Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings

Shunsuke Inenaga
Kyushu University, Japan
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.
- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.

![Diagram showing sensors, an index, and a user interacting with strings]

- $s_1$
- $s_2$
- $s_3$

Strings:
- $T_1$: abcaab
- $T_2$: cca
- $T_3$: bcdax
Fully-Online Indexing of Multiple Strings

- Goal: Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.

- Motivation: Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.

---

**Diagram:**

- **sensors:** \( s_1, s_2, s_3 \)
- **index:** \( T_1: \text{abcaabc}, T_2: \text{cca}, T_3: \text{bcdax} \)
- **user**
Fully-Online Indexing of Multiple Strings

- **Goal:** Indexing multiple strings in a **fully-online manner** where each string can grow **any time**.

- **Motivation:** Indexing multi online/streaming data.
  - Sensing data, trajectory data, SNS, etc.

![Diagram](image)
Overview of This Work

- We will consider **suffix trees** and **DAWGs** as indexing structures for fully-online multiple strings.

- For **suffix trees**, we propose a Weiner-type algorithm where strings grow **from right to left**.

- For **DAWGs**, we propose a Blumer et al.-type algorithm where strings grow **from left to right**.

- Our model of computation is the **pointer machine** that is strictly weaker than the word RAM.
The suffix tree of multiple strings is a path-compressed trie that represents all suffixes of the strings.

\[ T_1 = \text{cabaa}$ \]
\[ T_2 = \text{abaab}$ \]
Suffix Links

If \( av \) is a node and \( a \) is a character, then \( \text{suffix\_link}(av) = v \).

\[
T_1 = \text{cabaa}\$
\]
\[
T_2 = \text{abaab}\#
\]
If $av$ is a node and $a$ is a character, then $\text{suffix}\_\text{link}(av) = v$.

$T_1 = \text{cabaa}\$  
$T_2 = \text{abaab}\#$
Hard Weiner Links

The reversed suffix links with character labels are called hard Weiner links.

\[ T_1 = \text{cabaa}\$ \]
\[ T_2 = \text{abaab}\# \]

Not all links are shown.
Soft Weiner Links

Soft Weiner links are “generalized” Weiner links.

\[ T_1 = \text{cabaa}$ \]
\[ T_2 = \text{abaab#} \]

There is no node for \text{cabaa}.

Not all links are shown.
Soft Weiner Links

$T_1 = cabaa\$$

$T_2 = abaab\#$

Soft Weiner link of node $v$ with label $c$ points to the child node of locus for $cv$.

Soft Weiner links are “generalized” Weiner links.

Not all links are shown.
Soft Weiner Links

Soft Weiner links are “generalized” Weiner links.

\[ T_1 = \text{cabaa}$ \]
\[ T_2 = \text{abaab}$\]

Not all links are shown.
**DAWGs**
[Blumer et al. 1987]

The **DAWG** of multiple strings is a linear-size automaton that recognizes all substrings of the strings.

\[
S_1 = \$aabac \\
S_2 = \#baaba
\]
Duality of Suffix Trees and DAWGs

A) There is a one-to-one correspondence between the nodes of the suffix tree of strings and the nodes of the DAWG of the reversed strings.

Suffix Tree of $T_1 = cabaa\#$
$T_2 = abaab\#$

DAWG of $S_1 = $aabac
$S_2 = \#baaba$
**Duality of Suffix Trees and DAWGs**

B) There is a one-to-one correspondence between the Weiner links of the suffix tree of strings and the edges of the DAWG of the reversed strings.

