Content: Full-text indexing

- 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={d₁,...,dₙ}
  - 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

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

$O( |P| )$

S is not a set of words, but a single long string

- 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


1. **Suffix tree**
2. **Suffix array**
3. **Some applications**
4. **Finding motifs**

Partly based on:

- 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

---

**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: 578 000 Google hits

---

**Suffix trie and suffix tree**

- abaab
- baab
- aab
- ab
- b

\text{Trie}(abaab)

---

**Suffix trie and suffix tree**

- abaab
- baab
- aab
- ab
- b

\text{Trie}(abaab)\text{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)

- substring labels of edges represented as pairs of pointers

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 \(O(|P| + \text{#occ})\)
Find att from Tree(hattivatti)

Linear time construction of Tree(T)

On-line construction of Trie(T)

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

Trie(abaab)

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

Before
From here on the $a_i$-arc exists already $\Rightarrow$ stop updating here

After
New nodes
New suffix links

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. $v_i :=$ leaf of $Trie(t_1, \ldots, t_i)$ for string $t_1, \ldots, t_i$ (i.e., the deepest leaf)
4. $v := v_i$; $v' := 0$
5. while node $v$ has no outgoing arc for $t$ do begin
6. Create a new node $v'$ and an arc $son(v, t) = v'$
7. if $v' \neq 0$ then $slink(v) := v'$
8. $v := slink(v)$; $v' := v''$ end
9. for the node $v''$ such that $v'' = son(v, t)$ 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 $\Rightarrow$ time $O(|T|)$
• all trie nodes are conceptually still needed $\Rightarrow$ implicit and real nodes
Implicit and real nodes

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

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

Suffix links and open arcs

Suffix-tree on-line: main procedure

Create \(\text{Tree}(T)\): \(\text{slink}(\text{root}) = \text{root}\)
\( (v, \alpha) := (\text{root}, \varepsilon) \) (\(v, \alpha\) is the start node *Y*

for \(i := 2\) to \(n+1\) do
\(v := 0\) while there is no arc from \(v\) with label prefix \(\alpha\):
    do
        if \(\alpha \neq \varepsilon\) then
            /* divide the arc \(w = \text{son}(v, \alpha)\) into two *Y*/
            \(\text{son}(v, \alpha) := v\)
            \(\text{son}(v, \eta) := w\)
        else
            \(\text{son}(v, \eta) := \varepsilon''\)
            \(\text{v} := \varepsilon''\)
        if \(v \neq 0\) then \(\text{slink}(v) := v\)
    \( (v, \alpha) := \text{Canonize}(v, \alpha) \) /* (v, \alpha) = start node of the next round *Y*/
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

Applications of Suffix Trees

- **APL3**: substring problem for a database of patterns
  - Given a set of strings S=S1, ..., Sn --- a database Find all Si that have P as a substring
- Generalized suffix tree contains all suffixes of all Si
- Query in time O(|P|), and can identify the LONGEST common prefix of P in all Si

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= superlcalifornialives 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.:
  - \( \text{LCSS}(\text{AGATCGATCT},\text{CGCTCTATCG})=\text{TCTAT} \)
- A good solution is to build suffix tree for the shorter sequence and make a **descending suffix walk** with the other sequence.
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

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.
- 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 võimalik koostada lineaarse ajaga:
- APL: Finding all maximal palindromes in linear time.
- 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 $S$ and using the LCA one can ask for any position pair $(i, 2i-1)$, the longest common prefix in constant time.

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, test 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

Suffix array: example
- suffix array = lexicographic order of the suffixes

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

Algorithmics (6EAP)
Full text indexing
Suffix arrays
Jaak Vilo
2020 fall

Suffixes - sorted
- Sort all suffixes. Allows to perform binary search!
Suffix array

- suffix array \( SA(T) \) = 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

Pattern search from suffix array

Historically first?


- PAT index for Oxford English Dictionary project on CD-s

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.

Running example

- $T[0,n) = \, y\, a\, b\, b\, a\, d\, a\, b\, b\, a\, d\, a\, b\, a\, d\, o\, 0\, 0\, ...$
- $SA = (12,1,6,4,9,3,8,2,7,5,10,11,0)$

Notation

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

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

Step 0: Construct a sample

- for $k = 0,1,2$
  - $B_k = \{i \in [0,n] | i \mod 3 = k\}$
- $C = B_1 \cup B_2$ 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
  \[ R_k = [t_{k+1}t_{k+2} \ldots t_{k+maxB_k}] \]
  \[ R = R_1 \cup R_2 \] (concatenation of R1 and R2)

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

• Example:
  \( R = [abb][ada][bba][do0][bba][dab][bad][o00] \)
  \( R' = (1,2,4,6,5,3,7) \)
  \( SA_{12} = (8,0,1,6,4,2,5,3,7) \)
  \( \text{rank}(S_0) = 1 \text{, rank}(S_1) = 4 \text{, rank}(S_2) = 2 \text{, rank}(S_3) = 5 \text{, rank}(S_4) = 6 \text{, rank}(S_5) = 3 \text{, rank}(S_6) = 7 \text{, rank}(S_7) = 0 \)

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: \( S_{12} < S_6 < S_0 < S_1 < S_3 \) 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:
    • \( i \in B1: S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})) \)
    • \( i \in B2: S_i \leq S_j \iff (t_i, \text{rank}(S_{i+2})) \leq (t_j, \text{rank}(S_{j+2})) \)
  • 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)

Suffix tree vs suffix array

- suffix tree \(\iff\) suffix array + LCP table

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 \(Tree(T)\) \(\iff\) repeating substrings of \(T\)
- number of leaves of the subtree of a node for string \(P = \text{number of occurrences of } P\) 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: $3 \times 10^9$ bases
- Tree(HumanGenome) feasible

Longest repeat?

Occurrences at: 28,395,980, 28,401,554
Length: 2,559

Ten occurrences?

Length: 277

Using suffix trees: plagiarism

- find longest common substring of strings X and Y
- build Tree(XSY) 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_ TUKHOLMA

=> 2 deletions, 1 insertion, 1 change

Dynamic programming

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

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

mxn table

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

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 x Tree(T) with backtracking

Algorithmics (6EAP)

Burrows-Wheeler transformation

Jaak Vilo
2020 fall
Burrows-Wheeler Transformation

- BWT for text compression and indexing

Input: SIX.MIXED.FIXIES.SIFT.SIXTY.FIXIE.DUST.BOXES
Output: TXYDST.E.IIXIXXSMPS.B...S.EUSFXDOIIIT

BWT for text compression and indexing

• 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:
• Data Compression with the Burrows-Wheeler TransformMark Nelson, Dr. Dobb's Journal

---

BWT

String 0
| D | R | D | O | B | B | S |

String 6
| S | D | O | D | B | B | S |

... other strings...

Column L, start from 6

```
O
B
R
S
D
D
B
```

How so we decode the message? What is the next letter?
Next letter is the F column, where characters are in alphabetic order

First B in F is first B in L.
Letters follow stable sort.

CODE:
- t: hat acts like this:

"t: hat buffer to the constructor"
- t: hat corrupted the heap, or vo
- W: hat goes up must come down:
- t: hat happens, it isn’t likely
- w: hat if you want to dynamically
- t: hat removes arguments from
- t: hat looks 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...