How to search for words like he, sheila, hi. Do these occur in the trie?

Aho-Corasick

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 $j$.

AC Automaton (vs KMP)

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. state = 0
2. for $i = 1..n$ do
3. while ($\text{goto[state}, S[i]] = \emptyset$) and ($\text{fail[state]} != \text{state}$) do
4. state = $\text{fail[state]}$
5. state = $\text{goto[state}, S[i]]$
6. if (output[state] not empty )
7. then report matches output[state] at position $i$
Algorithm Aho-Corasick preprocessing I (TRIE)

Input: \( P = \{ P_1, ..., P_k \} \)
Output: goto[] and partial output[]
Assume: output[] is empty when a state \( s \) is created;
goto[0,a] is not defined.

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

Preprocessing II for AC (FAIL)

queue = \( \emptyset \)
for \( a \in \Sigma \) do
   if goto[0,a] \( \neq \emptyset \) then
      enqueue(queue, goto[0,a])
   fail[ goto[0,a] ] = 0
end
while queue \( \neq \emptyset \)
\( r = \text{take}(queue) \)
for \( a \in \Sigma \) do
   if goto[ r, a ] \( \neq \emptyset \) then
      \( s = \text{goto}[ r, a ] \)
      enqueue(queue, s) // breadth first search
      state = fail[ s ]
      while goto[ state, a ] \( \neq \emptyset \) do state = fail[ state ]
      fail[ s ] = goto[ state, a ]
      output[ s ] = output[ s ] + output[ fail[ s ] ]
end

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

  - You can download here my algorithm StringMatchingFastOnTheAverage (PDF, ~17.2 MB) or here StringMatchingFastOnTheAverage (extended abstract) (PDF, ~3 MB)

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 grep 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.

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 grep 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, ATAAAT, ATGTG

1. Build the trie of the inverted patterns

2. \( l_{\text{min}} = 4 \)

3. Table of shifts

4. Start the search
What are the possible limitations for C-W?

- Many patterns, small alphabet – minimal skips
- What can be done differently?

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.

Wu-Manber

- Citeseer: 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 [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.

- Instead of looking at characters from the text one by one, we consider them in blocks of size B.
- 
- 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, AATAAT, ATGTG

<table>
<thead>
<tr>
<th>1 symbol</th>
<th>2 symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>AT</td>
<td>1</td>
</tr>
<tr>
<td>CA</td>
<td>1</td>
</tr>
<tr>
<td>CG</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Experimental length: log[2] 2*Lmin*r

Backward Oracle

- Set Backwards oracle SBDM, SBOM
- Pages 68-72

Wu-Manber algorithm

Search for ATGTATG, TATG, AATAAT, ATGTG

Experimental length: log[2] 2*Lmin*r
5 strings

10 strings

100 strings

1000 strings

Factor Oracle

Factor Oracle: safe shift
2 Factor oracle

2.1 Construction algorithm

1. For $i$ from 0 to $n$
2. Create a new state $i$
3. For $i$ from 0 to $n - 1$
4. Build a new transition from $i$ to $i + 1$ by $p_{i+1}$
5. For $i$ from 0 to $n - 1$
6. Let $a$ be a minimal length word in state $i$
7. For all $a = \rho 
8. If set $a \in T(a) =$ prefix($p_{i+1}, \ldots, p_n$) by $a$

Definition 1: The factor Oracle of a word $w = p_1 \ldots p_n$ is the automaton build by the algorithm Build Oracle (figure 2) on the word $p_n$ whose all the states are

Factor oracle

This factor Oracle of the word $p_n$, although is given in an example figure 3.

On this example, it can be noticed that the word $w$ is recognized whereas it is not a factor of $p$.

Fig-ure: High-level construction algorithm of the Oracle

Figure: Factor Oracle of all factors. The word $a$ is recognized whereas it is not a factor.

Note: All the transitions that reach state $i$ of Oracle are labeled by $p_i$.

Lemma 3: Let $a \in \Sigma^*$ be a minimal length word among the words recognized in state $i$ of Oracle. Then, $a \in T(a)$ and $i = \text{prefix}(a, p_i)$.
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
- \( \text{Bits} = \begin{bmatrix} P_4 & P_3 & P_2 & P_1 \end{bmatrix} \)
- \( \text{Start} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \)
- \( \text{Match} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \)

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

Text Algorithms (6EAP)

Full text indexing

Jaak Vilo
2010 fall

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 \( \mathcal{O}(|P|) \)
  - 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

Trie for \( D = \{ \text{he}, \text{hers}, \text{his}, \text{she} \} \)

\[ O( |P| \log n ) \]

\[ O( |P| ) \]

\( S \neq \) set of words

- \( S \) of length \( n \)
- How to index?
- Index from every position of a text
- Prefix of every possible suffix is important

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

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: |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 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

