Double-Array Compression by Pruning Twin Leaves and Unifying Common Suffixes

Susumu Yata, Masaki Oono, Kazuhiro Morita, Toru Sumitomo and Jun-ichi Aoe
Department of Information Science and Intelligent Systems
Faculty of Engineering, University of Tokushima, JAPAN
Email: yata@is.tokushima-u.ac.jp

Abstract—A Minimal Prefix (MP) trie is a tree structure for key retrieval, and a double-array is an efficient data structure for the MP trie. This paper presents two compression methods for the double-array. One method removes leaf nodes following two-way arcs (named twin leaves) from the MP trie. The other method unifies common suffixes. Experimental results show that space usage of the double-array is reduced to about 60% by the two methods.

I. INTRODUCTION
A trie[1] is a tree structure for key retrieval. In the trie, each key is registered as the path from the root to the leaf. The trie enables fast longest prefix matching because the path from the root to an internal node corresponds to the prefix of a registered key. Because of this merit, the trie is utilized as a natural language dictionary [2], an index of information retrieval systems [3], and so on[4], [5].

A Minimal Prefix (MP) trie [6] is an improvement of the trie. In the MP trie, a key is divided into a minimal unique prefix and suffix. The prefix is registered as the path from the root to the leaf. The suffix is kept as a string and linked to the leaf. The MP trie is excellent in space usage.

Matrix and list forms are common structures for the MP trie, but they have disadvantages in space usage and retrieval speed, respectively. A double-array [7] is one of the structure for the MP trie and a solution of these disadvantages. The double-array has advantages of the common structures. Retrieval by the double-array is as fast as the matrix form’s and space usage of the double-array is as small as the list form’s.

This paper presents two compression methods for the double-array. The methods focus on the suffixes of the MP trie and consider the retrieval time of the double-array. One method prunes pairs of leaves following two-way arcs (named twin leaves) from the MP trie because such pairs little affect the retrieval time. The other method unifies common suffixes and makes some keys share the unified suffix. The number of nodes is reduced by pruning twin leaves, and space usage of suffixes is reduced by unifying common suffixes.

Experimental results for 20,000–100,000 English keys show that over 1/3 of nodes are removed and space usage of suffixes is reduced to 50–70% by applying both compression methods. As a result, total space usage of the double-array is reduced to about 60%.

II. DOUBLE-ARRAY
A. Double-Array Structure
A double-array consists of two one-dimensional arrays called BC and TAIL. Array BC represents nodes of the MP trie and a minimal unique prefix is registered as the path from the root to the leaf. Array TAIL stores a suffix as a string terminated by the special end marker symbol. BC[i] indicates the i-th element of BC and TAIL[j] indicates the j-th letter on TAIL.

Each element of BC corresponds to the node of the MP trie, and the element consists of a flag LEAF and two integers called BASE and CHECK. Node s indicates the node corresponding to the s-th element BC[s], and BC[s].X indicates the member X of BC[s], such as BC[s].BASE. BC[s].LEAF = TRUE means that node s is a leaf, else node s is not a leaf.

In the double-array, the arc from node src to node dest with letter c is represented as shown in Fig. 1. The destination node dest is calculated by BC[src].BASE + c and the source node src is confirmed by BC[dest].CHECK.

If node s is a leaf, BC[s].BASE stores the start position of the suffix on TAIL as shown in Fig. 2. Symbol ‘#’ means the special end marker. In other words, the leaf s is linked to the suffix by BC[s].BASE.

B. Retrieval Algorithm for Double-Array
When a query str is given, a retrieval agent ag is initialized to the root indexed root = 1. The agent ag follows arcs with letter by letter of str. If ag fails to follow the arc, ag returns FALSE because str is not registered. Else, ag reaches a leaf (BC[ag].LEAF = TRUE) and compares str’s suffix with the suffix starting
from TAIL[BC[ag].BASE]. If the two suffixes are the same, ag returns TRUE because str is registered, else ag returns FALSE.

Fig. 3 shows the double-array for key set \( K = \{ \) “ace,” “bear,” “bee,” “care” \( \} \) and query examples “bear,” “bell,” and “cat.” Internal codes of letters ‘a’ to ‘z’ and symbol ‘#' are 0–25 and 26, respectively. When the query “bear” is given, the double-array returns TRUE. When the queries “bell” and “cat” are given, the double-array returns FALSE.

C. Retrieval Time and Space Usage of Double-Array

A retrieval agent follows an arc with \( O(1) \) and compares the query’s suffix with the suffix on TAIL as strings. Therefore, retrieval time of the double-array depends on the query’s length.

Space usage of the double-array depends on the number of BC’s elements and the length of TAIL. Moreover, the two factors are equivalent to the number of nodes and the total length of suffixes, respectively.