Suffix Tree of

\[
T_1 = \text{cabaa}$
\]
\[
T_2 = \text{abaab}$
\]

DAWG of

\[
S_1 = \text{aabc}$
\]
\[
S_2 = \text{baaba}$
\]
### Previous and This Work (Suffix Trees)

<table>
<thead>
<tr>
<th>Right-to-Left Fully-Online Suffix Tree</th>
<th>Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>algorithm</strong></td>
<td><strong>single string</strong></td>
</tr>
<tr>
<td>Weiner</td>
<td>$O(n \log \sigma)$</td>
</tr>
<tr>
<td>Takagi et al.</td>
<td>$O(n \log \sigma)$</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td>$O(n (\log \sigma + \log d))$</td>
</tr>
</tbody>
</table>

$n$: total string length, $\sigma$: alphabet size, $d$: max. # in-coming Weiner links

Both $O(n \log \sigma) \subseteq O(n \log n)$ and $O(n (\log \sigma + \log d)) \subseteq O(n \log n)$ hold

→ The new algorithm achieves the same worst-case complexity on a **weaker model** of computation (**pointer machine**).
Previous and This Work (DAWGs)

<table>
<thead>
<tr>
<th>Left-to-Right Fully-Online DAWG Construction Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>algorithm</td>
</tr>
<tr>
<td>Blumer et al.</td>
</tr>
<tr>
<td>Takagi et al.</td>
</tr>
<tr>
<td>This work</td>
</tr>
</tbody>
</table>

$n$: total string length, $\sigma$: alphabet size, $d$: max. # in-coming Weiner links

Takagi et al.’s method only maintains an implicit representation of DAWG.

→ The new algorithm is the **first non-trivial algorithm** that maintains an **explicit representation of DAWG** for fully-online multiple stings.
**Pointer Machine** [cf. Tarjan 1979]

- The **pointer machine** is an abstract model of computation where the state of computation is stored as a digraph. Each node contains a constant number of data and pointers.

- The pointer machine supports instructions (1)-(3):
  1. creating / deleting nodes and pointers;
  2. manipulating data;
  3. performing comparisons,

but it **does NOT support** word RAM instructions (4)-(5):
  4. address arithmetics;
  5. unit-cost bit-wise operations.

- Still, the pointer machine serves as a good basis for modelling linked structures such as trees and graphs.
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Find the lowest ancestor $\nu$ of leaf $T$ that has Weiner link with character $a$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Find the lowest ancestor $v$ of leaf $T$ that has Weiner link with character $a$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Find the lowest ancestor $\nu$ of leaf $T$ that has Weiner link with character $a$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$. 

Follow the Weiner link labeled $a$ from $v$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Follow the Weiner link labeled $a$ from $v$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Follow the Weiner link labeled $a$ from $v$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Follow the Weiner link labeled $a$ from $v$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Split the incoming edge at string depth $|av| = |v| + 1$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Split the incoming edge at string depth $|av| = |v| + 1$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Redirect soft Weiner links.
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Redirect soft Weiner links.
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Insert new leaf for $aT$ as a child of $av$. 

Diagram showing the process of updating the string $T$ to $aT$ and inserting a new leaf for $aT$ as a child of $av$. The diagram includes arrows indicating the direction of the string updates and the location of the new leaf.
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$. 

Insert new leaf for $aT$ as a child of $av$. 
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$. 

BEFORE  

AFTER
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

Before

Online single string

$$\sum_{i=1}^{n} r_i \in O(n)$$

[Blumer et al. 1985]

Fully-online multiple strings

$$\sum_{i=1}^{n} r_i \in \Omega(n^{1.5})$$

[Takagi et al. 2020]
Weiner’s Algorithm (Blumer et al. version)

Update string $T$ to $aT$.

