Applications of Succinct Dynamic Compact Tries to Some String Problems

Takuya Takagi (IST, Hokkaido University, Japan)
Takashi Uemura (Chowa Giken Co., Japan)
Shunsuke Inenaga (Kyushu University, Japan)
Kunihiro Sadakane (National Institute of Informatics, Japan)
Hiroki Arimura (IST, Hokkaido University, Japan)
Background

- Massive amount of string data become available on Internet.
  - e.g. Genome sequences, Web pages, and Twitter
- Increasing interests in string processing technologies.
- **Compact and efficient data structure** for massive string data attracts much attention.
The dynamic compact trie [Fredkin ’60]

○ A classical data structure for a set of strings.
  • Each branch represents a string
  • A trie with path compression.

○ Plays essential role in many string problems.
  • online sparse suffix tree construction [Karkkainen & Ukkonen ’96]

1: ART
2: ARC
3: BAD
4: BAG
5: BUD
6: CODE
7: COOK

A set $S$ of $K = 7$ strings

A compact trie $T_S$ for $S$

The dynamic compact trie

- **Linear space:** $S = O(N \log \sigma + K \log N)$ bits for storing $K$ strings of total length $N$

- **Operations**
  - **Pattern matching** (from arbitrary node): $O(P \log \sigma)$ time
  - **Insert/delete:** $O(P \log \sigma)$ time
  - **parent and child:** $O(\log \sigma)$ time

- **Applications**
  - Sparse suffix tree construction [Karkkainen et al.’96]
  - Dynamic dictionary matching [Hon, Lam, et al. ’09]

$P$: length of a pattern string. $\sigma$: alphabet size
The dynamic compact trie

- Linear space: storing K strings

- Operations
  - Pattern matching (from arbitrary node): $O(P \log \sigma)$ time
  - Insert/delete: $O(P \log \sigma)$ time
  - Parent and child: $O(\log \sigma)$ time

- Applications
  - Sparse suffix tree construction [Karkkainen et al.'96]
  - Dynamic dictionary matching [Hon, Lam, et al. '09]

P: length of a pattern string. $\sigma$: alphabet size
The dynamic compact trie

- Linear space: storing K strings
- The time complexities of these operations dominates the total time complexities of

So, we want to speed up the dynamic compact trie on Word RAM!!!!!!!

Applications

- Sparse suffix tree construction [Karkkainen et al.’96]
- Dynamic dictionary matching [Hon, Lam, et al. ’09]

P: length of a pattern string. σ: alphabet size
Def: Word RAM model

- has **w-bit registers**.
- can perform **bitwise** (&, |, ~, >>, <<) and **arithmetic** (+, ×) operations in constant time.
- can read consecutive w bits in constant time.

*note: We do not use multiplication.*

- **Packed string technique** [Kiki&Bille, TCS, '12]
  
  **Basic idea:** By reading \( \alpha = \frac{w}{\lg \sigma} \) consecutive letters in one step.
# Problems Accelerated on Word RAM

<table>
<thead>
<tr>
<th>Problem</th>
<th>Classic RAM*</th>
<th>Word RAM</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>O(lg N)</td>
<td>O(1)</td>
<td>Deterministic hash [Szemeredi et al.]</td>
</tr>
<tr>
<td>Search (Predecessor)</td>
<td>O(lg N)</td>
<td>O(lg lg M)</td>
<td>Y-fast trie [Willard '84]</td>
</tr>
<tr>
<td>Sorting</td>
<td>O(N lg N)</td>
<td>O(N lg lg N)</td>
<td>Nearly linear sorting [Andersson, Hagerup, Nilsson and Raman '95]</td>
</tr>
<tr>
<td>String matching</td>
<td>O(N + M)</td>
<td>O((N + M)/w)†</td>
<td>Packed string [Belazzougui, IWOCA'10] [Kiki&amp;Bille, TCS, '12]</td>
</tr>
<tr>
<td>RegExp matching</td>
<td>O(NM)</td>
<td>O(NM/w)</td>
<td>Four-Russian [Myers’92]</td>
</tr>
<tr>
<td>uncompacted trie operations**</td>
<td>O(P lg σ)</td>
<td>O(P f(n)/α)</td>
<td>faster dynamic trie [Jansson, Sadakane &amp; Sung FSTTCS'09]</td>
</tr>
<tr>
<td>Compact trie operations**</td>
<td>O(P lg σ)</td>
<td>?</td>
<td>This work</td>
</tr>
</tbody>
</table>

