Algorithm Theory
12 Suffix Trees

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08
Text Search

- **Scenarios**
- **Static texts**
  - Literature databases
  - Library systems
  - Gene databases
  - World Wide Web
- **Dynamic texts**
  - Text editors
  - Symbol manipulators
Properties of Suffix Trees

› Search index
  • for a text $\sigma$ in order to search for several patterns $\alpha$

› Properties
  • Substring searching in time $O(|\alpha|)$
  • Queries to $\sigma$ itself, e.g.:
    - Longest substring of $\sigma$ occurring at least twice
  • Prefix search:
    - all positions in $\sigma$ with prefix $\alpha$
Properties of Suffix Trees

• **Range search:**
  - all locations (substrings) in $\sigma$ belonging to an interval $[\alpha, \beta]$ with $\alpha_{\text{lex}} \leq \beta$, e.g.
    * abarakadabra, acacia $\in$ [abc, acc],
    * abacus $\notin$ [abc, acc]

• **Linear complexity:**
  - Space requirement and construction time in $O(|\sigma|)$
Trie

- **Trie:**
  - A tree representing a set of keys.
  - for alphabet $\Sigma$, set $S$ of keys, $S \subset \Sigma^*$
  - Key: string in $\Sigma^*$

- **Edge of a trie $T$**
  - labeled with a single character of $\Sigma$

- **Neighboring edges**
  - edges that lead to different children of a node
  - labeled with different characters

- **A leaf represents a key:**
  - The corresponding key is the string consisting of the edge labels along the path from the root to the leaf.

- **Keys are not stored in nodes!**
Suffix Tries

- Trie representing all suffixes of a string
- Example: $\sigma = \text{ababc}$

suffixes:
- $\text{ababc} = \text{suf}_1$
- $\text{babc} = \text{suf}_2$
- $\text{abc} = \text{suf}_3$
- $\text{bc} = \text{suf}_4$
- $\text{c} = \text{suf}_5$
**Suffix Trie**

- Internal nodes of a suffix trie correspond to substrings of $\sigma$.
- Each proper substring of $\sigma$ is represented by an internal node.
- Let $\sigma = a^n b^n$. Then, there are $(n+1)^2$ different substrings (or internal nodes).
  $\Rightarrow$ space requirement $O(n^2)$
Suffix Tries

- A suffix trie $T$ satisfies some of the desired properties:
  - **String matching** for $\alpha$:
    - Following the path with edge labels $\alpha$ takes $O(|\alpha|)$ time.
    - Leaves of the subtree = occurrences of $\alpha$
  - **Longest substring occurring at least twice**:
    - Internal node with maximum depth having at least two children
  - **Prefix search**
    - All occurrences of strings with prefix $\alpha$ are represented by the nodes of the subtree rooted at the internal node corresponding to $\alpha$. 

```
Example: Trie representing all suffixes of a string

String: ababc

Suffixes:
- ababc
- babc
- abc
- bc
- c
```
A suffix tree is obtained from a suffix trie by contracting unary nodes:

suffix tree = contracted suffix trie
Internal Representation of Suffix Trees

- Child-sibling representation
  - substring: pair of numbers $(i,j)$
- Example: $\sigma = ababc$
  - node $v = (v.l., v.u, v.c., v.s)$
- Further pointers (suffix links are added later)

```
(1,2)  b  (2,2)  c  (5,$)
\big|    \big|    \big|    \big|
ab     b    c    abc
\big|    \big|    \big|    \big|
(3,$)  c  (5,$)  c  (5,$)
\big|    \big|    \big|    \big|
abc    abc  c    abc
```

Further pointers (suffix links are added later.)
Properties of Suffix Trees

- (S1) No suffix is prefix of another suffix.
  - This holds if the last character of $\sigma$ is $\notin \Sigma$.

