Speeding Up Compact Trie Structures on Word RAM and Its Applications

Takuya Takagi (Dept. E&E, Hokkaido University, Japan)
Takashi Uemura (Chowa Giken Co., 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 compact trie

- a classical data structure for a set of strings
  - a trie with path compression

\[ \Sigma = \{A, B, C, D\} \]

A set \( S \) of \( K \) strings

- \( s_1: \) AABAAACACAAAAA
- \( s_2: \) AABAACABAAAAA
- \( s_3: \) AABB
- ... 
- \( s_K: \) DDAB
The compact trie

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

- **Operations**
  - **General prefix search** (from arbitrary node): \( O(P \lg \sigma) \) time
  - **Insert (and delete)**: \( O(P \lg \sigma) \) time
  - **parent and child**: \( O(\lg \sigma) \) time

- **Applications**
  - (Sparse) suffix tree construction [Ukkonen’95]
  - Dynamic dictionary matching [Hon, Lam, et al. ’09]
The compact trie

- Linear space: Storing K strings
- Operations
  - General prefix search (from arbitrary node): $O(P \log \sigma)$ time
  - Insert (and delete): $O(P \log \sigma)$ time
  - parent and child: $O(\log \sigma)$ time
- Applications
  - (Sparse) suffix tree construction [Ukkonen’95]
  - Dynamic dictionary matching [Hon, Lam, et al. ’09]
Research goal

On **Word RAM**, we want to speed up the operations of compact tries:

- **prefix search** (from an arbitrary node) and **insert operations**

- To do this, we use
  - bit-parallel computation and
  - efficient predecessor dictionaries for a compact trie.
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 $\alpha = w/\log \sigma$ 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>$O((N + M)/w)^{\dagger}$</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>Compact trie operations**</td>
<td>$O(M , lg , \sigma)$</td>
<td>?</td>
<td>This work</td>
</tr>
<tr>
<td>Sparse suffix tree construction</td>
<td>$O(N)^{\dagger}$</td>
<td>?</td>
<td>This work</td>
</tr>
</tbody>
</table>

*Classic RAM supports comparison only, without supporting bit-wise or arithmetic operations.

$^{\dagger}$ for binary string. **General prefix search (from arbitrary node) and insert
Problems Accelerated on Word RAM

Open problems:

Speeding up compact trie operations and related problems using Word RAM.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Space 1</th>
<th>Space 2</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td>$O(N \log N)$</td>
<td>$O(N \log\log N)$</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, IWOC'A10] [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>Compact trie operations**</td>
<td>$O(M \log \sigma)$</td>
<td>?</td>
<td>This work</td>
</tr>
<tr>
<td>Sparse suffix tree construction</td>
<td>$O(N)$†</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. **General prefix search (from arbitrary node) and insert
Related work

- Predecessor dictionary for integers
  - D. E. Willard: Log-logarithmic worst-case range queries are possible in space $\Theta(n)$, IPL, 17, 1983. (y-fast tries.)
  - D. E. Willard: New trie data structure which support very fast search operations, JCSS, 28, 1984. (q-fast tries.)
  - D. Belazzougui, Boldi, and Vigna: Dynamic z-fast tries, SPIRE 2010. Z-fast trie supports prefix search for variable-length strings, but not general prefix search. Also, it is not deterministic.

- Packed string
Our result

- We propose speeding-up technique for a compact trie with general prefix search and insert operation on Word RAM.
- Key: augmentation of a branching of a trie by predecessor dictionary and LCA (lowest common ancestor information).
- As summary, we have the following results.

<table>
<thead>
<tr>
<th></th>
<th>Ordinary trie</th>
<th>Our result</th>
<th>Our result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base dictionary</td>
<td>None</td>
<td>Q-fast trie</td>
<td>dynamic Z-fast trie</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Willard ’84]</td>
<td>[Belazzougui’10]</td>
</tr>
<tr>
<td>Additional Space</td>
<td>O(K lg N) bit</td>
<td>O(K lg N) bit</td>
<td>O(K lg N) bit</td>
</tr>
<tr>
<td>Time for general prefix</td>
<td>O(P lg σ)</td>
<td>O(P√w/α )</td>
<td>O(P lg(w)/α )</td>
</tr>
<tr>
<td>search &amp; insert</td>
<td>in worst case</td>
<td>in worst case</td>
<td>on average</td>
</tr>
</tbody>
</table>

N: Total text length, K: # of strings, σ: Alphabet size, w: Register length, 
P: Pattern length, α = w/lg σ: speed-up factor
We propose speeding-up technique for a compact trie with general prefix search and insert operation on Word RAM.

Key: augmentation of a branching of a trie by predecessor dictionary and LCA (lowest common ancestor information).

As a result, we have the following results:

We obtained approx. $\alpha$ times speed-up!!!