Analysis of a string of symbols

- $T = \texttt{hattivatti}$ ‘text’
- $P = \texttt{att}$ ‘pattern’

- Find the occurrences of $P$ in $T$: $\texttt{hattivatti}$
- Pattern synthesis: $\#(t) = 4$ $\#(atti) = 2$
  $\#(t****t) = 2$

Solution: backtracking with suffix tree


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: |Tree(T)| = O(|T|)
- can be constructed in linear time O(|T|)
- has myriad virtues (A. Apostolico)
- is well-known: 366 000 Google hits


Trie(T) can be large

- |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 = node(α) if the path from root to v spells α
- one-to-one correspondence of leaves and suffixes
- |T| leaves, hence < |T| internal nodes
- |Tree(T)| = O(|T| + size(edge labels))

Tree(hattivatti)

Suffix trie and suffix tree

Tree(hattivatti)

Tree(T) is full text index

P occurs in T at locations 8, 31, …

P occurs in T if P is a prefix of some suffix of T
Path for P exists in Tree(T)
All occurrences of P in time O(|P| + #occ)

Find att from Tree(hattivatti)

Linear time construction of Tree(T)

On-line construction of Trie(T)
• T = t_1t_2 ... t_n$
• P_i = t_1t_2 ... t_i: i-th prefix of T
• on-line idea: update Trie(P) to Trie(P_{i+1})
• => very simple construction

Trie(abaab)
What happens in $\text{Trie}(P) \Rightarrow \text{Trie}(P_{i+1})$?

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

After: New nodes

New suffix links

What happens in $\text{Trie}(P) \Rightarrow \text{Trie}(P_{i+1})$?

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

1. Create Trie(t): nodes root and v, an arc son(root, t) = v, and suffix links slink(v) := root and slink(root) := root
2. for i := 2 to n do begin
   3.    vi := leaf of Trie(t1…ti) for string t1…ti (i.e., the deepest leaf)
   4.    v := vi;  v' := 0
   5.    while node v has no outgoing arc for ti do begin
          6.      Create a new node v'' and an arc son(v, ti) = v''
          7.      if v' ≠ 0 then slink(v') := v''
          8.      v := slink(v);  v' := v'' end
   9.    for the node v'' such that v'' = son(v, ti) do
           if v'' = v' then slink(v'') := root else slink(v'') := v''

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, α) is an implicit node in Tree(T) if v is a node of Tree and α is a (proper) prefix of the label of some arc from v. If α is the empty string then (v, α) is a 'real' node (= v).
• Let v = node(α') in Tree(T). Then implicit node (v, α) represents node(α'α) of Trie(T)

Suffix links and open arcs

Big picture

suffix link path traversed: total work O(n)
new arcs and nodes created: total work O(size(Tree(T)))
**On-line procedure for suffix tree**

Input: string $T = t_1 t_2 \ldots t_n$
Output: $\text{Tree}(T)$

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

Function $\text{Canonize}(v,a)$:

while $\text{son}(v,a) \neq 0$ where $a = a' a''$, $|a'| > 0$

$v := \text{son}(v,a')$; $a := a''$

return $(v,a)$

---

**Suffix-tree on-line: main procedure**

Create $\text{Tree}(t_1)$;

$v := (\text{root}, \epsilon)$  /* $(v,a)$ is the start node */

for $i := 2$ to $n+1$ do

$v := 0$

while there is no arc from $v$ with label prefix $a_i$ do

if $a \neq \epsilon$ then

/* divide the arc $w = \text{son}(v,a)$ into two */

$\text{son}(v,a) := v'$; $\text{son}(v',a) := v''$; $\text{son}(v'',a) := w$

else

$\text{son}(v,a) := v''$; $v' := v$

if $v' ≠ 0$ then $\text{slink}(v') := v'$

$v := v'$; $(v,a) := \text{Canonize}(v,a)$

if $v' ≠ 0$ then $\text{slink}(v') := v$

$(v,a) := \text{Canonize}(v,a, a_i)$  /* $(v,a)$ = 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)

---

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 $S_i$

---

Simple analysis task: LCSS

• Let $LCSS(A,B)$ denote the longest common substring two sequences $A$ and $B$. E.g.:
   $LCSS(AGATCTATCGCCTCTATG)=TCTAT$.

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

---

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 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

---

Suffix link

Node $v$ encountered with largest string depth is the solution.
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
- **APL5**: Finding 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 \( \text{alive} \).
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 alamist ühist vanemat (see vastab pikimale ühisele prefixile) on võimalik koostada lineaarse ajaga.
- **API**: Longest common extension: a bridge to inexact matching
- **API**: 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 [aabcbd => aabcbad#abcdab] 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

**Suffixes - sorted**

- Sort all suffixes. Allows to perform binary search!