- Search:
  - (T1) edge = non-empty substring of $\sigma$.
  - (T2) neighboring edges:
    corresponding substrings start with different characters
Properties of Suffix Trees

- **Size**
  - (T3) each internal node (≠ root) has at least two children
  - (T4) leaf = (non-empty) suffix of σ.

- **Let n = |σ| ≠ 1.**
  - (by T4) then the number of leaves is n
  - (by T3) number of intervals ≤ n-1
  - implies space requirement O(n)
Construction of Suffix Trees

- **Definitions:**
  - **Partial path:** Path from the root to a node in T.
  - **Path:** A partial path ending at a leaf.
  - **Location** of a string $\alpha$: Node where the partial path corresponding to $\alpha$ ends (if it exists).

```
abc c abc c
```

```
ab
b
c
```
Construction of Suffix Trees

- **Extension of a string α:**
  - string with prefix α

- **Extended location of a string α:**
  - location of the shortest extension of α whose location is defined

- **Contracted location of a string α:**
  - location of the longest prefix of α whose location is defined

Example:
```
Example: ab\%& ababc
```

Diagram:
```
    abc
   /  
  ab  c
 /   / 
|   |   
|   |   
|   |   
|   |   
|   |   
```

Diagram:
```
    b
   / 
  ab c
 /   / 
|   |   
|   |   
|   |   
```
Construction of Suffix Trees

- **Definitions**
  - $\text{suf}_i$: suffix of $\sigma$ beginning at position $i$,
    - e.g. $\text{suf}_1 = \sigma$, $\text{suf}_n = \$. 
  - $\text{head}_i$:
    - longest prefix of $\text{suf}_i$ which is also a prefix of $\text{suf}_j$ for some $j < i$.

- **Example:**
  
  $\sigma = \text{bbabaabc}$  
  $\alpha = \text{baa}$ (has no location)  
  $\text{suf}_4 = \text{baabc}$  
  $\text{head}_4 = \text{ba}$
Construction of Suffix Trees

$\sigma = \text{bbabaabc}$

Diagram showing the construction of a suffix tree for the string $\sigma = \text{bbabaabc}$.
Naive Suffix Tree Construction

- Start with the empty tree $T_0$
- The tree $T_{i+1}$ is constructed from $T_i$ by inserting the suffix $\text{suf}_{i+1}$
- Algorithm suffix-tree
  
  1. Input: string $\sigma$
  2. Output: suffix tree $T$ for $\sigma$

```
1 n:= |\sigma|; T_0 := \emptyset;
2 for i := 0 to n – 1 do
3 insert \text{suf}_{i+1} into $T_i$, store the result in $T_{i+1}$;
4 end for
```
Naive Suffix Tree Construction

- All suffixes $suf_j$ with $j \leq i$ have a location in $T_i$
  - $head_{i+1} =$ longest prefix of $suf_{i+1}$ whose extended location exists in $T_i$

- **Definition:**
  - $tail_{i+1} := sufi+1 - head_{i+1}$ i.e. $suf_{i+1} = head_{i+1} tail_{i+1}$
  - $\Rightarrow$ (by S1) $tail_{i+1} \neq \varepsilon$.  


Naive Suffix Tree Construction

- **Example:** $\sigma = ababc$
  
  $suf_3 = abc$
  
  $head_3 = ab$
  
  $tail_3 = c$

\[
\begin{align*}
T_0 &= \text{Leaf} \\
T_1 &= \text{Leaf} \\
T_2 &= \text{Leaf} \quad \text{ababc} \quad \text{babc}
\end{align*}
\]
Naive Suffix Tree Construction

