Generalized Suffix Trees

We begin with a useful generalization of suffix trees for indexing multiple strings \( \{ S_1, \ldots, S_k \} \) (instead of a single one)

A generalized suffix tree can be used to represent all suffixes of a set of strings

How to build generalized suffix trees?

A Better Approach

The previous technique is OK, if all the strings are given at the start
What if all strings are not given in advance?
Could we add them one by one?
Let’s sketch an incremental strategy for constructing generalized suffix trees

Simple Construction of Generalized Suffix Trees

A simple strategy:
Apply ordinary construction to a concatenation of the strings:

1. Build a suffix tree \( T \) for \( C = S_1S_2 \ldots S_k \), where \( S_i \) is a unique end-marker for each string \( S_i, i = 1, \ldots, k \)

2. Each leaf gets numbered by a start position of \( C \)

These can easily be converted to pairs \((i, j)\), where \( i \) identifies the string, and \( j \) is the start position in string \( S_i \)

Total time is \( O(|C|) \)

“Synthetic suffixes” that span boundaries of original strings can be eliminated easily. (See Gusfield, Sec. 6.4)

Incremental Construction of Generalized Suffix Trees

Incremental strategy: Add the strings one-by-one, using Ukkonen’s algorithm, as follows:

Build a suffix tree \( T \) for \( S_1 \), keeping the suffix links

Traverse the longest path from the root of \( T \) that matches a prefix \( S_2[1 \ldots i] \) of string \( S_2 \)

Now \( T \) encodes all suffixes of \( S_1 \), and all suffixes of \( S_2[1 \ldots i] \), as if the first \( i \) phases had been computed for \( S_2 \)

Continue from Phase \( i + 1 \) for \( S_2 \) (with up-ward steps and suffix link and path traversals)

After entering \( S_2 \), continue similarly with \( S_3 \)\ldots \( S_k \)

Total time is linear wrt the total length of the strings

Example of Construction (2)

Tree after adding suffixes of \( S_2[1 \ldots 2] = ba \):

 Consider adding \( S_1 = baaba \) to the (generalized) suffix tree \( T \) of \( S_1 = xaba \):

The longest prefix of \( S_2 \) already in \( T \) is \( S_2[1 \ldots 1] = b \)

Continue Ukkonen’s algorithm from Phase 2, to enter suffixes of \( S_2[1 \ldots 2] = ba \) in the tree

Example of Generalized Suffix-Tree Construction

Consider adding \( S_1 = baaba \) to the (generalized) suffix tree \( T \) of \( S_1 = xaba \):

The longest prefix of \( S_2 \) already in \( T \) is \( S_2[1 \ldots 1] = b \)
Example of Construction (3)

Generalized suffix tree for \( S_1 = \text{zobixa} \) and \( S_2 = \text{babaxba} \) (after completing all phases for \( S_2 \)):

Two Implementation Details

1. Edge-labels of a generalized suffix tree come from more than one string
   \( \rightsquigarrow \) An edge-label \( S[p \ldots q] \) needs to be represented in the compressed form by a triple \((i, p, q)\)

2. Some suffixes may occur in multiple strings
   (if they have a common end marker)
   \( \rightsquigarrow \) Store in each leaf \( v \) all pairs \((i, j)\) such that \( L(v) \) is a suffix of string \( S_i \) starting at position \( j \)

Applications to String Matching

Linear-time exact matching methods (Z, KMP, BM) solve the problem using

- \( O(n) \) time for preprocessing \((n = |P|)\), and
- \( O(m) \) time for searching \((m = |T|)\)

Suffix trees give the same asymptotic total time, with

- \( O(m) \) time for preprocessing, and
- \( O(n + z) \) time for searching,
  where \( z \leq m \) is the number of occurrences of \( P \)

Exact Matching on Static Texts

Suffix trees are not practical for single searches (because of larger constant factors)

If target \( T \) is static, a suffix tree of \( T \) is a winner:
After \( O(|T|) \) preprocessing, the search for occurrences of any \( P \) takes time \( O(|P| + z) \) only
(as opposed to \( O(|P| + |T|) \); Often \( |P|, z \ll |T| \))

This may be especially useful with a generalized suffix tree built of a database of sequences

Problems:

- Required space, even if \( O(m) \), can be large in practice
- What if we want to find inexact occurrences?

