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 $D = \{d_1, \ldots, d_n\}$
  - Build a binary search tree of $D$
  - B-Tree of $D$
  - Hashing
  - Sorting + Binary search
  - Build a keyword trie: search in $O(|P|)$
    - Assuming alphabet has up to a constant size $c$
    - See Aho-Corasick algorithm, Trie construction

Sorted array and binary search

- Sorted array and binary search

- $O(|P| \log n)$
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**

- 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

Literature on suffix trees


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

- $|\text{Trie}(T)| = O(|T|^2)$
- 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(T) is full text index

- $P$ occurs in $T$ at locations 8, 31, ...
- $P$ is a prefix of some suffix of $T$
- Path for $P$ exists in Tree(T)
- All occurrences of $P$ in time $O(|P| + \#occ)$
On-line construction of Trie(T)

- \( T = t_1t_2 \ldots t_n \$
- \( P_i = t_1t_2 \ldots t_i \) : \( i \)th prefix of \( T \)
- **on-line idea**: update \( \text{Trie}(P_i) \) to \( \text{Trie}(P_{i+1}) \)
- \( \Rightarrow \) very simple construction

Linear time construction of Tree(T)

Weiner (1973), ‘algorithm of the year’

McCreight (1976)

'on-line' algorithm (Ukkonen 1992)
Trie(abaab)

Add next symbol = b
From here on b-arc already exists

What happens in Trie(P_i) => Trie(P_{i+1})?

• time: O(size of Trie(T))
• suffix links:
  \text{slink(node(αα))} = \text{node(α)}

On-line procedure for suffix trie
1. Create Trie(t_1): nodes root and v, an arc \text{son(root, t_1)} = v, and suffix links \text{slink(v)} := \text{root} and \text{slink(root)} := \text{root}
2. for i := 2 to n do begin
3.   v := 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' := 0
5.   while node v has no outgoing arc for t do begin
6.     Create a new node v'' and an arc \text{son(v,t)} = v''
7.     if v' ≠ 0 then \text{slink(v)} := v''
8.     v := \text{slink(v)}; v' := v''
9.   end
10.  for the node v'' such that v'' = \text{son(v,t)} do
11.     if v'' = v' then \text{slink(v')} := \text{root else \text{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 => \text{implicit and real nodes}
Implicit and real nodes

- Pair \((v, \alpha)\) is an *implicit node* in Tree(T) if \(v\) is a node of Tree and \(\alpha\) is a (proper) prefix of the label of some arc from \(v\). If \(\alpha\) is the empty string then \((v, \alpha)\) is a *real* node (= \(v\)).

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

Suffix links and open arcs

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

On-line procedure for suffix tree

Input: string \(T = t_1t_2 \ldots t_n\)
Output: Tree(T)

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

Function \(Canonize(v, \alpha)\):

- while \(son(v, \alpha) \neq 0 \) where \(\alpha = a' \cdot \alpha'' ; |a'| > 0\) do
  
  \(v := son(v, a') ; a := a''\)

- return \((v, \alpha)\)

Suffix-tree on-line: main procedure

Create Tree(T); slink(root) = root

\((v, a) \equiv (\text{root}, \epsilon) \quad \forall \ (v, a)\) is the start node \(\forall\)

for \(i := 2 \text{ to } n+1\) do

\(v' := 0\)

while there is no arc from \(v\) with label prefix \(a\) do

if \(a \neq \epsilon\) then

  - divide the arc \(w = son(v, \alpha)\) into two \(\forall\)

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

else

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

if \(v'' \neq 0\) then slink(v') := v'''

\(v' := v''\); \(v := slink(v')\); \(v, a := Canonize(v, a)\)

\((v, a) := Canonize(v, \alpha)\) \(\forall\) \((v, a)\) = start node of the next round \(\forall\)
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 \( S=S_1, \ldots, S_n \) --- 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 \( LCSS(A,B) \) denote the longest common substring two sequences \( A \) and \( B \). E.g.:
  \[
  LCSS(AGATCTACT,CGCCTCTATG)=TCTAT.
  \]
- 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

Generalized suffix tree application

Case study continued
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...
- **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

- **APL1**: Finding all maximal palindromes in linear time
- Palindrome reads from central position the same to left and right. E.g.: kirk, saippua, vokaupias
- Build the suffix tree of $S$ and inverted $S$ (aaabcbad => 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 $O(n)$.
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.

Suffixes - sorted

- Sort all suffixes. Allows to perform binary search!

Suffix array: example

- suffix array = lexicographic order of the suffixes

Suffix array construction: sort!

