Biosequence Algorithms, Spring 2005
Lecture 8: Applications of Suffix Trees

Pekka Kilpeläinen
University of Kuopio
Department of Computer Science
Applications of Suffix Trees

Suffix trees have numerous applications, often providing linear-time solutions to challenging string problems.

A selection of them:

- Exact matching
- Common substrings, with applications
- Matching statistics
- Suffix arrays
- Genome-scale projects

Many others are given in the textbook.
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?
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 \( C = S_1\$_1S_2\$_2 \ldots S_k\$_k \), where \( \$_i \) is a unique end-marker for each string \( S_i \), \( i = 1, \ldots, k \)

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)
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.
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 Generalized Suffix-Tree Construction

Consider adding $S_2 = babxba$ to the (generalized) suffix tree $T$ of $S_1 = xabxa$:

The longest prefix of $S_2$ already in $T$ is $S_2[1\ldots1] = b$

$\leadsto$ Continue Ukkonen’s algorithm from Phase 2, to enter suffixes of $S_2[1\ldots2] = ba$ in the tree
Example of Construction (2)

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

![Diagram of suffix tree]

- $bxa$: (1,1)
- $xa$: (1,4)
- $a$: (1,2)
- $xa$: (1,5)
- $b$: (1,6)
- $a$: (1,3)
- $a$: (2,1)
Example of Construction (3)

Generalized suffix tree for $S_1 = xabxa$ and $S_2 = babxba$
(after completing all phases for $S_2$):
Two Implementation Details

1. Edge-labels of a generalized suffix tree come from more than one string

⇒ An edge-label $S_i[p \ldots q]$ needs to be represented in the compressed form by a \textit{triple} $(i, p, q)$

2. Some suffixes may occur in multiple strings (if they have a common end marker)

⇒ Store in each leaf $v$ \textit{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 $P_1, \ldots, P_k$, 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 $\mathcal{T}$ for $T[1 \ldots m]$, in time $O(m)$
2. For each pattern $P_i$ find its occurrences in $\mathcal{T}$; if $z_i \in \{0, \ldots, m\}$ is the number of them, this takes time $O(|P_i| + z_i)$

$\Rightarrow$ Total time of all searches is $O(\sum_{i=1}^{k} (|P_i| + z_i)) = O(n + z)$
A classic problem: How to find the longest substring common to two given strings?

Example: The longest common substring of “piilosana” 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 $v$
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

$\Downarrow \mathcal{L}(u)$ is a maximal common substring

Steps 2 and 3 can be implemented with simple tree-traversals in linear time (Exercise)
The above method gives us the result:

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

**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)
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** is a serious problem, which 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 $l$.

In other application areas this problem could be called plagiarism detection.
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

Total time is $\Theta(m)$, where $m$ the total length of the strings

(Since the string-length of the result could be $\Theta(m^2)$, linear time requires a compact representation of $L(v)$, say, as $(i, j)$ s.t. $L(v) = S_1[i \ldots j]$. Details left as an exercise.)
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

Why?

A: Substrings common to many bio-strings help to identify regions or patterns that may be critical to the function or structure of those strings (DNA, protein)

Why?

A: Critical regions are more highly conserved, since mutations in them have a small chance of surviving through generations (when compared to mutations in in-active parts)
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.
Let $S$ be a set of strings $\{S_1, \ldots, S_k\}$, of total length $n$

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 $S$

NB: These are unrelated to the $l'(i)$-values of BM preprocessing

We want to compute for each $i = 2, \ldots, k$ the value $l(i)$
(and a pointer to a corresponding substring)
Example of $l(i)$ values

Example:
The $l(i)$ values and corresponding substrings for $S = \{\text{sandollar, sandlot, handler, grand, pantry}\}$:

<table>
<thead>
<tr>
<th>$i$</th>
<th>$l(i)$</th>
<th>Substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>sand (or andl)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>and</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>and</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>an</td>
</tr>
</tbody>
</table>
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 \( \mathcal{T} \) for strings \( S_1$1, \ldots, S_k$ k \) (each terminated by a unique end marker).

\( \sim \) each leaf contains exactly one string identifier.

2. Compute how many \textit{distinct} string identifiers occur below any internal node of \( \mathcal{T} \).

Let \( C(v) \) be the number of distinct string identifiers in the leaves of \( \mathcal{T} \) below node \( v \).
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 $b[1 \ldots k]$ where $b[i] = 1$ iff identifier of string $i$ occurs in a leaf below $v$.