To avoid the $\Omega(n^{1.5})$ work, we reduce the sub-problem of redirecting Weiner links to the ordered split-insert-find problem.
Ordered Split-Insert-Find

The *ordered split-insert-find* problem is to maintain a data structure on ordered sets which supports the following operations and queries efficiently:

- **Make-set**, which creates a new list that consists only of a single element;
- **Split**, which splits a given set into two disjoint sets, so that one set contains only smaller elements than the other set;
- **Insert**, which inserts a new single element to a given set;
- **Find**, which answers the name of the set that a given element belongs to.
Reduction to Ordered Split-Insert-Find

Maintain the set of string depths of origin nodes of Weiner links

BEFORE

\{s_1, s_2, s_3, s_4, s_5, s_6\}
Reduction to Ordered Split-Insert-Find

Maintain the set of string depths of origin nodes of Weiner links

BEFORE

{\{s_1, s_2, s_3, s_4, s_5, s_6\}}

AFTER

{\{s_1, s_2, s_3\}}

{\{s_4, s_5, s_6\}}
Reduction to Ordered Split-Insert-Find

BEFORE

AFTER

$av$ is the parent of new leaf $aT$
Reduction to Ordered Split-Insert-Find

BEFORE

AFTER

New node $av$ copies out-going Weiner links from its child.

$av$ is the parent of new leaf $aT$
Reduction to Ordered Split-Insert-Find

BEFORE

AFTER

\textit{av} is the parent of new leaf \textit{aT}
Ordered Split-Insert-Find by AVL-trees

For each suffix tree node $u$, we maintain an AVL tree such that each AVL tree node stores the string depth of the origin node of an in-coming Weiner link.

AVL tree for node $u$

Works on the pointer machine.
Ordered Split-Insert-Find by AVL-trees

Now, each Weiner link to a suffix tree node $u$ points to the corresponding node in the AVL tree for node $u$. The root of this AVL tree is connected to suffix tree node $u$. 

![AVL tree for node $u$]
Ordered Split-Insert-Find by AVL-trees

An AVL tree of $d$ elements supports operations
Make-set, Split, Inert, and Find in $O(\log d)$ time each.

$\Rightarrow O(\log d)$-time maintenance of Weiner links.

BEFORE

AFTER

AVL tree for node $u$

AVL tree for node $av$

AVL tree for node $u$
Splitting an AVL-tree

- Let $X$ be an AVL tree for the set $\{s_1, \ldots, s_d\}$ of integers.
- Given an element $s_j$ in the set, we split the AVL tree $X$ into two AVL trees, $X_1$ for $\{s_1, \ldots, s_j\}$ and $X_2$ for $\{s_{j+1}, \ldots, s_d\}$.
- This split operation can be done in $O(\log d)$ time (next slide).
Splitting an AVL-tree

- Consider the search path for $s_j$ in the AVL tree $X$.
- By splitting $X$ with this path, we obtain
  - green nodes and subtrees containing elements at most $s_j$
  - orange nodes and subtrees containing elements larger than $s_j$
- Using monotonicity, we can merge each of them in $O(\log d)$ time.
Main Results

**Theorem 1**

There is a pointer-machine algorithm which builds the **suffix tree** of **right-to-left** fully-online multiple strings in $O(n (\log \sigma + \log d))$ time and $O(n)$ space. Each suffix-tree edge traversal takes $O(\log \sigma)$ time.

**Theorem 2**

There is a pointer-machine algorithm which builds the **DAWG** of **left-to-right** fully-online multiple strings in $O(n (\log \sigma + \log d))$ time and $O(n)$ space. Each DAWG-edge traversal takes $O(\log \sigma + \log d)$ time.

$n$: total string length, $\sigma$: alphabet size, $d$: max. # in-coming Weiner links
Conclusions and Open Question

We proposed pointer-machine algorithms for fully-online construction of suffix trees and DAWGs on multiple strings running in \( O(n (\log \sigma + \log d)) \) time and \( O(n) \) space.

We have not found an instance where the \( n \log d \) term in our time complexity becomes \( \Theta(n \log n) \) or \( \omega(n) \).

We have only found a bad instance which requires sub-linear \( \Omega(\sqrt{n} \log n) \) work to maintain the AVL trees.

Would it be possible to construct suffix trees / DAWGs for fully-online multiple multiple strings in \( O(n \log \sigma) \) time on the pointer machine?