- Ti+1 can be constructed from Ti as follows:
  1. Determine the extended location of head_{i+1} in Ti and split the last edge leading to this location into two new edges by inserting a new node.
  2. Insert a new leaf as location for suf_{i+1}

\[ x = \text{extended location of head}_{i+1} \]
Naive Suffix Tree Construction

Example: $\sigma = ababc$

$head_3 = ab$
$tail_3 = c$
Naive Suffix Tree Construction

Algorithm *suffix-insertion*

**Input:** tree $T_i$ and suffix $suf_{i+1}$

**Output:** tree $T_{i+1}$

1. $v :=$ root of $T_i$
2. $j := i$
3. **repeat**
4. find child $w$ of $v$ with $\sigma_{w.l} = \sigma_{j+1}$
5. $k := w.l - 1$
6. **while** $k < w.u$ and $\sigma_{k+1} = \sigma_{j+1}$ **do**
7. $k := k + 1$; $j := j + 1$
8. **end while**
9. **if** $k = w.u$ **then** $v := w$
10. **until** $k < w.u$ or $w = \text{nil}$

/* $v$ is the contracted location of head_{i+1} */
11. insert the location of head_{i+1} and tail_{i+1} below $v$ into $T_i$

Running time of suffix-insertion: $O(n-i)$
Total time required for the naive construction: $O(n^2)$
Mc Creight´s Algorithm

\\- **Idea:**
  - Extended location of head\textsubscript{i+1} in T\textsubscript{i} is determined in constant amortized time.
  - When the extended location of head\textsubscript{i+1} in T\textsubscript{i} has been found: Creating a new node and splitting an edge takes O(1) time

\\- **Theorem 1**
  - Algorithm M constructs a suffix tree \( \sigma \) with \(|\sigma|\) leaves and at most \(|\sigma|-1\) internal nodes in time \(O(|\sigma|)\)
Suffix Links

Definition:

- Let $x?$ be an arbitrary string where $x$ is a single character and $? \text{ some (possibly empty) substring.}$
- For an internal node $v$ with edge labels $x?$ the following holds:
  - If there exists a node $s(v)$ with edge label $?$, then there is a pointer from $v$ to $s(v)$ which is called a suffix link.

![Diagram of suffix links]
Suffix Links

- **Idea:**
  - By following the suffix links, we do not have to start each search for a splitting point at the root node.
  - Instead, we can use the suffix links in order to determine these nodes more efficiently, i.e. in constant amortized time.
Suffix Tree: Example

$T_0 = \quad T_1 = \quad \text{bbabaabc}$

$suf_1 = \text{bbabaabc}$

$suf_2 = \text{babaabc}$

$head_2 = b$
Suffix Tree: Example

x = b
?=ε

\( T_2 = \)

abaabc

babaabc

\( suf_3 = abaabc \)

\( head_3 = \varepsilon \)

\( T_3 = \)

abaabc

babaab

\( suf_4 = baabc \)

\( head_4 = ba \)
$T_4 =$

```
abc
   ↑ 
  b
abaabc
   ↓ 
  a
   |
  babaabc
   ↓ 
  abc
     ↓ 
   baabc
```

$suf_5 = aabc$

$head_5 = a$
Suffix Tree: Example

$T_5 = \begin{cases} x=a & \text{?} = \varepsilon \\ x=b & \text{?} = a \end{cases}$

location of $\text{head}_5$

$suf_6 = \text{abc}$

$head_6 = \text{ab}$
Suffix Tree: Example

$T_6 = \ldots$

location of head$_6$

suf$_7 = bc$

head$_7 = b$
Suffix Tree: Example

$$T_7 =$$

```
   a
  / \  
 b   babaabc
 /    /
 abc  a
 |    |
 c    c
```

$$\text{suf}_8 = c$$
Suffix Tree: Example

\[ T_8 = \]

```
            a
           /\  
          /   \  
         a     c
        /\   /\  
       /   b   b
      a     abc  
     /\   /\   /\  
    c c aabc abc  
   /\ /\   /\   /\  
  b aabc baabc  
```

