Compressed Suffix Arrays and Suffix Trees

with Applications to

Text Indexing and String Matching

Duke University

Jeff Vitter

Università di Pisa

Roberto Grossi

Compressed Suffix Arrays and Suffix Trees
String Matching: The Problem

Motivation: Huge collections of textual data.

Input:

★ Text $T$ of length $n$.
★ Pattern $P$ of length $m \leq n$.
★ Binary alphabet.

Types of Queries:

★ Existential: Does $P$ occurs in $T$?
★ Counting: Give number $occ$ of occurrences of $P$ in $T$.
★ Enumerative: List all positions where $P$ occurs in $T$.
## String Matching: Revisited Tour on the Bounds

<table>
<thead>
<tr>
<th>Search time</th>
<th>Space (words)</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(m + n)$</td>
<td>$O(m)$</td>
<td>Knuth-Morris-Pratt’77, ... long list.</td>
</tr>
<tr>
<td>$O(m + n)$</td>
<td>$O(1)$</td>
<td>Galil-Seiferas’83, Crochemore-Perrin.</td>
</tr>
<tr>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>Morrison’68, Weiner’73, ... long list.</td>
</tr>
<tr>
<td>$o(m)$ ?</td>
<td>$o(n)$ ?</td>
<td>Yes!! The topic of this talk.</td>
</tr>
</tbody>
</table>

- Typically pattern length $m \ll$ text length $n$.
- Binary alphabet and standard RAM with wordsize $O(\log n)$. 
String Matching: Text Indexing

- Basic idea from Morrison’68, Weiner’73, ...:
  1. Scan the text only initially.
  2. Build an index: e.g., suffix array or suffix tree (Patricia trie storing all suffixes of the text).

- Search time drops from $O(m + n)$ to $O(m)$ (for counting and existence queries).

- Additional output sensitive cost $O(occ)$ (for enumerative queries).

- Space increases from $O(1)$ to $O(n)$ words, i.e., $O(n \log n)$ bits!
String Matching: Space

☆ **Criticism on greediness of space:** $\Omega(n \log n)$ bits.

- Need to store $\Omega(n)$ positions explicitly:
  
  Pattern $P \rightarrow$ **Suffix tree index** $\rightarrow$ $P$ occurs at $i$

- Each position requires at least $\log n$ bits.
- Index at least $\log n$ times **larger** than the text!

☆ **Inverted lists take less space** (but less functionality).
Space Reduction Issues

★ Analysis of constants in $O(n \log n)$ bit space:
  Manber-Myers’93, Andersson-Nilsson’95, Kärkkäinen’95, Clark’96,
  Clark-Munro’96, Kurtz’98 (many refs), Giegerich-Kurtz-Stoye’99,

★ Making suffix trees sparse:
  Morrison’68, Gonnet-BaezaYates-Snider’92, Manber-Wu’94,
  Colussi-DeCol’96, Kärkkäinen-Ukkonen’96ab,
  Andersson-Larsson-Swanson’99

★ LZ and BW compression:
  Kärkkäinen-Sutinen’98, Ferragina-Manzini’00

★ Succinct representation:
  Jacobson’89, Clark-Munro’96, Munro-Raman’97,
  Munro-Raman-SrinivasaRao’98
Break through both the time barrier of $O(m)$ time and the space barrier of $O(n \log n)$ bits.

Compressed suffix arrays:

- *compress* in $O(n)$ bits and $O(n)$ time. [\(O(n \log \log n)\) bits]
- *lookup* in $O(\log^\epsilon n)$ time, $\epsilon < 1$. [\(O(\log \log n)\) time]

Provably as good as *inverted lists* in space usage and more functionality on arbitrary substrings.
Our Results (2)

- Compressed suffix trees in $O(n)$ bits: *same* space as that of text.
- Text indexing on $T$: only $O(n)$ bits.
  