*Classic RAM supports comparison only, without supporting bit-wise or arithmetic operations.
†for binary string. **Pattern matching (from arbitrary node) and insert/delete
## Problems Accelerated on Word RAM

<table>
<thead>
<tr>
<th>Problem</th>
<th>Classic RAM*</th>
<th>Word RAM</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Predicate)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String matching</td>
<td>$O(N + M)$</td>
<td>$O((N + M)/w)$†</td>
<td>Packed string [Belazzougui, IWOCA’10] [Kiki&amp;Bille, TCS, ’12]</td>
</tr>
<tr>
<td>RegExp matching</td>
<td>$O(NM)$</td>
<td>$O(NM/w)$</td>
<td>Four-Russian [Myers’92]</td>
</tr>
<tr>
<td>uncompacted trie operations**</td>
<td>$O(P \lg \sigma)$</td>
<td>$O(P f(n)/\alpha)$</td>
<td>faster dynamic trie [Jansson, Sadakane &amp; Sung FSTTCS’09]</td>
</tr>
<tr>
<td>Compact trie operations**</td>
<td>$O(P \lg \sigma)$</td>
<td>?</td>
<td>This work</td>
</tr>
</tbody>
</table>

*Classic RAM supports comparison only, without supporting bit-wise or arithmetic operations.
†for binary string.
**Pattern matching (from arbitrary node) and insert/delete
Succinct dynamic uncompacted trie [JSS’07]

- Jansson, Sadakane, & Sung [JSS’07] proposed the faster and succinct version of a **uncompacted** trie on Word RAM.
- This requires the **same space** as compact trie!

<table>
<thead>
<tr>
<th></th>
<th>Classic compact trie [Fredkin’60]</th>
<th>Succinct dyn. trie [JSS’07]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trie type</strong></td>
<td>compact</td>
<td>uncompacted</td>
</tr>
<tr>
<td><strong>Additional Space (bit)</strong></td>
<td>O(n lg n)</td>
<td>O(n lg σ)</td>
</tr>
<tr>
<td><strong>Time for operations</strong></td>
<td>O(P lg σ)</td>
<td>O((P lg σ) f(n) / log(n))</td>
</tr>
<tr>
<td><strong>Online sparse suffix tree construction</strong></td>
<td>Linear time</td>
<td><strong>Quadratic time</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A uncompacted trie</th>
</tr>
</thead>
</table>

n: Total text length, σ: Alphabet size, P: Pattern length, f(n) = (lglg n)²/(lglglg n): operation time of [Beame&Fich’02]. Speed-up factor α = lgσ n.
Succinct dynamic uncompacted trie [JSS’07]

- The succinct uncompacted trie [JSS’07] can not be applied to suffix tree!!
  - This requires quadratic time and space

<table>
<thead>
<tr>
<th></th>
<th>Linear time</th>
<th>Quadratic time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online sparse suffix tree construction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n: Total text length, σ: Alphabet size, P: Pattern length, \( f(n) = (\lg \sigma n)^2/(\lg \lg \sigma \lg n) \): operation time of [Beame&Fich’02]. Speed-up factor \( \alpha = \lg \sigma \ lg n \)
Research goal

★ We want to devise the compacted version of the succinct uncompacted dynamic trie of [JSS’07] without changing the time and space complexities of its operations

★ A sublinear time algorithm for sparse suffix tree construction

❖ To do this, we use
  • bit-parallel computation and
  • efficient predecessor dictionaries
Related work

- Faster dynamic uncompacted trie

- Predecessor dictionary for integers
  - D. Belazzougui, Boldi, and Vigna: Dynamic z-fast tries, SPIRE 2010. **Z-fast trie supports pattern matching for variable-length strings in** $O(P/\alpha)$ **average time.** But, it is not deterministic.
Our Results