- $O(\alpha / \sqrt{w})$ times in the worst case
- $O(\alpha / \lg(w))$ times on average

<table>
<thead>
<tr>
<th></th>
<th>Ordinary trie</th>
<th>Our result (Base dictionary)</th>
<th>Our result (Additional Space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base dictionary</td>
<td>None</td>
<td>Q-fast trie [Willard '84]</td>
<td>dynamic Z-fast trie [Belazzougui'10]</td>
</tr>
<tr>
<td>Additional Space</td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
</tr>
<tr>
<td>Time for general prefix search &amp; insert</td>
<td>$O(P \lg \sigma)$ in worst case</td>
<td>$O(P\sqrt{w}/\alpha)$ in worst case</td>
<td>$O(P \lg(w)/\alpha)$ on average</td>
</tr>
</tbody>
</table>

N: Total text length, K: # of strings, $\sigma$: Alphabet size, w: Register length, P: Pattern length, $\alpha = w/\lg \sigma$: speed-up factor
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.
PREFIX SEARCH

Part A) small case

inside a micro tree
of w-bit height
small case: branching subtree

We want to compute disagreement point ☆. In this case, the bit-parallel computation does not work due to many branching.

Thus, we use a predecessor dictionary associated to a micro trie to find the disagreement node.
Def. Dynamic predecessor dictionary

- Dictionary for a set $S$ of $N$ w-bit integers
- Using in $O(N)$ words space
- Supports following operations:
  - $\text{INSERT}(x)$: Insert a new element $x$ in $S$
  - $\text{PREDECESSOR}(x)$: Find the least element in $S$ not less than $x$.

- In the worst case, we use $q$-fast trie [Willard ’84] that supports above operations in $O(\sqrt{w})$ time.
- In the average case, we use $z$-fast trie [Belazzouguï, Boldi, Vigna, SPIRE’10] that supports above operations in $O(\log w)$ time.
small case: branching subtree

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

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

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

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

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

**Step 3:** Return the reference pointer (○, string(○, ☆))

**CODE for STEP 1:**
the bit depth $c$ of ☆ = $\text{MAX}\{\text{LCP}(X, \text{PRED}(x)), \text{LCP}(X, \text{SUCC}(x))\}$
small case: branching subtree

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

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

**Step 3:** Return the reference pointer

(⋄, string(⋄,☆))

**CODE for STEP 1:**
the bit depth c of ☆ =
\[
\max\{LCP(X, PRED(x)), LCP(X, SUCC(x))\}
\]

**CODE for STEP 2:**
\[
a(L) = SUCC(x[1..c]0^w-c);
a(R) = PRED(x[1...c]1^w-c);
\]
⋄ = lowest of
\[
LCA(a(L-1), a(L)) \text{ and } LCA(a(R), a(R+1))
\]
PREFIX SEARCH
Part B) large case
efficient prefix search on the whole compact trie
Large case: time complexity analysis

We can implement general prefix search in the large case by using prefix search in the small case.

- **On non-branching paths using bit-parallelism**: $O(P \lg(w) / \alpha)$ time
- **On branching subtrees using predecessor dictionary**: $O(P \sqrt{w} / \alpha)$ time

General prefix search in $O(P \sqrt{w} / \alpha)$ time.
Large case: space complexity analysis

- We assume the Q-fast trie.

- Analysis
  - Only the roots and leaves belong to two micro trees.
  - If each micro tree $S_i$ contains $K_i$ branching nodes, then $\Sigma_i K_i \leq 2K$.
  - For each micro tree $S_i$ with $K_i$ nodes, the corresponding Q-fast trie takes $O(K_i \log N)$ bits.
  - Total space $S = \Sigma_i O(K_i \log N) \leq O((\Sigma_i K_i) \log N) = O(K \log N)$ bits

- The whole data structure takes $O(K \log N)$ bits for the trie in addition to $O(N \log \sigma)$ bits for an input text.
Main result

- Assumption: $D$ stores $N$ $w$-bit integers in $s(w, N)$ bits, where, $s$ satisfies: $s(w, N1) + s(w, N2) \leq s(w, N1+N2)$ supporting predecessor and insert in $f(w, N)$ time.

**Theorem 1**: We can implement a data structure that stores $K$ strings of total size $N$ letters in Space $O(N \lg \sigma + K\lg N + s(w,N))$ bits supporting general prefix search in $O(P \cdot \frac{f(w,N)}{\alpha})$ time and insert in $O(P \cdot \frac{f(w,N)}{\alpha} + \lg \sigma)$ time.

$w$: register length, $P$: pattern length, $\alpha = \frac{w}{\lg \sigma}$.

