Algorithmics (6EAP)

Full text indexing

Jaak Vilo
2019 fall
Key take home messages

• Suffix trie, **Suffix tree**
  – Application examples

• **Suffix array**

• Burrows Wheeler Transform (BWT)
  – Compact Suffix tree representation: BWT + succinct
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 \( \mathbf{O}(|P|) \)
  – 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 $D=\{ \text{he, hers, his, she} \}$

$O(|P|)$
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)
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

Suffix tree and suffix array techniques for pattern analysis in strings

Esko Ukkonen

Univ Helsinki

Erice School 30 Oct 2005


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

Veli Mäkinen

Department of Computer Science

University of Helsinki
ttttttttttttttgagacggagtctcgctctg
tcgcccaaggctggagtgcagtggcggg
atctcgggctcactgcaagctccgcctcc
cgggttcacgccattttctctgcctcagcc
tcccaagtaggtgtgggactacagggcgc
cgccactacgcctgggttaattttgtatttttttagtagagacggggtttcaccgttttagc
cgggatggtctcgatctcctgacctcgtg
atccgcccgcctcggcctcccaaagtgc
tgggattacagggcgt

1. **Suffix tree**

2. Suffix array

3. Some applications

4. Finding motifs

The suffix tree $\text{Tree}(T)$ of $T$

- data structure **suffix tree**, $\text{Tree}(T)$, 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)

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 at locations 8, 31, …

P occurs in T $\iff$ P is a prefix of some suffix of T
$\iff$ 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' algorithm (Ukkonen 1992)

- Weiner (1973), 'algorithm of the year'
- McCreight (1976)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti	tti
ti
i
On-line construction of Trie(T)

• $T = t_1 t_2 \ldots t_n$
• $P_i = t_1 t_2 \ldots t_i \quad \text{i:th prefix of } T$
• on-line idea: update Trie($P_i$) to Trie($P_{i+1}$)
• => very simple construction
**Trie(abaab)**

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

```
  a
  b
```

```
  a  b
  a  b
```

```
  a  b
  a  a
```

chain of links connects the end points of current suffixes

aba
baa
a
εa
ε
Trie(abaab)

Trie(abaab)

Trie(aba)

Trie(aba)
Trie(abaab)

Add next symbol = b

Trie(abaab)

Add next symbol = b
Trie(abaab)

Trie(abaa)

Add next symbol = b

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

**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 $Trie(t_1)$: nodes root and $v$, an arc $son(root, t_1) = v$, and suffix links $slink(v) := root$ and $slink(root) := root$

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

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

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

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

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

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

8. $v := slink(v); \; v^{\prime} := v^{\prime\prime}$ end

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

   if $v^{\prime\prime} = v^{\prime}$ then $slink(v^{\prime}) := root$ else $slink(v^{\prime}) := v^{\prime\prime}$
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 \(v = \text{node}(\alpha \, \acute{\,})\) in Tree(T). Then implicit node \((v, \alpha)\) represents \(\text{node}(\alpha \, \acute{\alpha})\) of Trie(T)
Implicit node

\[ \alpha \]

\[ (v, \alpha) \]
Suffix links and open arcs

\[ \text{slink}(v) \]

\( \alpha \)  

\( \text{root} \)

\( a\alpha \)

\[ [i,*] \] instead of \([i,j]\) if \( w \) is a leaf and \( j \) is the scanned position of \( T \)
Big picture

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

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

Input: string $T = t_1 t_2 \ldots t_n$

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 $Canonize(v, \alpha)$:

\[
\text{while } son(v, \alpha') \neq 0 \text{ where } \alpha = \alpha' \cdot \alpha'' \text{ and } |\alpha'| > 0 \text{ do}
\]

\[
v := son(v, \alpha'); \alpha := \alpha''
\]

\[
\text{return } (v, \alpha)
\]
Suffix-tree on-line: main procedure

Create $Tree(t_i)$;  $slink(root) := root$

$(v, \alpha) := (root, \epsilon)$  /* $(v, \alpha)$ is the start node */

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

$v^\prime := 0$

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

if $\alpha \neq \epsilon$ then  /* divide the arc $w = son(v, \alpha\eta)$ into two */

$son(v, \alpha) := v^\prime$;  $son(v^\prime, t_i) := v^\prime \prime$;  $son(v^\prime, \eta) := w$

else

$son(v, t_i) := v^\prime \prime$;  $v^\prime := v$

if $v^\prime \neq 0$ then $slink(v^\prime) := v^\prime$\n
$v^\prime := v^\prime$;  $v := slink(v)$;  $(v, \alpha) := Canonize(v, \alpha)$

if $v^\prime \neq 0$ then $slink(v^\prime) := v$

$(v, \alpha) := Canonize(v, at_i)$  /* $(v, \alpha)$ = start node of the next round */
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 *alive*. 
Simple analysis task: LCSS

• Let \( \text{LCSS}(A,B) \) 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.
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
Another common tool: Generalized suffix tree

ACCTTA...ACCT#CACATT..CAT#TGTCGT...GTA#TCACCACC...C$

node info:
subtree size 47813871
sequence count 87
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...#......#...

