c-trie++: A Dynamic Trie Tailored for Fast Prefix Searches

Kazuya Tsuruta¹, Dominik Köppl¹,², Shunsuke Kanda³, Yuto Nakashima¹, Shunsuke Inenaga¹, Hideo Bannai¹, Masayuki Takeda¹

1. Department of Informatics, Kyushu University, Fukuoka, Japan.
2. Japan Society for Promotion of Science.
3. RIKEN Center for Advanced Intelligence Project, Tokyo, Japan.
Problem Definition (1/4)

【Problem】 Dynamic Prefix Search

Maintain a data structure for a dynamic set \( S = \{T_1, T_2, T_3, T_4, T_5\} \) of strings that, given a query pattern \( P \), can compute the pair

a) \( \max \{l : P[1..l] = T_i[1..l] \text{ for some } i \in [1..k]\} \) and

b) \( I_P = \{i : T_i[1..l] = P[1..l]\} \) efficiently.

Example:

\( S = \{T_1, T_2, T_3, T_4, T_5\} \)

\( T_1 = \text{idea} \)

\( T_2 = \text{interface} \)

\( T_3 = \text{internet} \)

\( T_4 = \text{infinite} \)

\( T_5 = \text{laboratory} \)

\( P = \text{inter} \)

output = (5, \{2, 3\})
Problem Definition (2/4)

【Problem】 Dynamic Prefix Search

Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern $P$, can compute the pair

a) $\max\{l : P[1..l] = T_i[1..l] \text{ for some } i \in [1..k]\}$ and

b) $I_P = \{i : T_i[1..l] = P[1..l]\}$ efficiently.

Example:

$S = \{T_1, T_2, T_3, T_4, T_5\}$

$T_1 = \text{idea}$

$T_2 = \text{interface}$

$T_3 = \text{internet}$

$T_4 = \text{infinite}$

$T_5 = \text{laboratory}$

$P = \text{inner}$

output = $(2, \{2, 3, 4\})$
Problem Definition (3/4)

【Problem】 Dynamic Prefix Search
Maintain a data structure for a dynamic set \( S = \{T_1..T_k\} \) of strings that, given a query pattern \( P \), can compute the pair
a) \( \max \{ l : P[1..l] = T_i[1..l] \text{ for some } i \in [1..k] \} \) and
b) \( I_P = \{ i : T_i[1..l] = P[1..l] \} \)
efficiently.

Example:
\( S = \{T_1, T_2, T_3, T_4, T_5, T_6\} \)
\( T_1 = \text{idea} \)
\( T_2 = \text{interface} \)
\( T_3 = \text{internet} \)
\( T_4 = \text{infinite} \)
\( T_5 = \text{laboratory} \)
\( T_6 = \text{indexing} \)

Supports insertion of a string into \( S \).

\( \text{insert}(T_6) \)
【Problem】 Dynamic Prefix Search
Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern $P$, can compute the pair
a) $\max\{l : P[1..l ] = T_i[1..l ] \text{ for some } i \in [1..k]\}$ and
b) $I_P = \{i : T_i[1..l ] = P[1..l ]\}$ efficiently.

Example:
$S = \{T_1, T_2, T_3, T_4, T_5, T_6\}$
$T_1 = \text{idea}$
$T_2 = \text{interface}$
$T_3 = \text{internet}$
$T_4 = \text{infinite}$
$\textbf{T}_5 = \text{laboratory}$
$T_6 = \text{indexing}$

Supports deletion of a string from $S$.

\text{delete}(T_5)$
prefix search applications
- input method editors
- query auto-completion
- range query filtering

(entering japoni on an Android phone)
Introduction (2/3)

- prefix search applications
  - input method editors
  - query auto-completion
  - range query filtering

(entering japani on google)
Introduction (3/3)

- prefix search applications
  - input method editors
  - query auto-completion
  - range query filtering

```
SELECT User, Word
FROM UserQueries
WHERE Word LIKE 'japani%'
AND ...
```

<table>
<thead>
<tr>
<th>User</th>
<th>Date</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>661</td>
<td>12/03/2020</td>
<td>refund-ticket</td>
</tr>
<tr>
<td>457</td>
<td>11/03/2020</td>
<td>trip-cancellation</td>
</tr>
<tr>
<td>139</td>
<td>01/03/2020</td>
<td>corona-virus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User</th>
<th>Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>japanican</td>
</tr>
<tr>
<td>83</td>
<td>japanification</td>
</tr>
<tr>
<td>89</td>
<td>japanistry</td>
</tr>
<tr>
<td>97</td>
<td>japani</td>
</tr>
</tbody>
</table>
Compact Trie [Morrison, 1968]

- **Trie**: represents strings where common prefixes are compressed to a single path (front encoding).
- **Compact Trie**: reduces the number of nodes by replacing branchless path segments with a single edge.