- We devise the compacted version of the succinct dynamic trie of [JSS’07] that has the same time and space complexities as the original.
- As applications, we obtained the first sublinear-time sparse suffix tree construction algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Classic compact trie [Fredkin’60]</th>
<th>Succinct dyn. trie [JSS’07]</th>
<th>This</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trie type</td>
<td>compact</td>
<td>uncompacted</td>
<td>compact</td>
</tr>
<tr>
<td>Additional Space (bit)</td>
<td>$O(n \lg n)$</td>
<td>$O(n \lg \sigma)$</td>
<td>$O(n \lg \sigma)$</td>
</tr>
<tr>
<td>Time for operations</td>
<td>$O(P \lg \sigma)$</td>
<td>$O((P \lg \sigma)f(n)/\log(n))$</td>
<td>$O((P \lg \sigma)f(n)/\log(n))$</td>
</tr>
<tr>
<td>Online sparse suffix tree construction</td>
<td>Linear time</td>
<td>Quadratic time</td>
<td>Sub-linear time</td>
</tr>
</tbody>
</table>

$n$: Total text length, $\sigma$: Alphabet size, $P$: Pattern length, $f(n) = (\lg \lg n)^2/(\lg \lg \lg n)$, $\alpha = \lg \sigma$ $n$: speed-up factor
We obtained

\[ \Gamma = O(\frac{\lg n}{f(n)}) \]

\[ = O(\frac{\lg n \cdot (\lg \lg \lg n)}{(\lg \lg n)^2}) \]

times speed-up!!!!!
Algorithm
Basic idea: micro tree decomposition

1. We split the trie by every $w$-bit ($\alpha$-letter) length.
2. We attach a predecessor dictionary to each **micro tree** region containing at least one branching nodes (*)

(*) is necessary for obtaining linear words space bound.
Basic idea: micro tree decomposition

Pattern matching operation in our faster dynamic compact trie.

- **Predecessor dictionary**
- **O(P f(n)/\(\alpha\)) time**
- Use predecessor dictionary on branching subtrees
- Use bit-parallelism on non-branching paths
- **O(P lg(w)/\(\alpha\)) time**

Pattern matching operation in **O(P f(n)/\(\alpha\)) time**
Main result

- Assumption: \( D \) stores \( k \) \( O(lg \ n) \) bit integers in \( O(k \ lg \ n) \) bits, supporting predecessor and insert/delete in \( f(n) \) time.

**Theorem 1**: We can implement a data structure that stores \( k \) strings of total size \( n \) letters in Space \( O(n \ lg \sigma + klg \ n) \) bits supporting pattern matching and insert/delete in \( O(P \ f(n) / \alpha) \) time, where \( \alpha = \Theta(lg \sigma \ n) \) and \( |P| \) is pattern length.
Main result

- If we employ the dynamic data structure for $O(lg n)$-bit integers by Beame and Fich as auxiliary structure, which supports $f(n) = O((lg lg n)^2/lg lg lg n)$ predecessor and insertion/deletion operations, then we have the following results.

**Theorem 2**: The proposed dynamic compact trie stores a set $S$ of mutually distinct variable-length strings with total size $n$ letters in space $O(n lg \sigma)$ bits supporting pattern matching and insert/delete operation in $O((|P|/\alpha)((lg lg n)^2/lg lg lg n))$ time in the worst case, where $\alpha = \Theta(lg_\sigma n)$ and $|P|$ is pattern length.
## Applications

<table>
<thead>
<tr>
<th></th>
<th>Space (bit)</th>
<th>query time</th>
<th>update time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Online sparse suffix tree construction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Karkkainen &amp; Ukkonen</td>
<td>$O(n \lg \sigma)$</td>
<td>$O(P \lg \sigma)$</td>
<td>$O(n \lg \sigma)$</td>
</tr>
<tr>
<td><strong>Our results</strong></td>
<td>$O(n \lg \sigma)$</td>
<td>$O(P f(n) / \alpha)$</td>
<td>$O(n f(n) / \alpha)$</td>
</tr>
<tr>
<td><strong>Succinct substring index</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kolpakov, Kucherov, &amp; Starikovskaya</td>
<td>$O(n \lg \sigma)$</td>
<td>$O(P (\lg \sigma) f(n))$</td>
<td>$O(n \lg \sigma)$</td>
</tr>
<tr>
<td><strong>Our results</strong></td>
<td>$O(n \lg \sigma)$</td>
<td>$O(P (\lg \sigma) f(n))$</td>
<td>$O(n f(n) / \alpha)$</td>
</tr>
<tr>
<td><strong>Dynamic dictionary matching</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amir et al.</td>
<td>$O(n \lg n)$</td>
<td>$O((n \lg n + \text{occ})$ \ (\lg n / \lg \lg n))</td>
<td>$O(P \lg n / \lg \lg n)$</td>
</tr>
<tr>
<td>Hon et al.</td>
<td>$O(n \lg \sigma)$</td>
<td>$O(n \lg n + \text{occ})$</td>
<td>$O(P \lg \sigma + \lg n)$</td>
</tr>
<tr>
<td><strong>Our results</strong></td>
<td>$O(n \lg \sigma)$</td>
<td>$O(n (\lg \sigma) f(n))$</td>
<td>$O(P f(n) / \alpha)$</td>
</tr>
</tbody>
</table>

