Proposal of Search Method Compressed the One-Way Branch based on the Double-Array Structure

YASUMASA NAKAMURA  HISATOSHI MOCHIZUKI
Osaka Kyoiku University
Information Science
Asahigaoka 689-1 Kashiwara-shi Osaka 582-8582
JAPAN

Abstract: Digital search is expressed by the digital search tree which considers a part of keys as transitions. Therefore, it is frequently used for various applications, such as natural language dictionaries because of being dependent on the key length for search processing. There is the compressed digital search tree which compressed the one-way branches after the leaf and reduces nodes on digital search tree. However the one-way branches has still existed on the compressed digital search tree. In this paper, we present the search method which used the digital search tree compressed all one-way branches used the double-array structure. The simulation results confirmed that the proposed method is more efficient than the original method.

Key–Words: Information Retrieval, Digital Search, the Double-Array, Patricia, Data Structure, Trie

1 Introduction

Search processings such as hash based on comparison between keys are dependent on the number of keys. On the other hand, digital search comparisons between a part of keys[2]. Therefore, it is dependent on the key length and frequently used for various applications, such as natural language dictionaries, database systems and compilers.

Digital search is expressed by the digital search tree (DS-tree) which considers a part of keys as transitions. DS-trees are divides into three types, which are full DS-tree (FDS-tree), compressed DS-tree (CDS-tree) and Patricia. FDS-tree has all parts of keys. On the other hand, CDS-tree and Patricia have some parts of keys. CDS-tree expresses the common prefix. Patricia does not have nodes which have only one child (one-way branch) [1, 2]. DS-tree often expressed as the binary tree, actually Patricia is the binary tree. Because of the binary tree is easier to realize than the multi-way tree and its space efficiency are better than the multi-way tree.

However, the binary tree has many nodes and its search speed is slower than multi-way tree. There is the double-array structure as an effective data structure which realizes multi-way tree. It has quick search speed and high space efficiency. At the present, a variety of the high speed updating techniques are proposed to the double-array[3, 4].

The present work is intended to reduce the one-way branches of multi-way tree used the double-array. Therefore, the search technique and the updating techniques of having introduced the node which stored transition informations are proposed.

The rest of the paper is organized as follows. In section 2, we give overviews of the double-array structure. In section 3, we propose a search method compressed the one-way branches used the double-array structure. In section 4, we evaluation our proposed methods. Finally, we conclude the paper and describe future works in section 5.

2 The Double-Array Structure

The double-array is a data structure which realized FDS-tree and CDS-tree. In this section, we explain the double-array structure taking the case of key set $K$. Where numerical values of ‘a’, ‘b’, . . . , ‘o’ are 1, 2, . . . , 15 respectively and they are translated into binary numbers of 4 bits. $K = \{ \text{academe, academic, cable, cache, call} \}$

The double-array uses one-dimensional arrays BASE and CHECK[5]. Their index numbers correspond to node numbers in DS-tree. The BASE value of the node on DS-tree is the basis value which is the index number of the child. The CHECK value of the node on DS-tree is the index number of the parent.

The concept of the double-array is to represent transition by giving Eq.1 and Eq.2. Where $B[x]$ and $C[x]$ are the values of element $x$ of BASE and
CHECK respectively, \( s \) is the original node, \( t \) is the child of \( s \) and \( a \) is the numerical value of transition. The transition by the double-array is shown in Fig.1.

\[
B[s] + a = t \quad (1) \\
C[t] = s \quad (2)
\]

For example, CDS-tree and the double-array for \( K \) are shown in Fig.2. As to a transition from original node 10 by ‘b’, a child of 10 is \( B[10] + b = 2 + 2 = 4 \) used Eq.1. After that, \( C[4] = 10 \) is compared with an original node 10 used Eq.2, in this case both are equal. Thus, a transition from 10 to 4 by ‘b’ is success.

Moreover, a doubly list called E-Link is created with the elements of the double-array called the empty element[5, 3, 4]. The empty element is not used as a node of DS-tree. E-Link is given by the BASE and CHECK values of the empty element. In the example of the double-array, for simplicity, the values of the empty elements are made blank.