A

C

c

subtree size 4398
blue sequences 12/15
red sequences 2/62
Case study continued

- Suffix tree of genome
- Motif?
- 5 blue, 1 red

Genome regions with ChIP-seq matches
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 ...
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.
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 = lexicographic order of the suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hattivatti</td>
</tr>
<tr>
<td>2</td>
<td>attivatti</td>
</tr>
<tr>
<td>3</td>
<td>ttivatti</td>
</tr>
<tr>
<td>4</td>
<td>tivatti</td>
</tr>
<tr>
<td>5</td>
<td>ivatti</td>
</tr>
<tr>
<td>6</td>
<td>vatti</td>
</tr>
<tr>
<td>7</td>
<td>atti</td>
</tr>
<tr>
<td>8</td>
<td>tti</td>
</tr>
<tr>
<td>9</td>
<td>ti</td>
</tr>
<tr>
<td>10</td>
<td>i</td>
</tr>
<tr>
<td>11</td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suffix array construction: sort!

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

- 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

C A T A C T
1 2 3 4 5 6
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>Suffix Array</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>hattivatti</td>
<td>ε</td>
<td>11</td>
</tr>
<tr>
<td>attivatti</td>
<td>atti</td>
<td>7</td>
</tr>
<tr>
<td>ttivatti</td>
<td>attivatti</td>
<td>2</td>
</tr>
<tr>
<td>tivatti</td>
<td>hattivatti</td>
<td>1</td>
</tr>
<tr>
<td>ivatti</td>
<td>i</td>
<td>10</td>
</tr>
<tr>
<td>vatti</td>
<td>ivatti</td>
<td>5</td>
</tr>
<tr>
<td>atti</td>
<td>ti</td>
<td>9</td>
</tr>
<tr>
<td>tti</td>
<td>tivatti</td>
<td>4</td>
</tr>
<tr>
<td>ti</td>
<td>tti</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>ttivatti</td>
<td>3</td>
</tr>
<tr>
<td>ε</td>
<td>vatti</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 $T = T[0,n) = t_0t_1 \ldots t_{n-1}$

• suffix $S_i = T[i,0) = t_it_{i+1} \ldots t_{n-1}$

• 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

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

  \[ R = R_1 \^ R_2 \] concatenation of $R_1$ 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:

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

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

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

\[
\text{rank}(S_i) = 14 - 26 - 53 - 78 - 00
\]
Step 2: Sort nonsample suffixes

• for each non-sample $S_i \in S_{B0}$ (note that 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 $S_j \in S_{B_0}$, distinguish two cases:

  • $i \in B_1$: $S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$
  • $i \in B_2$: $S_i \leq S_j \iff (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2}))$

• 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] = length of the longest common prefix of suffixes $S_{SA[i]}$ 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 Dictionary  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 $\Leftrightarrow$ 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
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

ttagggtacatgtgcacaacgcaggttgggtactatagttatacagctggcatgatgttcgtgcacacattaactctatcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaactcatttagctagcgttaggtatatctccgatagtctggctcacattgatggtgctgcacccattaact
Ten occurrences?

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
Dynamic programming

d_{i,j} = \min(\text{if } a_i = b_j \text{ then } d_{i-1,j-1} \text{ else } \infty, \n d_{i-1,j} + 1, \n d_{i,j-1} + 1) \n
= \text{distance between } i\text{-prefix of } A \text{ and } j\text{-prefix of } B \n \text{(substitution excluded)}

mxn table d

\begin{array}{|c|c|}
\hline
A & d & B \\
\hline
a_i & d_{i-1,j-1} & b_j \\
\hline
d_{i,j-1} & d_{i-1,j} & +1 \\
\hline
d_{i,j} & d_{i,j} & +1 \\
\hline
d_{m,n} & & \\
\hline
\end{array}
\[d_{i,j} = \min(\begin{cases} 
\text{if } a_i = b_j \text{ then } d_{i-1,j-1} & \text{else } \infty, \\
 d_{i-1,j} + 1, \\
 d_{i,j-1} + 1 \end{cases})\]

<table>
<thead>
<tr>
<th>A\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
Burrows-Wheeler Transformation

- BWT for text compression and indexing

<table>
<thead>
<tr>
<th>Input</th>
<th>SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>TEXYDST.E.IXIXIXXSSMPPS.B..E.S.EUSFXDIIIOIIIT</td>
</tr>
</tbody>
</table>
Burrows-Wheeler