*n: Total text length, \(\sigma: \) Alphabet size, \(w: \) Register length, \(f(n): (\lg \lg n)^2/(\lg \lg \lg n), \quad P: \) Pattern length, \(\alpha = \lg n/\lg \sigma: \) speed-up factor*
Conclusion

- **Faster dynamic compact trie** using linear words space supports the pattern matching and insert/delete operations in:
  - \( O(n \frac{f(n)}{\alpha}) \) worst-case time, where \( f(n) = \frac{(\lg \lg n)^2}{\lg \lg \lg n} \) by using Beame and Fich.

- **Application**: We obtained a faster algorithm for
  - online sparse suffix tree construction.
  - succinct substring index.
  - dynamic dictionary matching.
  - We obtained \( \alpha/f(n) > 1 \) times speed-up!

**Future work**: Extension of dynamic z-fast trie for sparse suffix tree construction.

\( \sigma \): alphabet size, \( w \): register length, \( P \): pattern length, \( \alpha = \frac{w}{\lg \sigma} \): speed up factor
Thank you
Experimental Results

- input: All Japanese Wikipedia titles (30MB)
- query: sparse suffix tree construction

![Graph showing time to construction in seconds for different methods: DCT_B, DCT_L, and DCT_LH. The graph indicates that DCT_B takes the longest time, followed by DCT_L and DCT_LH.](image-url)
small case: branching subtree

**Step 1:** Compute the depth of the disagreement node ☆ by PRED and SUCC in $O(\sqrt{w} + \log \sigma)$ time

**Step 2:** We compute immediate branching ancestor ★ of ☆ by LCA in $O(\sqrt{w} + \log \sigma)$ time.

**Step 3:** Return the reference pointer (★, string(★, ☆))
**small case: branching subtree**

**Step 1:** Compute the depth of the disagreement node \(\star\) by PRED and SUCC in \(O(\sqrt{w} + \lg \sigma)\) time.

**Step 2:** We compute immediate branching ancestor \(\bullet\) of \(\star\) by LCA in \(O(\sqrt{w} + \lg \sigma)\) time.

**Step 3:** Return the reference pointer (\(\bullet\), string(\(\bullet\), \(\star\)))

CODE for STEP 1:
the bit depth \(c\) of \(\star = \max\{LCP(X, PRED(x)), LCP(X, SUCC(x))\}\)
small case: branching subtree

Step 1: Compute the depth of the disagreement node \( \star \) by PRED and SUCC in \( O(\sqrt{w} + \lg \sigma) \) time.

Step 2: We compute immediate branching ancestor \( \bullet \) of \( \star \) by LCA in \( O(\sqrt{w} + \lg \sigma) \) time.

Step 3: Return the reference pointer \((\bullet, \text{ string}(\bullet, \star))\)

CODE for STEP1:
the bit depth \( c \) of \( \star = \max \{ \text{LCP}(X, \text{PRED}(x)), \text{LCP}(X, \text{SUCC}(x)) \} \)

CODE for STEP2:
\[
\begin{align*}
a(L) &= \text{SUCC}(x[1..c]0^{\sqrt{w}-c}) \\
a(R) &= \text{PRED}(x[1...c]1^{\sqrt{w}-c}) \\
\bullet &= \text{lowest of} \\
& \text{LCA}(a(L-1), a(L)) \text{ and} \\
& \text{LCA}(a(R), a(R+1))
\end{align*}
\]