III. PRUNING TWIN LEAVES

A. Outline of Pruning Twin Leaves

Twin leaves indicate a pair of leaves following two-way arcs, such as nodes 2 and 6 in Fig. 3. A retrieval agent uses twin leaves only for selecting the correct key from two key candidates. For example, the retrieval agent for the query “bear” uses nodes 2 and 6 for selecting the correct key from two key candidates “bear” and “bee” in Fig. 3. Twin leaves don’t hold an important position in retrieval, but are the important factor in space usage. In short, pruning of twin leaves removes many nodes from the MP trie without degrading retrieval time.

B. Structure Modification by Pruning Twin Leaves

The pruning turns the parent of twin leaves to the leaf. However, a leaf having no sibling is not suitable for the concept of the MP trie. Therefore, if a new leaf produced by the pruning has no sibling, the new leaf should be removed, as in node 7 in Fig. 3.

A retrieval agent must refer to two suffixes from a new leaf. As a simple way, two suffixes which are stored successively enable the agent to refer to both suffixes, see Fig. 4. However, the agent must access the whole first suffix “ear” to refer to the second suffix “ee.” In addition, the agent can’t utilize the matching result of the query’s suffix and the first suffix for the second matching.

Against these problems, this paper presents a new structure for these two suffixes as shown in Fig. 5. The two suffixes are divided into four parts and stored in TAIL successively. The first part has the common prefix “e” and allows the agent to avoid repetition of matching for the common prefix. The second and third parts have the length and body of the first rest “ar” without ‘#.’ The length enables the agent to refer to the second rest passing over the first rest. The last part has the second rest “e,” but if matching for the head letter of the first rest succeeds, query’s rest is not equal to the second rest.

In the following description, a complex suffix indicates a pair of suffixes linked to a new leaf, and a simple suffix indicates an original suffix.

A leaf will have an additional flag TWIN to specify whether a simple or complex suffix is linked. The most significant bit of BASE is allocated to this purpose, and the link to TAIL is represented by the remaining bits (named LINK). When a retrieval agent finds the leaf end, the agent will switch the action depending on BC[end].TWIN. If BC[end].TWIN = FALSE, the agent compares query’s suffix with the simple suffix starting from TAIL[BC[end]].LINK. Else the agent compares query’s suffix with the complex suffix on TAIL.

C. Example of Pruning Twin Leaves

Fig. 6 shows the result of pruning twin leaves for Fig. 3 and query examples “beam” and “bee.” BASE of a leaf is divided into TWIN and LINK. By pruning, nodes 2, 6, and 7 are removed, and nodes 3–5 are reindexed to 2–4. Removed nodes correspond to TAIL[7], TAIL[9], and TAIL[4] in Fig. 6. As a result, the length of TAIL is expanded from 10 to 14, but the number of BC’s elements is reduced from 7 to 4.

Consider the case query “beam” is given. The retrieval agent is initialized to root and follows the arc from root = 1 to node 3 with letter ‘b’ because of BC[1].BASE + ‘b’ = 3 and BC[3].CHECK = 1. From BC[3].LEAF = TRUE and BC[3].TWIN = TRUE, it turns out that node 3 is a leaf linked to a complex suffix. Query’s suffix “eam” is compared with the complex
suffix starting from TAIL[BC][3].LINK = TAIL[4]. In this case, the suffixes have the common prefix “e,” but query’s rest “am” is not equal to the first rest “ar.” Finally, the agent returns FALSE because “am” and “ar” have the common head letter ‘a’.

Consider the case query “bee” is given. Similar to the case of query “beam,” the agent reaches leaf 3 and matching for the prefix succeeds. After that, query’s rest “e” is compared with the first rest “ar,” but matching for the head letters fails. In this case, query’s rest is equal to the second rest, so the agent returns TRUE.

IV. UNIFYING COMMON SUFFIXES

A. Unifying Simple Suffixes

In the original double-array, a suffix may contain another suffix as the postfix, and such suffixes can be unified because suffixes are stored in TAIL as strings terminated by ‘#.’ Fig. 7 shows the result of unifying common suffixes for Fig. 3 and query examples “bear” and “bee.” The suffix “e” of “bee” is the postfix of the suffix “ar” of “bear,” and these suffixes are unified. As a result, suffixes of “bear” and “bee” share TAIL[5], and the length of TAIL is reduced from 10 to 9.

B. Unifying Complex Suffixes

A complex suffix contains two strings terminated by ‘#;’ the prefix and rest. In addition, the prefix and rest are stored in order. X.pre and X.rest indicates the prefix and rest of a complex suffix X, respectively. If X indicates the complex suffix in Fig. 6, X.pre = “e” and X.rest = “are.”