Algorithms Theory
Winter 2008/09
Suffix Tree: Application

- Usage of a suffix tree $T$:
  1. **Search for a string $\alpha$:**
     Follow the path with edge labels $\alpha$ (takes $O(|\alpha|)$ time).
     Leaves of the subtree = occurrences of $\alpha$
  2. **Search for the longest substring occurring at least twice:**
     Find the location of a substring with maximum weighted depth that is an internal node.
  3. **Prefix search:**
     All occurrences of strings with prefix $\alpha$ are represented by the nodes of the subtree rooted the location of $\alpha$ in $T$. 
4. Range search for \([\alpha, \beta]\)
$t = x \ a \ b \ x \ a \ \$ \\
1 \ 2 \ 3 \ 4 \ 5 \ 6$

Suffix Tree
Ukkonen’s Algorithm: Implicit Suffix Trees

- **Definition:**
  - An implicit suffix tree is a tree obtained from the suffix tree for $t$ by
    1. deleting every copy of $\$\$ from the edge labels,
    2. deleting edges that have no label,
    3. deleting unary nodes.
Ukkonen's Algorithm: Implicit Suffix Trees

\[ t = x a b x a \$ \]

1 2 3 4 5 6

Diagram of implicit suffix tree for the string $t = x a b x a \$$. The tree is constructed such that each node represents a suffix of $t$.
Ukkonen’s Algorithm: Implicit Suffix Trees

(1) deleting $ from the edge labels
Ukkonen’s Algorithm: Implicit Suffix Trees

(2) deleting edges that have no label

\[ t = x \ a \ b \ x \ a \$ \]

1 2 3 4 5 6

[Diagram of suffix tree with labels and nodes]
Ukkonen’s Algorithm: Implicit Suffix Trees

(3) deleting unary nodes

t = x a b x a $
1 2 3 4 5 6

Diagram: Implicit suffix tree for the string $t = x a b x a$$

Node labels:

- Node 1: $x a b x a$
- Node 2: $a b x a$
- Node 3: $b x a$
- Edge labels: $a b x a$

The diagram illustrates the structure of the implicit suffix tree for the string $t = x a b x a$.
Ukkonen´s Algorithm

- Let $t = t_1t_2t_3 ... t_m$.
- Ukk is an online algorithm: The suffix tree $ST(t)$ is constructed step by step by constructing a sequence of implicit suffix trees for the prefixes of $t$:
  - $ST(\varepsilon), ST(t_1), ST(t_1t_2), ..., ST(t_1t_2... t_m)$
- $ST(\varepsilon)$ is the empty implicit suffix tree, consisting of the root only.
- $ST(t_1t_2...t_i)$ is the implicit suffix tree containing all suffixes of $t_1t_2...t_i$
Ukkonen´s Algorithm

- This is an online approach in the sense that in each step, the implicit suffix tree for a prefix of \( t \) is created without knowledge of the rest of the input string \( t \).
- Since the algorithm reads the input string character by character from left to right, it works incrementally.
Ukkonen’s Algorithm

- **Incremental construction of an implicit suffix tree:**

  - **Induction basis:** \( ST(\varepsilon) \) consists of the root only.
  - **Induction step:** \( ST(t_1 \ldots t_i) \) is extended to \( ST(t_1 \ldots t_it_{i+1}) \) for all \( i < m \).

  - Let \( T_i \) be the implicit suffix tree for \( t[1\ldots i] \).
    - At first, we construct \( T_1 \): This tree has a single edge labeled with character \( t_1 \).
    - In phase \( i+1 \), we construct tree \( T_{i+1} \) from \( T_i \).
    - We iterate for \( i = 1 \ldots m-1 \).
Ukkonen’s Algorithm