Application to Exact Set Matching

Aho-Corasick solves exact set matching in time
\( O(n + m + z) \), where \( n \) is the total length of the given patterns \( P_1, \ldots, P_L \), and \( z \) is the total number of pattern occurrences in text \( T[1 \ldots m] \)

Same asymptotic time results from applying suffix trees:

1. Build the suffix tree \( T \) for \( T[1 \ldots m] \), in time \( O(m) \)
2. For each pattern \( P_i \) find its occurrences in \( T \);
   If \( z_i \in \{0, \ldots, m\} \) is the number of them, this takes time \( O(|P| + z_i) \)
   \( \rightsquigarrow \) Total time of all searches is
   \[ O(\sum_{i=1}^{L} (|P| + z_i)) = O(n + z) \]

Longest Common Substrings

A classic problem: How to find the longest substring common to two given strings?

Example: The longest common substring of “piliosana” and “namiloma” is “ilo”

This longest common substring problem is easy to solve in linear time using suffix trees

- Pre-dating suffix trees, in 1970, D. Knuth conjectured this to be impossible!

For each node \( v \) of a suffix tree, define the string-depth of \( v \) to be the length of the path-label \( L(v) \) of \( v \)

Finding Longest Common Substrings

Maximal substrings common to strings \( S_1 \) and \( S_2 \) can be found as follows:

1. Build a generalized suffix tree for \( S_1 \) and \( S_2 \)
2. Mark each internal node \( v \) by 1 (resp. 2) if the subtree below \( v \) contains a leaf for a suffix of \( S_1 \) (resp. \( S_2 \))
3. Traverse the tree to find nodes marked by both 1 and 2, and choose any \( u \) of them with a maximal string-depth
   \( \rightsquigarrow L(u) \) is a maximal common substring

Steps 2 and 3 can be implemented with simple tree-traversals in linear time (Exercise)

Finding Longest Common Substrings (2)

The above method gives us the result:

**Theorem 7.4.1** A longest common substring of two strings can be found in linear time

\( \square \)

**NB** The above method builds a generalized suffix tree of size \( \Theta(|S_1| + |S_2|) \)

The problem can be solved also in less space, with a suffix tree built for the shorter of the strings only
(as an application of matching statistics, to be discussed later)
**Appl: DNA Contamination**

Various laboratory processes cause unwanted DNA to become inserted into the string of interest, for example:
- fragments of a vector used to place the desired DNA in a host organism
- DNA of the host organism

This DNA contamination should be recognized to avoid unnecessary work and erroneous conclusions.

**DNA Contamination as a String Problem**

DNA sequences of many possible contaminants (cloning vectors, PCR primers, genome of the host organism) are known.

This suggests the following DNA contamination problem:
- Given a string $S_1$ (a newly sequenced DNA) and a set of strings $S$ (sources of possible contamination), find all substrings of $S$ that occur in $S_1$, and are longer than some given length $l$.

**Solving the DNA Contamination Problem**

The problem can be solved as an extension of the previous one:
1. Build a generalized suffix tree $T$ for $S \cup \{S_1\}$
2. Mark all internal nodes of $T$ whose subtree contains leaves for both $S_1$ and some string of $S$
3. Traverse $T$; For any marked node $v$ with string-depth $\geq l$ report $L(v)$ as a suspicious substring

**Remarks**

This problem concerns substrings that have exact occurrences in many of the given strings. Locating “similar” substrings, allowing some errors or differences, will be considered later.

Finding exactly matching common substrings occurs as a subproblem of many heuristics developed for multiple alignment, which we’ll also discuss later.

**Example of $l(i)$ values**

The $l(i)$ values and corresponding substrings for $S = \{s_1, s_2, s_3, s_4, s_5\}$:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$l(i)$</th>
<th>Substring</th>
<th>Obs: $l(i) \geq l(i+1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>sand</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>sand</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>and</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>an</td>
<td></td>
</tr>
</tbody>
</table>

**Frequent Common Substrings**

Instead of looking for the longest common substrings, we could be interested in substrings that are frequent, that is, occur in a large number of distinct strings.