### Strings

- $T_1 = \text{ababaabcbb}$
- $T_2 = \text{ababbbcb}$
- $T_3 = \text{ababcbbab}$
- $T_4 = \text{bbcbaaba}$
- $T_5 = \text{bbcbaabcbb}$
- $T_6 = \text{bcbabccbbba}$
- $T_7 = \text{bcbabccbbbb}$
<table>
<thead>
<tr>
<th>Previous Works for Dynamic Prefix Search (1/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space [bits]</strong></td>
</tr>
<tr>
<td>Trie</td>
</tr>
<tr>
<td>Compact Trie [Morrison, 1968]</td>
</tr>
</tbody>
</table>

$|T|$ : trie size (front encoding size)

$k = |S|$, $m = |P|$, $\sigma = |\Sigma|$, $\Sigma$ : alphabet, $occ$ : number of occurrences
### Previous Works for Dynamic Prefix Search (2/2)

<table>
<thead>
<tr>
<th></th>
<th>Space [bits]</th>
<th>Prefix Search Expected Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trie</td>
<td>(O(</td>
<td>T</td>
</tr>
<tr>
<td>Compact Trie [Morrison, 1968]</td>
<td>(</td>
<td>T</td>
</tr>
<tr>
<td>Z-Fast Trie [Belazzougui et al., 2010]</td>
<td>(</td>
<td>T</td>
</tr>
<tr>
<td>Packed C-Trie [Takagi et al., 2017]</td>
<td>(</td>
<td>T</td>
</tr>
</tbody>
</table>

\(|T|\): trie size (front encoding size)

\(n = \sum_i |T_i|, k = |S|, m = |P|, \sigma = |\Sigma|, \Sigma: \) alphabet, \(occ: \) number of occurrences

\(w: \) machine word size \((w = \Omega(\log n))\), \(\alpha = O(w / \log \sigma)\)
## Our Contribution for Dynamic Prefix Search

<table>
<thead>
<tr>
<th></th>
<th>Space [bits]</th>
<th>Prefix Search Expected Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trie</td>
<td>$O(</td>
<td>T</td>
</tr>
<tr>
<td>Compact Trie [Morrison, 1968]</td>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>Z-Fast Trie [Belazzougui et al., 2010]</td>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>Packed C-Trie [Takagi et al., 2017]</td>
<td>$</td>
<td>T</td>
</tr>
<tr>
<td>C-Trie++ [Ours]</td>
<td>$</td>
<td>T</td>
</tr>
</tbody>
</table>

$|T|$: trie size (front encoding size)

$n = \sum_i |T_i|, k = |S|, m = |P|, \sigma = |\Sigma|, \Sigma$: alphabet, $occ$: number of occurrences

$w$: machine word size ($w = \Omega(\log n)$), $\alpha = O(w / \log \sigma)$

$\alpha < w$ always holds.
Word Packing

In the word RAM model with word size $w$ bits, we can compare ($=, <, >$) two $O(w)$ bits integers in $O(1)$ time.

⇒ Let $\alpha = w / \log \sigma$.

$$
\begin{array}{cccc}
  i & d & e & a \\
  01101001 & 01100100 & 01100101 & 01100001 \\
\end{array}
$$

a character uses $\lceil \log \sigma \rceil$ bits
⇒ strings of length $\alpha$ use $O(w)$ bits

We can compare two strings of length $\alpha$ in $O(1)$ time.

⇒ We can compare two strings of length $m$ in $O(m / \alpha)$ time.