Pseudo code for Ukk:
Construct tree $T_1$
for $i = 1$ to $m-1$ do
  begin{phase $i+1$}
    for $j = 1$ to $i+1$ do
      begin{extension $j$}
        In the current tree find the end of the path from the root labeled $t[j ... i]$. If necessary, extend that path by adding character $t[i+1]$, thus ensuring that string $t[j...i+1]$ is in the tree.
      end;
  end;
end;
Ukkonen’s Algorithm

t = a c c a $

\begin{align*}
T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
1 & \quad 1 & \quad 1 & \quad 1, 3, 2 \\
2 & \quad 2 & \quad 2 & \\
3 & & & \\
\end{align*}

step 1 \quad step 2 \quad step 3 \quad step 4
Ukkonen\’s Algorithm

- In extension j of phase i+1, the end of the path from the root labeled with substring t[j...i] is determined. Then, this substring is extended by adding the character t[i+1] to its end (unless t[i+1] already appears there).
- In phase i+1, string t[1...i+1] is first inserted into the tree, followed by strings t[2...i+1], t[3...i+1],.... (in extensions 1,2,3,...., respectively).
- Extension i+1 of phase i+1 inserts the single character string t[i+1] into the tree (unless it is already there).
Ukk: Suffix Extension Rules

› Extension j (in phase i+1) results from applying one of the following rules:

• **Rule 1:**
  - If the path t[j...i] ends at a **leaf**, character t[i+1] is added to the end of
    the label on that leaf edge.

• **Rule 2:**
  - If no path from the end of string t[j...i] starts with character t[i+1], then
    a new leaf edge labeled with character t[i+1] is created. A new internal
    node will also be created there if t[j...i] ends inside an edge.
    * This is the only extension that increases the number of leaves! The
      new leaf represents the suffix starting at position j.

• **Rule 3:**
  - If some path from the end of string t[j ...i] starts with character t[i+1], then
    string t[j...i+1] is already in the current tree, so we do nothing.
Ukkonen´s Algorithm

\[ t = a\ c\ c\ a\ $ \]
\[ t[1\ldots3] = acc \]
\[ t[1\ldots4] = acca \]

extend suffix 1
rule 1

extend suffix 2
rule 1

extend suffix 3
rule 2

\[ a \text{ is already in the tree} \]
rule 3
Ukkonen´s Algorithm

- During phase $i+1$ (when $T_{i+1}$ is constructed from $T_i$) the following holds:

  (1) If rule 3 applies in extension $j$, then the path labeled $t[j...i]$ in $T_i$ must continue with character $t[i+1]$. So, any path labeled $t[j'...i]$ for $j'\geq j$ also continues with character $t[i+1]$.

- Therefore, rule 3 again applies in extensions $j'=j+1,...,i+1$.

- Once rule 3 applies in an extension of phase $i+1$, this phase may be ended.
  - Why? If in extension $j$ rule 3 applies. Then,
    - $t[j,...,i+1]$ is prefix of some $t[k,...,i]$ with $k<j$
    - for any $m>1$: $t[j+m,...,i+1]$ is prefix of some $t[k+m, ...,i]$
    - $t[k+m,...,i]$ is already in $T_i$
Ukkonen´s Algorithm

(2) If a leaf is created in $T_i$, then it will remain a leaf in all successive trees $T_{i'}$ for $j' \geq j$ (once a leaf, always a leaf!).

- Reason: A leaf edge is never extended beyond its current leaf.
- $t = a\ c\ c\ a\ b\ a\ a\ c\ b\ a$
Ukkonen´s Algorithm

- Implication:
  - Leaf 1 is created in phase 1. In each phase i+1 there is an initial sequence of successive extensions (starting with extension 1) where rule 1 or 2 applies
    - The first time rule 3 applies the phase is terminated
  - Let $j_i$ denote the last extension in this sequence of phase i where rule 1 or rule 2 is applied
  - Then $j_i \leq j_{i+1}$
    - Phase i: extension j with $j \leq j_i$
      * If rule 1 is applied then in the next phase also rule 1 is applied in extension j of the phase i+1
        - then at the very same leaf t[i+1] is added
      * If rule 2 is applied then in the next phase rule 1 in extension j will be applied, since then t[i+1] can be added to the leaf
Ukkonen’s Algorithm