The vector for $v$ is computed by ORing the vectors of the child nodes $v_1, \ldots, v_l$ of $v$. That takes time $\Theta(lk)$.

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

They can counted in time $O(k)$ 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

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

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

3. Initialize $V(i)$ to zero for each $i = 2, \ldots, k$

4. Traverse tree $T$. At each internal node $v$ having string-depth $d$ and $C(v) = i$ set $V(i) := \max\{V(i), d\}$
Finding Frequent Substrings (3)

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

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

\[
l(k) := V(k);
\]

\[
\text{for } i := k - 1 \text{ downto } 2 \text{ do } l(i) := \max\{l(i + 1), V(i)\};
\]

Example: \( V(i) \) and \( l(i) \) for \( \{aba, abx, bcda, bcdb, bcdc\} \):

<table>
<thead>
<tr>
<th>( i )</th>
<th>( V(i) )</th>
<th>( l(i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 (b)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3 (bcd)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2 (ab)</td>
<td>3</td>
</tr>
</tbody>
</table>

(Computation of pointers to substrings is left as an exercise)
Consider the following extension of exact matching:

Define **matching statistic** $ms(i)$ to be the length of the longest substring of $P[1 \ldots n]$ that occurs in $T$ starting at position $i$.

**Example:** For $T = abcxabcdex$ and $P = qwabcrtyuioabcdw$, we have $ms(1) = 3$ (← $abc$) and $ms(5) = 4$ (← $abcd$).

Clearly $P$ has an occurrence starting at posn $i$ of $T$ iff $ms(i) = |P|$

Matching statistics can be used to reduce space requirements of some suffix-tree based solutions.

They are also central to a fast approximate matching method used in searching sequence databases.
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 $T[i \ldots m]$ 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 $\Theta(nm)$ time

Avoid explicit path traversals by simulating them with suffix links and skip/count tricks
Value $ms(1)$ is computed by matching $T[1 \ldots m]$ against $T$ explicitly (as the length of the matching path)

Computation of subsequent values $ms(j + 1)$ can be done similarly to computing extension $j + 1$ in Ukkonen’s algorithm:

Let $T[j \ldots i]$ be the path followed in $T$ to find out $ms(j)$

From the end of this path, back up at most one level to the closest node $v$ that either (a) is the root or (b) has a suffix link $(v, s(v))$

In case (a), compute $ms(j + 1)$ by traversing path $T[j + 1 \ldots]$ starting from the root
In case (b), let $\mathcal{L}(v) = x\alpha$, and $T[j \ldots i] = x\alpha\beta$

Now $s(v)$ is labeled by $\alpha$. Since path $\alpha\beta$ is in $T$, its end can be found by following path $\beta$ employing the skip/count trick down-wards from $s(v)$

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

Complexity of this method way is seen to be linear:
Theorem 7.8.1 All $m$ matching statistics are found in $O(m)$ time

Proof. Similar to the proof of complexity of Ukkonen’s algorithm (based on considering the current node depth)

In addition to that, how much work is done for explicit character comparisons along edges?

Comparisons for computing $ms(j+1)$ start either from the character that ended the computation of $ms(j)$ (with a mismatch), or from the next char (if the path led to a leaf)

$\Rightarrow$ at most one common text char examined for both of $ms(j)$ and $ms(j+1)$

$\Rightarrow$ less than $2m$ explicit character comparisons
An Extension, and an Application

Computation of $ms(i)$ can be extended to give also pointers $p(i)$ to corresponding pattern substrings (Gusfield, Sec. 7.8.3)

Note that $ms(i)$ determines the longest pattern substring that occurs starting at posn $i$ of $T$, but not its location in $P$

We’ve discussed finding longest common substrings of two strings, $S_1$ and $S_2$, using a generalized suffix tree built for them

Matching statistics can often be used to reduce the space needed by suffix-tree based solutions involving more than one string; This applies also here:
1. Let $S_1$ be the shorter of the strings. Build a suffix tree $T$ for it.

2. Using $T$, compute $ms(i)$ for each position of $S_2$.

3. Find and report a maximal common substring:

   $l := 0$; // length of a maximal common substr
   
   for $i := 1$ to $|S_2|$ do
     if $ms(i) \geq l$ then
       $l := ms(i)$; $j := i$;
     endif;
   endfor;
   
   Output $S_2[j \ldots j + ms(j) - 1]$;

Time: $\Theta(|S_1| + |S_2|)$ (as before), but now a smaller suffix tree is built.
Suffix arrays require only linear space, but with significant constant factors, space may become a problem, especially if the alphabet is large.