- The above theorem gives a general technique that boosts any predecessor dictionary $D$ for integers to a data structure for storing variable-length strings.
Corollaries

- By substituting the following linear words space data structures for a predecessor dictionary D, we have:
  - **Q-fast trie** with $O(\sqrt{w})$ worst case time [Willard ’84]
  - **dynamic Z-fast trie** with $O(\lg(w))$ average case time [Belazzougui ’10]

<table>
<thead>
<tr>
<th></th>
<th>Ordinary trie</th>
<th>Our result</th>
<th>Our result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base dictionary</strong></td>
<td>None</td>
<td>Q-fast trie [Willard ’84]</td>
<td>dynamic Z-fast trie [Belazzougui’10]</td>
</tr>
<tr>
<td><strong>Additional Space</strong></td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
</tr>
<tr>
<td><strong>Time for general prefix search &amp; insert</strong></td>
<td>$O(P \lg \sigma)$ in worst case</td>
<td>$O(P\sqrt{w}/\alpha)$ in worst case</td>
<td>$O(P \lg(w)/\alpha)$ on average</td>
</tr>
</tbody>
</table>

N: Total text length, K: # of strings, $\sigma$: Alphabet size, w: Register length, P: Pattern length, $\alpha = w/\lg \sigma$: speed-up factor
Corollaries

- By substituting the following linear words space data structure:
  - Q-fast trie [Willard ’84]
  - dynamic Z-fast trie [Belazzougui’10]

We obtained
approx. $\alpha$ times speed-up!!!

$O(\alpha/\sqrt{w})$ times in the worst case
$O(\alpha/\lg(w))$ times on average

<table>
<thead>
<tr>
<th></th>
<th>Ordinary trie</th>
<th>Our result</th>
<th>Our result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base dictionary</td>
<td>None</td>
<td>Q-fast trie</td>
<td>dynamic Z-fast trie</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Willard ’84]</td>
<td>[Belazzougui’10]</td>
</tr>
<tr>
<td>Additional Space</td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
<td>$O(K \lg N)$ bit</td>
</tr>
<tr>
<td>Time for general prefix search &amp; insert</td>
<td>$O(P \lg \sigma)$ in worst case</td>
<td>$O(P\sqrt{w}/\alpha)$ in worst case</td>
<td>$O(P \lg(w)/\alpha)$ on average</td>
</tr>
</tbody>
</table>

N: Total text length, K: # of strings, $\sigma$: Alphabet size, w: Register length, P: Pattern length, $\alpha = w/\lg \sigma$: speed-up factor
An Application

Faster sparse suffix tree construction

- The **sparse suffix tree** on a binary prefix code $\Delta$ is a compact trie for a subset of $K$ suffixes of an input text $T$ with total length $N$ bits using $O(K)$ words, where each suffix starts at a code boundary.

- The **previous online construction algorithm** requires $O(N)$ time using $O(\delta)$ preprocessing and $O(K+\delta)$ word space [Uemura, Arimura, CPM’11].

- Using our speed-up technique, we have:

Theorem 2 (This work): For finite prefix code, the modified algorithm constructs a sparse suffix tree online in $O(N/\sqrt{w} + K/\sqrt{w})$ time using $O(\delta)$ preprocessing and $O(K+\delta)$ word space.

$$\delta = ||\Delta||: \text{the size of a code, } w: \text{register length}$$
An Application

Faster sparse suffix tree construction

- The **sparse suffix tree** on a binary prefix code $\Delta$ is a compact trie for a suffix array constructed using $O(K\log \frac{N}{K})$ time.
- The sparse suffix tree can be computed using $O(K)$ words of memory.
- Using the sparse suffix tree, we have speed-up of $O(\sqrt{w})$ times in the worst case
  and $O(w/\log(w))$ times on average

**Theorem 2 (This work):** For finite prefix code, the modified algorithm constructs a sparse suffix tree online in $O(N/\sqrt{w} + K\sqrt{w})$ time using $O(\delta)$ preprocessing and $O(K + \delta)$ word space.

$$\delta = ||\Delta||: \text{the size of a code, } w: \text{register length}$$
Conclusion

A technique for boosting any predecessor dictionary for integers to a compact trie data structure for variable-length strings combined with bit-parallelism on Word RAM.

Faster compact trie using linear words space that supports the general prefix search and insert operations in:

✓ $O(P \sqrt{w/\alpha})$ worst-case time using Q-fast trie.
✓ $O(P \lg(w)/\alpha)$ average-case time using dyn. Z-fast trie.

Application: a faster algorithm for online sparse suffix tree construction.

Future work: (1) Application to dynamic dictionary matching. (2) Extension of dynamic z-fast trie for directly supporting general prefix search to obtain the same result.

$\sigma$: alphabet size, $w$: register length, $P$: pattern length, $\alpha = w/\lg \sigma$: speed up factor.
Thank you