64-bits architecture
\[ \sigma = 256 \]
\[ w = 32 \]
\[ \alpha = 4 \]
Compact Trie

dissect at word boundaries
Introduction of Micro Trie (2/4)

Compact Trie

Make boundary node
Introduction of Micro Trie (3/4)

C-Trie++

Make boundary node

Micro Tries
Equip each Micro Trie with a Hash Table

Hash Table:
- jump from root to leaf in $O(1)$ expected time.
- key: string of length $\alpha$
- value: leaf node
Trie Traversal (1/5)

C-Trie++

Traverse from root

Try search P
If key exists, we can traverse in $O(1)$ time with hash table.
We can compare $\alpha$ characters in $O(1)$ time with word packing.
We can compare $\alpha$ characters in $O(1)$ time with word packing.
We perform prefix search in last micro trie.
Dynamic Prefix Search in Micro Tries

- We improve prefix search (expected) time in micro trie
  - Belazzougui et al., 2010: $O(\log (m \log \sigma) + \text{occ})$
  - Takagi et al., 2017: $O(\log \omega + \text{occ})$
  - This work: $O(\log \min\{\alpha, m\} + \text{occ})$

Z-Fast Trie
(binary tree)
[Belazzougui et al., 2010]

Alphabet Aware
Z-Fast Trie
(σ-ary tree)
[This work]
Prefix Search Time

C-Trie++

total: $O(m / \alpha + \log \min\{\alpha, m\} + \text{occ})$ expected time

$O(m / \alpha)$ expected time

$O(\log \min\{\alpha, m\} + \text{occ})$ expected time
Insertion Time

C-Trie++

\[
total : O\left(\frac{m}{\alpha} + \log \min\{\alpha, m\}\right) \text{ expected time}
\]
Deletion Time

C-Trie++

total: $O(m / \alpha + \log \min\{\alpha, m\})$ expected time

$O(m / \alpha)$ expected time

$O(\log \min\{\alpha, m\})$ expected time
Experimental Setup

- CPU: Intel Xeon X5560 @2.80 GHz
- Memory: 198GB
- OS: CentOS 6.10
- Language: C++

Implementations
- Compact Trie [Takagi et al.]
- Z-Fast Trie [Ours]
- Packed C-Trie [Takagi et al.]
- C-Trie++ [Ours]
Datasets

Characteristics

<table>
<thead>
<tr>
<th>Datasets</th>
<th>size[MB]</th>
<th>$\sigma$</th>
<th>$k[10^3]$</th>
<th>average length</th>
<th>avg. LCP</th>
<th>C-Tries nodes$[10^3]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>proteins</td>
<td>864.14</td>
<td>26</td>
<td>2,982</td>
<td>302.8</td>
<td>38.8</td>
<td>5,778</td>
</tr>
<tr>
<td>dblp.xml</td>
<td>164.89</td>
<td>96</td>
<td>2,950</td>
<td>57.6</td>
<td>34.4</td>
<td>5,899</td>
</tr>
</tbody>
</table>

We split a data set into strings at delimiters such as carriage returns, which form our input set $S$.

In our experiment for prefix search, we took the prefixes of length 10%, 20%, …, 100% of the strings of $S$ as patterns, and measured the average query time.
Experimental Results

Prefix Search Time [microsec]

- **proteins**
- **dblp.xml**

- **Compact Trie**
- **Packed C-Trie**
- **Z-Fast Trie**
- **C-Trie++**

Percentages of pattern length:

- 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Experimental Results

- **Memory Usage [MB]**

<table>
<thead>
<tr>
<th></th>
<th>proteins</th>
<th>dblp.xml</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact Trie</td>
<td>10^4</td>
<td>10^3</td>
</tr>
<tr>
<td>Packed C-Trie</td>
<td>10^3</td>
<td>10^2</td>
</tr>
<tr>
<td>Z-Fast Trie</td>
<td>10^2</td>
<td>10^2</td>
</tr>
<tr>
<td>C-Trie++</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>
Conclusions

**Summary**

- We proposed c-trie++:
  - Space: $|T| \log \sigma + \Theta(kw)$ bits.
  - Prefix Search: $O(m / \alpha + \log \min\{\alpha, m\} + \text{occ})$ time.
  - Insert: $O(m / \alpha + \log \min\{\alpha, m\})$ time.
  - Delete: $O(m / \alpha + \log \min\{\alpha, m\})$ time.
- Our computational experiments support the claim that c-trie++ is the fastest trie for prefix search.

**Future Work**

- Use SIMD instruction sets that allow larger machine word sizes (here $w = 64$ bits).