**Suffix array: example**

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

- suffix array = lexicographic order of the suffixes

**Suffix array construction: sort!**

**Reducing space: 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)
Suffix array

• Many algorithms on suffix tree can be simulated using suffix array...
  — ... and couple of additional arrays...
  — ... forming so-called 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>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>hattivatti</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>attivatti</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>tti</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>tti</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>vatti</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>hattivatti</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>vatti</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>atti</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>tti</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>tti</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

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 $C \setminus$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

• \( T[0, n) = y a b b a d a b b a d o 0 0 \ldots \)

• \( SA = (12, 1, 6, 4, 9, 3, 8, 2, 7, 5, 10, 11, 0) \)

Step 0: Construct a sample

• for \( k = 0,1,2 \)
  \[ B_k = \{ i \in [0, n) \mid i \text{ mod } 3 = k \} \]

• \( C = B_1 \cup B_2 \) sample positions

• \( S_C \) 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
  \[ R_k = [t_{s_{k1}}, t_{s_{k2}}], t_{s_{k3}}, t_{s_{k4}}, t_{s_{k5}}) \ldots \]
  \[ [t_{\text{max}}k1, t_{\text{max}}k2}] \]

• \( R = R_1 \cup R_2 \) concatenation of \( R_1 \) and \( R_2 \)

Suffixes of \( R \) correspond to \( S_C \); suffix \([t_{s_{k1}}, t_{s_{k2}}] \ldots \) 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:
  \[ \text{rank}(S_i) = \text{the rank of } S_i \text{ in } S_C; \text{rank}(S_{n+1}) = \text{rank}(S_{n+2}) = 0 \]

• Example:
  \( R = [abb][ada][bba][do0][bba][dab][bad][o00] \)
  \( R' = (1, 2, 4, 6, 4, 5, 3, 7) \)
  \( SA'_C = (8, 0, 1, 6, 4, 2, 5, 3, 7) \)
  \( \text{rank}(S_i) = 1 4 - 2 6 - 5 3 - 7 8 - 0 0 \)

Step 2: Sort nonsample suffixes

• for each non-sample \( S_i \in S_{B0} \) (note that \( \text{rank}(S_i) \) 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: \( S_1 < S_2 < S_9 < S_3 < S_0 \) 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. \( i \in B1: S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})) \)

  ii. \( i \in B2: S_i \leq S_j \iff (t_i, t_j, \text{rank}(S_{i+1})) \leq (t_j, t_j, \text{rank}(S_{j+1})) \)

  • note that the ranks are defined in all cases!

  • \( S_1 < S_2 \) as \((a,4) < (a,5)\) and \( S_3 < S_4 \) 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:
  \[ \text{LCP}[i] = \text{length of the longest common prefix of suffixes } S_{SA[i]} \text{ and } S_{SA[i+1]} \]
- build inverse array $SA^{-1}$ from $SA$ in linear time
- then LCP table from $SA^{-1}$ in linear time (Kasai et al, CPM2001)

Suffix tree vs suffix array
- suffix tree $\Leftrightarrow$ suffix array + LCP table

1. Suffix tree
2. Suffix array
3. Some applications
4. Finding motifs
Substring motifs of string T

- String $T = t_1 \ldots t_n$ 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 $Tree(T)$ ↔ repeating substrings of $T$
- Number of leaves of the subtree of a node for string $P$ = number of occurrences of $P$ 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: 3x$10^9$ bases
- $Tree(HumanGenome)$ feasible

Longest repeat?

| Occurrences at: 28395980, 28401554 | Length: 2559 |

| Ten occurrences? |

| ttttttttttttagacgggtacctcgcctctggtgtgaaggatatgcagtcctgctgattcatgtgctgactgtgagggatcctcgc | Length: 277 |

| Occurrences at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125, 42580925 |
| 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

B: TUKHOLMA

• 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_{ij} = \min(\begin{cases} 0 & \text{if } a_i = b_j \\ d_{i-1,j-1} & \text{else if } a_i = b_j \\ 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

A

B

0 1 2 3 4 5 6 7 8

s t o c k h o l m

1 1 2 2 3 4 5 6 7 8

t u k h o l m a

2 3 3 4 5 6 7 8 9

k h o l m

3 4 4 5 6 7 8 9

h o l m

4 5 5 6 7 8 9

o l m

5 6 6 7 8 9

l m

6 7 7 8 9

m

7 8 8 9

A

B

0 1 2 3 4 5 6 7 8 9

s t o c k h o l m

1 1 2 2 3 4 5 6 7 8

t u k h o l m a

2 3 3 4 5 6 7 8 9

k h o l m

3 4 4 5 6 7 8 9

h o l m

4 5 5 6 7 8 9

o l m

5 6 6 7 8 9

l m

6 7 7 8 9

m

7 8 8 9

a

8 9 8 7 8 9

b

9 8 7 6 5

optimal alignment by trace-back
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