**Formalizing the Problem**

Let $S = \{S_1, \ldots, S_k\}$ be a set of strings, of total length $n$.

Which of the common substrings are both frequent and long?

Define $l(i)$, for each $i = 2, \ldots, k$, as the length of a maximal substring that is common to at least $i$ strings of $S$.

**Finding Frequent Common Substrings**

The problem can be solved in $O(n)$ time using suffix trees.

We discuss a simpler $O(kn)$ time solution as a generalization of the longest common substring method:
1. Compute a generalized suffix tree $T$ for strings $S_1, s_2, \ldots, S_k$ (with end markers $s_i \neq S_j$ when $i \neq j$)
2. Compute how many distinct string identifiers occur below any internal node of $T$ and a pointer to a corresponding substring.
### Computing the $C(v)$ numbers

The number of leaves in any subtree of $T$ is easy to compute in linear time, but it’s more difficult to count occurrences of distinct string identifiers.

For this, compute for each internal node $v$ a bit-vector $b[i]$ where $b[i] = 1$ iff identifier of string $i$ occurs in a leaf below $v$.

The vector for $v$ is computed by $B$-linking the vectors of the child nodes $v_1, v_2, \ldots, v_n$ of $v$. That takes time $O(kl)$.

$C(v)$ is now the number of 1-bits in the vector of $v$.

They can counted in time $O(kl)$ per node.

Since the total number of (child) nodes is $O(n)$, the total time is $O(kn)$.

### Finding Frequent Substrings (2)

Having computed the $C(v)$ values continue as follows:

Compute the $l(i)$ values in terms of longest substrings that occur in exactly $i$ strings; For this, define

$$V(i) = \max \{0\} \cup \{\{w\} | w \text{ occurs in exactly } i \text{ strings}\}$$

The $V(i)$ values can be computed as follows:

1. Initialize $V(i)$ to zero for each $i = 2, \ldots, k$.
2. Let $C(v) = i$ set $V(i) = \max \{V(i), d\}$.

### Computing Matching Statistics

Matching statistics $ms(i)$ can be computed for each posn $i$ of $T$ as follows:

First build a suffix tree $T$ for $P$ yes. Keep the suffix links.

The longest substring that occurs at posn $i$ of $T$ is the longest prefix of $T[1 \ldots n]$ that matches a root-to-leaf path in $T$. Traversing them explicitly for each start posn $i$ gives the correct $ms(i)$ values, but may require $O(nm)$ time.

Avoid explicit path traversals by simulating them with suffix links and skip/count tricks.

### Finding Frequent Substrings (3)

Now obviously $l(i) = \max \{V(j) | i \leq j \leq k\}$

$\therefore$ The $l(i)$ values can be computed from $V(i)$ as follows:

$$l(k) = V(k)$$

for $i := k - 1$ downto 2 do $l(i) = \max \{l(i+1), V(i)\}$.

**Example:** $V(i)$ and $l(i)$ for $[aba, abx, bold, bolde]$:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$V(i)$</th>
<th>$l(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3 (foo)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2 (o de)</td>
<td>3</td>
</tr>
</tbody>
</table>

(Computation of pointers to substrings is left as an exercise)

### Computing Matching Statistics (3)

In case (a), compute $ms(j + 1)$ by traversing path $T[j + 1 \ldots i]$ starting from the root. (Apply skip/count to $T[j + 1 \ldots i]$, and then compare $T[b + 1 \ldots j]$ against the path as long it matches)

In case (b), let $L(v) = \alpha_x$, and $T[j \ldots i] = x\beta$

Now $s(e)$ is labeled by $\alpha$. Since path $\alpha \beta$ is in $T$, path $\beta$ can be followed down-wards from $s(e)$ applying the skip/count trick.

From the end of path $\alpha \beta$, continue by matching further characters of $T$ explicitly along downward edges of $T$.