• The method described in the original paper is really a composite of three different algorithms:
  – the block sorting main engine (a lossless, very slightly expansive preprocessor),
  – the move-to-front coder (a byte-for-byte simple, fast, locally adaptive noncompressive coder) and
  – a simple statistical compressor (first order Huffman is mentioned as a candidate) eventually doing the compression.
• Of these three methods only the first two are discussed here as they are what constitutes the heart of the algorithm. These two algorithms combined form a completely reversible (lossless) transformation that - with typical input - skews the first order symbol distributions to make the data more compressible with simple methods. Intuitively speaking, the method transforms slack in the higher order probabilities of the input block (thus making them more even, whitening them) to slack in the lower order statistics. This effect is what is seen in the histogram of the resulting symbol data.
• Please, read the article by Mark Nelson:
<table>
<thead>
<tr>
<th>String</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>O</td>
<td>B</td>
<td>B</td>
<td>S</td>
</tr>
<tr>
<td>S1</td>
<td>R</td>
<td>D</td>
<td>O</td>
<td>B</td>
<td>B</td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td>S2</td>
<td>D</td>
<td>O</td>
<td>B</td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>R</td>
</tr>
<tr>
<td>S3</td>
<td>O</td>
<td>B</td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>R</td>
<td>D</td>
</tr>
<tr>
<td>S4</td>
<td>B</td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>O</td>
</tr>
<tr>
<td>S5</td>
<td>B</td>
<td>S</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>O</td>
<td>B</td>
</tr>
<tr>
<td>S6</td>
<td>S</td>
<td>D</td>
<td>R</td>
<td>D</td>
<td>O</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>
BWT
5 * => 5 2 4 0 1 6 3
0 DRDOBBS
1
2
3
CODE:

t: hat acts like this:

t: hat buffer to the constructor

t: hat corrupted the heap, or wo

W: hat goes up must come down

t: hat happens, it isn't likely

w: hat if you want to dynamically

t: hat indicates an error.

t: hat it removes arguments from

t: hat looks like this:

t: hat looks something like this

t: hat looks something like this

t: hat once I detect the mangled
Example

• Decode: errktreteoe.e

• Hint: Is the last character, alphabetically first...
Burrows-Wheeler Transform

text -> bwt(text)

naive
- create table of rotations
- sort table
- extract last row

LF - mapping
- shows where the row will be after a rotation

Occ - naive
- count the occurrences
  - time: O(n)
  - space: O(f)

suffix array/tree
- create suffix array/tree
- length of suffixes
- find letter left of suffix

bwt(text) -> text

inverse
- follow LF-mapping on bwt(text)

FM-index
- uses bwt(text) for full text search

searching

FM-index is a memory efficient full text index
- it uses bwt(text) as an index
- it uses LF mapping property to search in the index
- it requires memory linear to the size of the text
- if compression is used then its memory use can be sublinear to the size of the text

count
- because top points to the first suffix and $b$ points to the last $t$

FM-index is a memory efficient full text index
- it uses bwt(text) as an index
- it uses LF mapping property to search in the index
- it requires memory linear to the size of the text
- if compression is used then its memory use can be sublinear to the size of the text
text -> bwt(text)

naïve

create table of rotations

sort table

extract last row

text

aacaacg

cacaacg

cacagac

cagacag

cgacga

cacaacg

aacaacg

cacagac

cagacag

cgacga

cacaacg

cacaacg

bwt(text)

gc

c

suffix array/tree

create suffix array/tree

length of suffixes

find letter left of suffix

text

aacaacg

cacaacg

cacagac

cagacag

cgacga

cacaacg

1
5
7
4
6
3
2

bwt(text)

gc

c

a
a
a
a
LF - mapping

shows where the row will be after a rotation

'C[c]' is the first occurrence of letter 'c' in the F column

LF can be calculated with:

\[ LF(i, c) = C[c] + \text{Occ}(c, i) \]

for transformation 'c = bwt[i]'
FM-index uses BWT(text) for full text search

FM-index is a memory efficient full text index

it uses BWT(text) as an index

it uses LF mapping property to search in the index

it requires memory linear to the size of the text

if compression is used then its memory use can be sublinear to the size of the text

proof of top update:
for i = len(q) - 1 .. 0
    c = q[i]
    top = LF(top, c)
    bot = LF(bot, c)
if (top >= bot)
    return none
return [top..bot]

count:
top, bot = search(q)
return bot - top

location:
top, bot = search(q)
for i in range(top, bot)
c = 0
while BWT[i] != q
    i = LF[i]
c++
query is at pos c

can be sped up with checkpointing similarly to OCC