A one-dimensional array TAIL is used to realize CDS-tree[5]. The double-array expresses the common prefix, TAIL expresses the one-way branches after the common prefix called the suffix. Where \( T[x] \) is the value of element \( x \) of TAIL and \( T + x \) is the suffix which begin from \( T[x] \).

The value of BASE as to the leaf on CDS-tree is negative value, and considered as the link of the suffix on TAIL by giving Eq.3. Where leaf is the node number of leaf on CDS-tree and \( loc \) is index number on TAIL of the suffix. For example, as to a suffix of a leaf 4 is \( T + (-B[4]) = T + 4 = "le" \) used Eq.3.

\[
B[leaf] = -loc \quad (3)
\]

### 3 An Approach Compressed the One-Way Branches

In this section, we propose the search methods focused on the one-way branches which exist on CDS-tree. It is important to mention here that the node which stored transitions information is defined.

#### 3.1 Data Structure

As to the one-way branches \( a_{i+1}, a_{i+2}, \ldots, a_{i+n} \) of length \( n \), the parent of \( a_{i+1} \), transition to the child of \( a_{i+n} \), transition are unique identifiable. To reduce the node on FDS-tree, the suffixes on the double-array are replaced on TAIL.

However, CDS-tree has the one-way branches called the prefix. Therefore, the prefixes on the double-array are also replaced on TAIL. The node called the prefix node is defined to manage the prefix. Then, the definition of the prefix node is shown in
Where Abs\( (\text{parent}) \) which has the prefix node is negative value.

It is explained. In (S-1), store 0 in \( \text{pos} \) which is the position of the \( \text{key} \) and store 1 in \( s \) which is the parent.

For example, the double-array used the prefix node and SIBLING for \( K \) is shown in Fig.6. Where numerical values of PCODE is 0. A next sibling of a node 11 is Abs\( (S[11]) = 13 \), and a next sibling is Abs\( (S[13]) = 11 \). In addition, a node 11 has a prefix node because of \( S[11] \) is the negative value. A prefix node of \( 11 \) is \( B[11] + \text{PCODE} = 3 \). A prefix node 3 shows a prefix \( T + (-B[3]) = \) “cadem” of length \( -C[3] = 5 \).

Patricia for \( K \) is shown in Fig.4(b). Patricia is the binary tree and two nodes are needed to distinguish “cable”, “cache” and “call”. However, DS-tree used the prefix node needs only one node.

### 3.2 A Searching Algorithm

A searching algorithm is presented in Fig.7. There are three conditions in which search succeed. First, transition from a root to a leaf on DS-tree by satisfied Eq.1 and Eq.2 in (S-2) and (S-3). Unlike the double-array, the transitions about the prefixes are not checked. Second, all prefixes which prefix nodes shows are equal key in (S-4). It is the point that this also differs from the double-array. Last, the suffix which leaf shows is equal key in (S-5).

\[ B[p] = -\text{loc}, C[p] = -n \]  
(4)

The transition to the prefix node is the constant PCODE. If two or more parents with an equal BASE value exist, each parents do not have the prefix node or one of them has the prefix node. In the case of the later, it can not judge which the prefix node has the prefix node, because of the CHECK value is not the index number of its parent.

Therefore, a one-dimensional array SIBLING is defined. SIBLING manages the siblings which are the other children of its parent, in order to fast computation of the children searching in the updating processing. The SIBLING value of the child is other child of its parent, and create the circular list which is shown in Fig.5. In this paper, this list called S-Link.

Then, the SIBLING value is given by Eq.5. Where \( S[x] \) is the values of element \( x \) of SIBLING, the default value of \( S[x] \) is \( x \), \( b \) is the number of the children of the arbitrary parent \( s \) and \( t_1, t_2, \ldots, t_b \) are index numbers of the children of \( s \).

\[ S[t_i] = t_{i+1} \quad (1 \leq i \leq b-1), \quad S[t_b] = t_1 \]  
(5)

In addition, SIBLING manages whether the parent has the prefix node. The SIBLING value of the parent which has the prefix node is negative value. Where Abs\( (v) \) is a function which return to the absolute value of \( v \).
store 11 in $s$, $0 + 1$ in $pos$.