Central design issue for suffix trees (time/space trade-off):

How to locate edges that start by a given character?

- with a $\Theta(|\Sigma|)$ size array at each internal node
  $\leftrightarrow \Theta(1)$ access time to edges

- storing existing edges in some ordered structure
  $\leftrightarrow \Omega(\min\{\log |\Sigma|, \log m\})$ access time to edges
  (and similar additional factor to construction and search times)
Space Requirement of Suffix Trees

Example: (A single observation)

A suffix tree built from a 6000 bases long DNA sequence (using strmat software by Gusfield et al.) took 164,448 bytes, about 27.4 bytes/char

Suffix array (Manber & Myers, 1993) is a space-efficient variant of suffix trees, which solves exact matching and the substring problem almost as efficiently as a suffix tree.
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 (sanakirjajärjestys).

That is, alphabetic order, and longer strings preceded by their proper prefixes:

- auto
- autokauppa
- automaatti
- automaattinen

**NB:** A suffix array stores only $m$ numbers, so normally it can be stored in $m$ words (or $4m$ bytes), independent of the size of the alphabet.
**Example of a Suffix Array**

Example: Suffix array for string “mississippi” ($m = 11$)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$\text{Pos}[i]$</th>
<th>$T[\text{Pos}[i] \ldots m]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11: i</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8: ippi</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5: issippi</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2: ississippi</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1: mississippi</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10: pi</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9: ppi</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7: sippi</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>4: sisippi</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6: ssippi</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3: ssississippi</td>
<td></td>
</tr>
</tbody>
</table>
Exact Matching with a Suffix Array

Let $P[1 \ldots n]$ be the pattern searched for in $T[1 \ldots m]$

$P$ is a substring of $T$ iff it is a prefix of some suffix of $T$

The start positions for suffixes of $T$ that share a common prefix occur as a consecutive block of array $Pos$. (Why?)

Let $i$ be the smallest index of $Pos$ such that $P$ is a prefix of $T[Pos[i] \ldots m]$. It can be found (possibly not to exist) using binary search on array $Pos$, with $O(\log m)$ string comparisons

Define and compute similarly the largest index $j$ such that $P$ is a prefix of $T[Pos[j] \ldots m]$

Then $P$ occurs in $T$ starting at positions $Pos[i], \ldots, Pos[j]$
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 $O(n + \log m)$ \textit{worst-case} search time (by computing $O(m)$ “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 $R = m$
Improving the Search Time (2)

Maintain in a variable \( l \) the length of the prefix of \( P \) known to match the substrings starting at both positions \( Pos[L] \) and \( Pos[R] \) of \( T \).

All suffixes starting at positions \( Pos[L], \ldots, Pos[R] \) then share a common prefix \( P[1 \ldots l] \)

\( \Rightarrow \) it is sufficient to start the string comparison btw \( P \) and the suffix starting at \( Pos[(L + R)/2] \) from character \( l + 1 \)

The worst-case time is still \( \Theta(n \log m) \), but Myers and Manber report this to run in practice as fast as their \( O(n + \log m) \) worst-case time method.
Constructing a Suffix Array

A suffix array can be constructed in linear time by first constructing a suffix tree

- This is reasonable, if we have sufficient space for the construction, but need to store the result later in less space
- A direct $O(m \log m)$ time construction exists also

Suffix array $\text{Pos}$ for $T$ can be obtained by a “lexical” depth-first traversal of the suffix tree $\mathcal{T}$ for $T$

Let $e_1 = (v, u)$ and $e_2 = (v, w)$ be two edges in $\mathcal{T}$, 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
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 $T[1\ldots m]$ can be constructed in $O(m)$ 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.
Example of a Lexical Traversal

Example: Lexical depth-first traversal of the suffix tree for $xabxa$ (with the branch for the empty suffix excluded):

![Diagram of a suffix tree for $xabxa$ showing a lexical depth-first traversal.]
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 cerevisae* (brewer’s yeast) at the Max-Plank 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)
the bacterium causing Borreliosis, \(\sim\) 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 \(\sim\) 2 MB handled in quarter of an hour

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

Efficient external-storage (disk) implementations needed