- Extensions according to rule 1 may be performed implicitly!
Ukkonen´s Algorithm

- Improving the algorithm:
  - In phase $i+1$, rule 1 applies in all extensions $j$ for $j \in [1, j_i]$.
    - Only constant time is required to do those extensions implicitly.
  - If $j \in [j_i + 1, i+1]$, then find the end of the path labeled $t[j...i]$ and extend it by character $t[i+1]$ according to rules 2 or 3.
  - If rule 3 applies, set $j_{i+1} = j - 1$ and end phase $i+1$. 
Ukkonen´s Algorithm

Example:

phase 1: compute extensions 1 ... j_1
phase 2: compute extensions j_1+1 ... j_2
phase 3: compute extensions j_2+1 ... j_3
... 
phase i-1: compute extensions j_{i-2}+1 ... j_{i-1}
phase i: compute extensions j_{i-1}+1 ... j_i
Ukkonen’s Algorithm

- As long as explicit extensions are performed, keep track of the index $j^*$ of the current explicit extension.
- During the execution of the algorithm, $j^*$ never decreases.
- As there are only $m$ phases (where $m = |t|$) and $j^*$ is bounded by $m$, the algorithm performs only $m$ explicit extensions.
Ukkonen’s Algorithm

- Extended pseudo code for Ukk:

  Construct tree $T_1$; $j_1 = 1$

  for $i = 1$ to $m-1$ do
    begin{phase $i+1$}
    Do all implicit extensions.
    for $j = j_i + 1$ to $i + 1$ do
      begin{extension $j$}
      In the current tree find the end of the path from the root labeled $t[j ... i]$. If necessary, extend that path by adding character $t[i+1]$, thus ensuring that string $t[j...i+1]$ is in the tree.
      $j_{i+1} := j$;
      if rule 3 was applied then $j_{i+1} := j - 1$ and phase $i+1$ ends;
    end;
  end;
Ukkonen´s Algorithm

\[ t = \text{pucupcupu} \]

\begin{align*}
  i: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
  \varepsilon & *p & pu & puc & pucu & pucup & pucupc & pucupcu & pucupcup & pucupcupu \\
  *u & uc & ucu & ucup & ucupc & ucupcu & ucupcup & ucupcupu \\
  *c & cu & cup & cupc & cupcu & cupcup & cupcupu \\
  u & *up & upc & upcu & upcup & upcupu \\
  p & *pc & pcu & pcup & pcupu \\
  c & cu & cup & *cupu \\
  u & up & *upu \\
  p & pu \\
  u
\end{align*}

- Suffixes that cause an extension according to rule 2 are marked with *.
- Underlined suffixes indicate the last extension where rule 2 applies.
- Suffixes that end a phase (the first time rule 3 applies) are colored blue.
Ukkonen's Algorithm

- **Idea:**
  - By following the suffix links, we do not have to start each search for a split point at the root node. Instead, we can use the suffix links in order to determine these nodes more efficiently, i.e. in constant amortized time.
Ukkonen’s Algorithm

- Using suffix links, extensions rules 2 and 3 can be applied more efficiently.
- An explicit extension takes amortized O(1) time (not shown here).
- Since there are only $m$ explicit extensions, the total running time of Ukkonen’s algorithm is $O(m)$ (where $m=|t|$).
Ukkonen’s Algorithm

- The true suffix tree:
- The final implicit suffix tree $T_m$ can be converted to a true suffix tree in $O(m)$ time.

1. Add a terminal symbol $\$\$ to the end of $t$.
2. Let Ukkonen’s algorithm continue with this character.

- The resulting tree is the true suffix tree where no suffix is a prefix of another suffix and where each suffix ends at a leaf.
Algorithm Theory

12 Text Search

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08