In (S-2), 11 has a prefix node 3 which shows a prefix of length $-C[3] = 5$. Then, enqueue $\{1, 3\}$ and store $1 + 5 = 6$ in $pos$. In (S-3), transit from $s = 11$ to $t = B[11] + key[6] = 12$ by $key[6] = i'$ and store 12 in $s$, $6 + 1 = 7$ in $pos$. After that, proceed to (S-4) because of $B[12] = 7$ is the negative value.

In (S-4), as to $\{1, 3\}$, compare $key + 1 = \text{"cadem"}$ and $T + (-B[3]) = \text{"cadem"}$. In this case both are equal and queue is empty, then proceed to (S-5). In (S-5), compare $key + 7 = \text{"c"}$ and $T + (B[12]) = \text{"c"}$, in this case both are equal. Thus, searching “academic” is success.

### 3.3 Insertion Algorithms

The insert processing is performed when the search processing is fail.

A case of fail on DS-tree in (S-3), call InsLeaf($s$, $t$, key, pos) to create a leaf for key[3]. This function, if a new leaf $t$ which is a child of $s$ by key[pos] is an empty element, create a leaf $s$, then proceed to (S-2). Otherwise, update $B[s]$ or $B[s']$ and create a leaf on $B[s] + key[pos][5]$. A case of fail on the TAIL in (S-4) and (S-5), call InsPrefixNode(t, key, pos, $T + (-B[t])$, n) to update the prefix or the suffix. Where $n$ is the minimum comparison position where key + pos differs from $T + (-B[t])$.

The insert processing utilizes following variables and functions.

**tailPos**: A global variable which indicates the minimum index of available entries of TAIL.

**L**: The set of transitions.

**NewBase(L)**: Return to a BASE value to which all element of $L$ can transit used E-Link.

**InsNode(x)**: $x$ is deleted from E-Link in order to $s$ is made usable on DS-tree. And, update S-Link.

**DelNode(x)**: $x$ is inserted to E-Link in order to $s$ is deleted from DS-tree. And, update S-Link.

**CreateLeaf(s, t, suffix)**: Call InsNode(t) and store $s$ in $C[t]$. Store $-tailPos$ in $B[t]$ and a suffix suffix is stored from $T[tailPos]$.

**RenewalCheck(new, old)**: As to the children of new except a prefix node, the CHECK value old is updated to new used S-Link.

**InsPrefixNode** is presented in Fig.8. This function is called the case of updating prefix or suffix.

In the first case, determine the BASE value to which \{key[pos + n], tail[n]\}, PCODE can transit, and create a prefix node $p$ which shows prefix $tail$ of length $n$. Then create a leaf $t$ of $s$ by key[pos + n] and a child $t'$ of $s$ by tail[n]. And update a prefix $pp$ of $t'$ which shows prefix $tail + n + 1$.

**Function**: InsPrefixNode($s$, key, pos, tail, n)

**Step(P-1)**: Initialization

Store $B[s]$ in base which is a old BASE value of $s$. Set L to \{key[pos + n], tail[n]\}.

**Step(P-2)**: Definition of a prefix node of $s$

If $n$ is equal to 0, store NewBase(L) in $B[s]$ and proceed (P-3). Otherwise, store NewBase(L ∪ \{PCODE\}) in $B[s]$ and $B[s] + PCODE$ in $p$. Call InsNode($p$) and $S[s]$ is set as a negative value. A prefix tail of length $n$ is stored from $T[tailPos]$ and store $-tailPos$ in $B[p]$, $-n$ in $C[p]$.

**Step(P-3)**: Definition of a node about tail

Store $B[s] + tail[n]$ in $t$. Call InsNode($t$) and store $s$ in $C[t']$. If base is the negative value, proceed to (P-4), otherwise proceed to (P-5).

**Step(P-4)**: Updating of the $B[t'](t' > 0)$ is a leaf

Store base − $n$ in $B[t']$. If $n$ is more than 0, store base in $B[p]$. Proceed to (P-6).