Complexity is seen to be linear.
Suffix Arrays

**Definition of Suffix Arrays**

A suffix array for string \( T[1 \ldots m] \) is an array \( Pos[1 \ldots m] \) which gives start positions of all nonempty suffixes of \( T \) in increasing lexicographic order (sanakirjäljäristys).

- That is, alphabetic order, and longer strings preceded by their proper prefixes:
  - auto
  - autokauppa
  - automaatti
  - automaatinen

**Example:** A suffix array for string “mississippi” (\( m = 11 \))

<table>
<thead>
<tr>
<th>( i )</th>
<th>( Pos[i] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

**Complexity of Matching with a Suffix Array**

The outlined method gives the following result:

**Theorem 7.14.2** All occurrences of \( P \) in \( T \) can be found using a suffix array in time \( O(n \log m) \)

**Proof.** At most \( n \) characters are examined at each of the \( O(\log m) \) string comparisons.

The above bound is pessimistic; Normally strings differ after just a few characters, and expected time should be \( O(n + \log m) \) only.

In many applications \( \log m = O(n) \), and thus the expected search time would be only \( O(n) \).
**Improving the Search Time**

Exact matching using a suffix array can be implemented with $O(n + \log m)$ worst-case search time (by computing $O(n)$ "longest common prefix values"; Gusfield, Sec. 7.14.4 and 7.14.5)

Following is a simpler method, which achieves the same search time in practice:

Let $L$ and $R$ be the boundaries of the "current search interval" of the binary search.

Initially $L = 1$ and $R = m$

**Constructing a Suffix Array**

SA can be built in linear time by first building a suffix tree

- reasonable, if enough space for the construction, but need to store the result in less space

- a direct $O(n)$ construction recently found by Kärkkäinen & Sanders (ICALP'03)

Suffix array $Pos$ for $T$ can be obtained by a "lexical" depth-first traversal of the suffix tree $T$ for $T$

Let $e_1 = (v, u)$ and $e_2 = (v, w)$ be two edges in $T$, with edge-labels $a_1 \ldots a_k$ and $b_1 \ldots b_l$.

Edge $e_1$ is lexically less than edge $e_2$ iff $a_1 < b_1$

For this we treat $\$ as the first character of the alphabet

**Example of a Lexical Traversal**

**Lexical Depth-First Traversal**

A path in $T$ along the lexically least (last) edges leads to the start position of the lexically first (last) suffix of $T$

Generally, suffixes of $T$ can be listed in increasing lexical order by a depth-first traversal of $T$ in increasing lexical order of the edges. (See the next slide)

**Theorem 7.14.1** Suffix array $Pos$ for a string $T[1 \ldots n]$ can be constructed in $O(n \log n)$ time

**Proof.** First construct the suffix tree $T$ for $T$

Then perform a lexical depth-first traversal of $T$, filling entries of array $Pos$ with the numbers of the leaves in the order they are encountered

**Genome-Scale Projects**

Gusfield mentions three genome-scale projects (of mid-90’s) that apply suffix trees (with variations):

- *Arabidopsis thaliana* project at the Michigan State Univ. and the Univ. of Minnesota
- *Analysis of Saccharomyces cerevisiae* (brewer’s yeast) at the Max-Planck Institute
- *Borrelia burgdorferi* project at the Brookhaven National Laboratory

We’ll bring out some points of these:

**Arabidopsis thaliana**

- a classic model organism in molecular biology of plants
- Genome about 100 million base-pairs
- Goal: an EST map
  - roughly, listing of relative locations for known fragments of genes
- Generalized suffix trees applied to
  - check DNA contamination by known vector sequences
  - find duplicates and similarities with previously sequenced fragments
  - find biologically significant patterns (represented as regular expressions)

**Borrelia Burgdorferi**

- the bacterium causing Borreliosis, ~ 1 MB long
- Suffix trees and arrays used in fragment assembly
  - (deducing the underlying long sequence from its fragments)
  - to speed up suffix-prefix overlap detection
- for this, 4,612 fragments of total length ~ 2 MB handled in quarter of an hour

Implementation details are crucial in large scale suffix-tree applications (e.g., ~ 20 million chars for *Arabidopsis* sequences)

Efficient external-storage implementations needed