  - Existential & Counting in $o(m)$ time, specifically
    \[
    \begin{cases}
      O(1) & \text{for } m < \epsilon \log n; \\
      O(m/\log n + \log^\epsilon n) & \text{otherwise.}
    \end{cases}
    \]
  
  - Enumerative for $occ$ occurrences in additional time
    \[
    \begin{cases}
      O(occ) & \text{for } m = \Omega((\log^3 n) \log \log n) \text{ or } occ = \Omega(n^\epsilon); \\
      O(occ \log^\epsilon n) & \text{otherwise.}
    \end{cases}
    \]
Definition of *suffix array* \(SA\):

\[SA[i] = \text{starting position of } i\text{th lexicographically smallest suffix.}\]

Example for \(n = 7\) (text length 6):

( Assume that \(a < \# < b\) )

\[
\text{Input text: } \hspace{2cm} \text{Sorted list of suffixes}
\]

\[
\begin{align*}
\text{bababa } \# \\
\text{Suffix array:} & \quad 6 \ 4 \ 2 \ 7 \ 5 \ 3 \ 1 \\
\end{align*}
\]

\[
\begin{align*}
6 : \text{a}\# \\
4 : \text{aba}\# \\
2 : \text{ababa}\# \\
7 : \# \\
5 : \text{ba}\# \\
3 : \text{baba}\# \\
1 : \text{bababa}\#
\end{align*}
\]
Suffix Trees (Patricia's)
The Patricia topology:

\[ (n \log n) o \log u \text{ bits} \]

The suffix pointers (i.e., suffix array): \( u \log n \) bits

Munro-Raman-Srinivasan '98

\[ O(1) \text{ bits each and retrieval time} \]

Clark-Munro '96

\[ O(1) \log \log n \text{ retrieval time} \]

The Patricia trees compress a chain of many nodes, so the total number of

State of the Art on Suffix Trees
Guarantee on the time complexity:

- Same set of supported operations.

Each data structure occupies \( O(\log C \log n) \) bits.

\[ \text{DATA STRUCTURE} \]

\[ \text{DATA STRUCTURE} \]

This talk: suffix arrays + operations compress + lookups.

Abstract Data Type Optimization [Jacobson'89]
**Example: Suffix array permutations for $n = 5$**

<table>
<thead>
<tr>
<th>aaaa#</th>
<th>aaab#</th>
<th>aaba#</th>
<th>aabb#</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>12354</td>
<td>14253</td>
<td>12543</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abaa#</th>
<th>abab#</th>
<th>abba#</th>
<th>abbb#</th>
</tr>
</thead>
<tbody>
<tr>
<td>34152</td>
<td>13524</td>
<td>41532</td>
<td>15432</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>baaa#</th>
<th>baab#</th>
<th>baba#</th>
<th>babb#</th>
</tr>
</thead>
<tbody>
<tr>
<td>23451</td>
<td>23514</td>
<td>42531</td>
<td>25143</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bbaa#</th>
<th>bbab#</th>
<th>bbaa#</th>
<th>bbbb#</th>
</tr>
</thead>
<tbody>
<tr>
<td>34521</td>
<td>35241</td>
<td>45321</td>
<td>54321</td>
</tr>
</tbody>
</table>

- Among the $n!$ permutations, only $C(n) = 2^{n-1}$ are valid.
- $\implies$ data structure size $\geq \log C(n) = n - 1$ bits.
- 1-1 correspondence between suffix arrays and strings:
  $\implies$ naïve $O(n)$-time compress & lookup.  
  *(TOO SLOW!)*
Our Idea: Recursive Deconstruction (1)

1. Link each valid permutation to the suffixes.
2. Start out with $SA^0$ suffix array.
3. For $0 \leq \log u \leq u$, perform recursive step $k$.

This replaces $SA^k$ with a suffix array $SA^{k+1}$ of half its size.

After $\log u$ recursive steps, we have

\[
\frac{n}{u} \leq \frac{\log |SA^1|}{u} = \frac{\log |SA^2|}{u} = \frac{\log |SA^3|}{u} \leq \frac{n}{2^\theta k+1}
\]

As a result, \( j_{SA^k+j} = j_{SA^j} + \frac{n}{2^\theta k+1} \).

Center for Geometric Computing

\[(u)O \leftarrow \frac{u \log |SA^k|}{u} = \frac{\log |SA^k_j|}{u} \]

\[\text{Index only the suffixes starting at text positions } \gamma \text{, rank } \gamma \text{, } \gamma \text{, } \gamma + 1, \ldots, \gamma + 2^\theta k+1 \text{ of half its size:}
\]

\[\forall A \cap \{ \gamma \text{, rank } \gamma \text{, } \gamma \text{, } \gamma + 1, \ldots, \gamma + 2^\theta k+3 \}
\]
Let $SA_k \leftrightarrow \{ B_k, \ rank_k, \ \Psi_k \} \cup SA_{k+1}$.