- suffix array = lexicographic order of the suffixes
Suffix array

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

Reducing space: suffix array

Pattern search from suffix array

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** \( S_i = T[i,0) = t_{i}t_{i+1} \ldots t_{n-1} \)
- 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 \( S_{SA[0]} < S_{SA[1]} < \ldots < S_{SA[n]} \)

Running example

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

Step 0: Construct a sample

- for \( k = 0,1,2 \)
  - \( B_k = \{ i \in \{0,n\} | i \mod 3 = k \} \)
  - \( C = B_1 \cup B_2 \)
  - \( S_C \) 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
  - \( R_k = [t_{i_0}t_{i_1+2}] [t_{i_3}t_{i_4+2}] \ldots [t_{i_{maxB_k}}t_{i_{maxB_k}+2}] \)
  - \( R = R_1 \cup R_2 \)
  - Concatenation of \( R_1 \) and \( R_2 \)

Suffixes of \( R \) correspond to \( S_C \): suffix \( [t_{i_0}t_{i_1+2}] \ldots \) corresponds to \( S_{i_1} \); 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: \( |R'| = \frac{2n}{3} \).

Step 1 (cont.)

- once the sample suffixes are sorted, assign a rank to each: \( \text{rank}(S_i) = \) the rank of \( S_i \) in \( S_C \): \( \text{rank}(S_{max}) = \text{rank}(S_{max+2}) = 0 \)
- Example:
  - \( R = [abb][ada][bb][doo][bb][dab][bad][o00] \)
  - \( R' = (1,2,4,6,4,5,3,7) \)
  - \( SA_{R'} = (8,0,1,6,4,2,5,3,7) \)
  - \( \text{rank}(S_0) = 1 \) - 2 - 6 - 5 - 3 - 7 - 8 - 0 - 0
Step 2: Sort nonsample suffixes

- for each non-sample \( S \in S_{B0} \) (note that rank\((S_{i+1})\) is always defined for \( i \in B0 \)):
  \( S \leq S_i \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})) \)
- radix sort the pairs \((t_i, \text{rank}(S_{i+1}))\).
- Example: \( S_{12} < S_6 < S_9 < S_3 < S_0 \) because \((0,0) < (a,5) < (a,7) < (b,2) < (y,1)\)

Step 3: Merge

- merge the two sorted sets of suffixes using a standard comparison-based merging:
- to compare \( S_i \in S_C \) with \( S_j \in S_{B0} \), distinguish two cases:
  - \( i \in B1: S_i \leq S_j \iff (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1})) \)
  - \( i \in B2: S_i \leq S_j \iff (t_i, t_i+1, \text{rank}(S_{i+2})) \leq (t_j, t_j+1, \text{rank}(S_{j+2})) \)
- note that the ranks are defined in all cases!
- \( S_1 < S_6 \) as \((a,4) < (a,5)\) and \( S_3 < S_8 \) as \((b,a,6) < (b,a,7)\)

Running time \( O(n) \)

- excluding the recursive call, everything can be done in linear time
- the recursion is on a string of length \( 2n/3 \)
- thus the time is given by recurrence
  \[ T(n) = T\left(\frac{2n}{3}\right) + O(n) \]
- 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 \( SA^+ \) from \( SA \) in linear time
- then \( \text{LCP} \) table from \( SA^+ \) in linear time (Kasai et al, CPM2001)
Suffix tree vs suffix array

• suffix tree ⇔ suffix array + LCP table

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

Substring motifs of string T

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

Counting the substring motifs

• internal nodes of $Tree(T)$ ⇔ repeating substrings of $T$
• number of leaves of the subtree of a node for string $P = \text{number of occurrences of } P \text{ in } T$

Substring motifs of hattivatti

Counts for the $O(n)$ maximal motifs shown

Finding repeats in DNA

• human chromosome 3
• the first 48 999 930 bases
• 31 min cpu time (8 processors, 4 GB)
• Human genome: 3x10^9 bases
• Tree(HumanGenome) feasible
Longest repeat?

Occurrences at: 28395980, 28401554
Length: 2559

Ten occurrences?

Length: 277

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__KHLMA

=> 2 deletions, 1 insertion, 1 change

Dynamic programming

d_{ij} = \min(\begin{cases} 0 & \text{if } a_i = b_j \\ d_{i-1,j-1} + 1 & \text{if } a_i \neq b_j \text{ and } a_i \neq b_j \\ d_{i-1,j-1} & \text{if } a_i \neq b_j \text{ and } a_i = b_j \\ \infty & \text{if } a_i = b_j \text{ and } a_i = b_j \end{cases})

= distance between i-prefix of A and j-prefix of B
(substitution excluded)
\[
d_{ij} = \min \begin{cases} 
  a = b & \text{then } d_{i-1,j-1} + 1, \\
  a \neq b & \infty
\end{cases}
\]

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

Burrows-Wheeler Transformation

- BWT for text compression and indexing

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-by-byte adaptive noncompressive coder)
  - a simple statistical compressor (first order Huffman is mentioned as a candidate eventually along the way)
- 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 back 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 Transform - Mark Nelson, Dr. Dobb's Journal
CODE:
- t: hat acts like this: c1 = t1 = c1
- t: hat buffer to the constructor
- t: hat corrupted the heap, or we
- 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, c1 = t1
- t: hat removes arguments from
- t: hat looks like this: c1 = t1 = c1
- t: hat looks something like this
- t: hat once I detect the mangled

Example
- Decode: errkteoteoe.e
- Hint: . Is the last character, alphabetically first...

Burrows-Wheeler Transform
- text -> bwt(text)