To unify complex suffixes, three unifying rules A–C are applied in order. In the following description, X and Y indicate complex suffixes, and Z indicates a simple suffix. Fig. 8 shows an example of each rule.

A) Z is unified to X.pre or X.rest.

B) In case of X.rest = Y.rest, X.pre is unified to Y.pre.

C) X.pre is unified to Z or X.rest.

Notice that the unifying result depends on the combination of suffixes. For example, “e” may be unified to “ge,” “re,” “te,” and so on. However, only the combination in rule C affects the length of TAIL as shown in Fig. 9. If X.pre is unified to Z, X is not unified. Else if X.pre is unified to Z, X’.pre is unified to X.rest. Therefore, a complex suffix having the long prefix should be unified in priority.

C. Example of Unifying Common Suffixes

Fig. 10 shows the result of unifying common suffixes for Fig. 6 and the path corresponding to each key. In this case, the simple suffix of “care” is unified to the complex suffix of “bear” and “bee” by rule A, and the prefix of the complex suffix is unified to the simple suffix of “ace” by rule C. TAIL[2] and TAIL[3] are shared by the suffixes of “ace,” “bear,” and “bee.” TAIL[5] to TAIL[8] are shared by the suffixes of “bear,” “bee,” and “care.”

As a result, the length of TAIL is reduced from 14 to 8. Moreover, with the comparison to the original double-array in Fig. 3, the number of BC’s elements is reduced from 7 to 4, and the length of TAIL is reduced from 10 to 8.

V. EVALUATION

A. Theoretical Observation

Space usage of the double-array depends on the number of BC’s elements, the length of TAIL, and the size of each element. In the following description, these three factors are indicated by n, m, and k, respectively.
Each element of BC consists of three members LEAF, BASE, and CHECK. LEAF requires 1 bit. BASE requires \( \log(n, m) \) bits, but if the double-array unifies common suffixes, BASE requires \( \log(n, 2m) \) because leaf's BASE is divided into TWIN and LINK. CHECK requires \( \log n \) bits. In addition, each letter of TAIL requires \( \log k \) bits.

As a result, space usage of the original double-array is calculated as follows:

\[
\begin{align*}
\text{BC} & \quad n \times \left(1 + \log \left(\max(n, m)\right)\right) + \log n \\
\text{TAIL} & \quad m \times \log k
\end{align*}
\]

Space usage of the double-array unifying common suffixes is calculated as follows:

\[
\begin{align*}
\text{BC} & \quad n \times \left(1 + \log \left(\max(n, 2m)\right)\right) + \log n \\
\text{TAIL} & \quad m \times \log k
\end{align*}
\]

B. Experimental Observation

An experiment to examine the validity of the two compression methods, pruning and unifying, was performed. The experiment constructed the double-array using each combination of pruning and unifying. Key sets for the experiment were made by extracting 20,000–100,000 keys from an English dictionary at random. Table I shows the details of the dictionary.

Table II shows the details of constructed double-arrays. In Table II, DA\textsubscript{org} indicates the original double-array, DA\textsubscript{prune} means using only pruning, DA\textsubscript{unify} means using only unifying, DA\textsubscript{both} means using both pruning and unifying. An item size indicates space usage (kbytes) of the double-array. In addition, decimals under \( n, m \), and size indicate the ratio to the original double-array.

From the result of DA\textsubscript{prune} in Table II, it turns out that pruning removes over 1/3 of nodes, but expands the length of TAIL about 24–38%. However, each element of BC requires over 30 bits, and each letter of TAIL requires just 6 bits. In other words, the weight of BC's element is about 5 times more than the weight of TAIL's letter. As a result, space usage of the double-array is reduced to about 80% by pruning.

The result of DA\textsubscript{unify} in Table II shows that unifying reduces the length of TAIL to 15–35% without increasing the number of BC's elements. As a result, space usage of the double-array is reduced to 80–86%.

The result of DA\textsubscript{both} in Table II shows that the mix of pruning and unifying removes nodes just like pruning.

This is because DA\textsubscript{both} is constructed by applying unifying to DA\textsubscript{prune}. Moreover, the mix method reduces the length of TAIL to 40–74%. As a result, space usage of the double-array is reduced to 60–64%.

From this observation, it is clear that the presented methods are effective for compressing the dictionary based on the double-array.

VI. Conclusion

Compression methods for the double-array have been presented in this paper. The methods prune twin leaves to reduce the number of BC's elements and unify common suffixes to reduce the length of TAIL. In addition, the validity of the methods has been shown by the experimental observation.

The future study could focus on dynamic updates of the compressed double-array.

REFERENCES