**Step(P-5)**: Updating of the $B[t'](t' > 0)$ is not a leaf

Store base in $B[t']$ and call RenewalCheck($t'$, $s$). Store base + PCODE in $pp$ which a prefix node of $t'$. If $n$ is not equal to 0, store $B[pp]$ in $B[p]$ and $-n$ in $C[p]$. If $n + 1$ is equal to $-C[pp]$, call DelNode($pp$) and $S[t']$ is set as a positive value, otherwise advance $B[pp]$ by $(n + 1)$ and $C[pp]$ by $n + 1$.

**Step(P-6)**: Definition of a leaf about key

Store $B[s] + key[pos + n]$ in $t$ and call CreateLeaf($s$, $t$, key + (pos + n + 1)).

**Fig. 8**: Algorithm of InsPrefixNode

In the second case, if $n$ is equal to 0, create a leaf $t$ of $s$ by key[pos]. As to tail, create a child $t'$ of $s$ by tail[0] and update $B[t']$ to updating tail. If $n$ is not equal to 0, need to create a prefix node $p$.

For example, insertion “account” in Fig.6 is explained. Searching “account” is fail on the TAIL and call InsPrefixNode(11, “account”, 1, “cadem”, 1). InsPrefixNode processes in the following order, (P-1), (P-2), (P-3), (P-5), (P-6). In (P-2), call NewBase(“’c’, ‘a’, PCODE”) and create a prefix node 6, and update a prefix node 3 in (P-5). In (P-6), create a leaf which shows “count” on 9. DS-tree and the double-array after insertion processing are shown in Fig.9.

### 3.4 A Deletion Algorithm

The delete processing is performed when the search processing is success. Delete is presented in Fig.10. This function consists of deletion of key and maintaining a form of DS-tree. The first processing is (D-2), the second processing is (D-3) to (D-7). The deletion processing utilizes following functions.

**leaf**: A leaf which shows a delete key.
In (D-2), delete a leaf 9. In (D-3), set “ca” to tail. Searching “account” is success on a leaf 9.

In connection with the BASE value of 9, the SIBLING value of 9 is updated oldBase to newBase, all children of 9 are moved. If the child of 9 has children, call RenewalCheck.

For example, deletion “account” in Fig.9 is explained. Searching “account” is success on a leaf 9 and call Delete(9). In Delete, it processes in the following order, (D-1), (D-2), (D-3), (D-5), (D-6), (D-7). In (D-2), delete a leaf 9. In (D-3), set “ca” to tail and delete a prefix node 6. In (D-5), tail is updated “cadem”, and update a prefix shown a prefix 3 to “cadem” in (D-6). In (D-7), delete a node 7. DS-tree and the double-array after deletion processing are return to Fig.6 and Fig.4(a) respectively.

**4 Experimental Observations**

We conducted simulation experiments to evaluate the proposed methods. Where DA, DA+T and DA+TS methods are the double-array and DA method does not use TAIL[5], DA+T and DA+TS methods used TAIL[5], DA+TS method used SIBLING. In addition, P method is Patricia[1]. In this simulation, we used two set of key which are randomly selected 200,000 English words called $S_1^p$ and 5,000,000 URI called $S_1^a$. The average lengths of key are 10.47 and 58.52 bytes respectively.

Simulation results of the searching processing are shown in Tab.1. Tab.1 shows that the average of searching times on DS-tree and on TAIL. In addition, shows the average of search lengths on DS-tree and on TAIL.

From the simulation results, the searching speed of the proposed method is faster than the original methods for $S_1^p$, about 1.16 to 1.32 times. On the other hand, later than DA+T method for $S_1^a$, about 0.78 to 0.90 times. Notice that the averages of search length on DS-tree are reduced by the proposed method for both set. However, the searching speed on DS-tree of proposed method is later than DA and DA+T methods. This indicate that a processing of enqueue in (S-2) is taken time.