- $B_k$ = Bit vector, such that $B_k[i] = 1$ iff $SA_k[i]$ is even.
- $rank_k(j) = \#1s$ in the first $j$ bits of $B_k$.
- Companion items:
  
  $\Psi_k(i) = \begin{cases} 
  j & \text{if } SA_k[i] \text{ is odd and } SA_k[j] = SA_k[i] + 1; \\
  i & \text{otherwise.}
  \end{cases}$

- $SA_{k+1} \leftrightarrow$ Pack the even values and divide each of them by 2.
Level $k = 0$: 

- $a$ list: $\{\}$
- $b$ list: $\{\}$

Level $k = 1$: 

- $a$ list: $\{1, 13, 17, 21, 27\}$
- $b$ list: $\{7, 8, 10, 12, 16\}$

Level $k = 2$: 

- $a$ list: $\{2, 14, 15, 18, 23, 28, 30, 31\}$
- $b$ list: $\{2, 4, 5\}$

Implementation of $\Psi_\gamma$'s
Compressed Suffix Arrays

✿ **compress**: Apply $\ell = \Theta(\log \log n)$ recursive steps.

- Level $k < \ell$ stores only $B_k$, $\text{rank}_k$, $\Psi_k$ in compressed form.
- Last level $\ell$ stores only $SA_\ell$ in $O(n)$ bits.
- Reconstruct $SA_k$ from $SA_{k+1}$ by the formula:

$$SA_k[i] = 2 \cdot SA_{k+1}[\text{rank}_k(\Psi_k(i))] + (B_k[i] - 1).$$

✿ **rlookup$(i, k)$**:

```
if $k = \ell$ then $SA_\ell[i]$
else $2 \times rlookup(\text{rank}_k(\Psi_k(i)), k + 1) + (B_k[i] - 1)$.
```

Top-level: $k = 0$ to get $SA[i]$. 
Bounds for compressed suffix arrays:

\[ \text{lookup}(i) \in O(\log \log n) \text{ time;} \]
\[ \text{compress in } O(n \log n) \text{ bits and } O(n) \text{ preprocessing time;} \]

\[ \text{lookup}(i) \in O(\log n) \text{ time;} \]
\[ \text{compress in } O(n \log n) \text{ bits and } O(n \log n) \text{ preprocessing time;} \]
Multi-Level Text Index: Old and New Ingredients

★ LZ-index for short patterns of length $m < \epsilon \log n$.
  
  [Kärkkäinen-Sutinen’98]

★ For patterns of length $m \geq \epsilon \log n$:
  
  • Top level: Sparse suffix tree (Patricia) with $O(n/\log n)$ nodes.
    
    [Kärkkäinen-Ukkonen’96ab]
  
  • $O(1)$ middle levels: Space efficient Patricias with $O(\log^\epsilon n)$ nodes.
    
    [Munro-Raman-SrinivasaRao’98]
  
  • Last level: Our compressed suffix array.
  
  • Trick: Perfect hash to speed up the Patricia traversal.
Compressed suffix trees and text indexing:

- Index data structure on text $T$ in $O(n)$ bits.
- Any pattern string $P$ of $m$ bits packed into $O(m/\log n)$ words:
  
  i. Existential $\exists$ Counting in $o(m)$ time:
      \[
      \begin{align*}
      O(1) & \quad \text{for } m = o(\log n); \\
      O(m/\log n + \log^\epsilon n) & \quad \text{otherwise}
      \end{align*}
      \]
  
  ii. Enumerative for $occ$ occurrences in additional time
      \[
      \begin{align*}
      O(occ) & \quad \text{for } m = \Omega((\log^3 n) \log \log n) \text{ or } occ = \Omega(n^\epsilon); \\
      O(occ \log^\epsilon n) & \quad \text{otherwise}
      \end{align*}
      \]
The first index structure to break through both the time barrier of $O(m)$ time and the space barrier of $O(n \log n)$ bits.

- $O(n)$-bit compress and $O(1)$-time lookup?
- Characterize combinatorially the suffix array permutations?
- Small number of errors in the pattern queries?

Follow-ups: lower bound on the index size; compressed texts.