As to the updating processing, the initial states
are set to 20,000 words for $S_1$ and 500,000 URI for $S_1$, at random, and insertion and deletion processing used $S_2$ and $S_2\alpha$. Where $S_2\epsilon$ and $S_2\alpha$ are randomly selected 200,000 words from $S_1\epsilon$ and 5,000,000 URI from $S_1\alpha$, respectively. The number of insertion keys and deletion keys are almost equal and the number of keys on search methods are almost equal during the updating processing.

This simulation results are shown in Tab.2. Tab.2 shows that the average of insertion and deletion times. In addition, shows the average of search length for the children searches which are GetLabels, RenewalCheck and the judgment whether parent has only one child.

Moreover, shows that the number of nodes on DS-tree, used elements on TAIL and the used space. In addition, as to the proposed method for $S_2\epsilon$ and $S_2\alpha$, the number of the prefix nodes are 2,592 and 172,708, the average of prefix are 2.01 and 8.02 respectively. Where an used space of the P method is computed from a position of key, two information of child, an index on TAIL of key and TAIL.

From the simulation result, the insertion speed of the proposed method is faster than the original methods for $S_2\alpha$, about 1.28 to 2.64 times. And for $S_2\epsilon$, equivalent to the insertion speeds of the original methods. This indicates that the inserted number of nodes on DS-tree is reduced for the prefix node, about 0.94 to 0.28 times. Especially for $S_2\alpha$, the proposed method is smaller than the original method, about 0.10 to 0.76 times. This indicates that the number of nodes on DS-tree is reduced for the prefix node.

Simulation results prove that the proposed method is more efficient.

## 5 Conclusion

This paper proposed the search method which used the digital search tree compressed all one-way branches used the double-array structure, and the searching and updating algorithms are also shown. In addition, the space and time efficiency of the proposed method is shown from the simulation result.

As a future research work, we plan to implement the proposed method on an actual system to verify practicality and evaluate applicability.

### References:


---

**Tab. 1: The simulation results of search processing.**

<table>
<thead>
<tr>
<th></th>
<th>searching time(µs)</th>
<th>search length(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>On DS-tree On TAIL</td>
<td>Total</td>
</tr>
<tr>
<td>English</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>0.77</td>
<td>0.00</td>
</tr>
<tr>
<td>DA+T</td>
<td>0.54</td>
<td>0.60</td>
</tr>
<tr>
<td>P</td>
<td>1.29</td>
<td>1.44</td>
</tr>
<tr>
<td>Proposed</td>
<td>0.65</td>
<td>0.76</td>
</tr>
<tr>
<td>URI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>4.03</td>
<td>4.03</td>
</tr>
<tr>
<td>DA+T</td>
<td>3.65</td>
<td>3.83</td>
</tr>
<tr>
<td>P</td>
<td>3.87</td>
<td>4.36</td>
</tr>
<tr>
<td>Proposed</td>
<td>2.44</td>
<td>3.73</td>
</tr>
</tbody>
</table>

**Tab. 2: The simulation results of updating processing.**

<table>
<thead>
<tr>
<th></th>
<th>insertion deletion search length(µs)</th>
<th>search length(µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time(µs)</td>
<td>time(µs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>English</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>7.60</td>
<td>9.60</td>
</tr>
<tr>
<td>DA+T</td>
<td>4.48</td>
<td>5.68</td>
</tr>
<tr>
<td>DA+TS</td>
<td>4.46</td>
<td>4.11</td>
</tr>
<tr>
<td>P</td>
<td>4.86</td>
<td>4.30</td>
</tr>
<tr>
<td>Proposed</td>
<td>4.42</td>
<td>4.31</td>
</tr>
<tr>
<td>URI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td>13.421</td>
<td>26.73</td>
</tr>
<tr>
<td>DA+T</td>
<td>16.502</td>
<td>13.18</td>
</tr>
<tr>
<td>DA+TS</td>
<td>15.721</td>
<td>7.64</td>
</tr>
<tr>
<td>P</td>
<td>7.987</td>
<td>6.67</td>
</tr>
<tr>
<td>Proposed</td>
<td>6.234</td>
<td>6.60</td>
</tr>
</tbody>
</table>