Research of an Improved Algorithm for Chinese Word Segmentation Dictionary Based on Double-Array Trie Tree

Wenchuan Yang, Jian Liu, and Miao Yu

Beijing University of Posts and Telecommunication, Beijing, 100876, China
yangwenchuan@bupt.edu.cn

Abstract. Chinese word segmentation dictionary based on the Double-Array Trie Tree has higher efficiency of search, but the dynamic insertion will consume a lot of time. This paper presents an improved algorithm-iDAT, which is based on Double-Array Trie Tree for Chinese Word Segmentation Dictionary. After initialization the original dictionary. We implement a Hash process to the empty sequence index values for base array. The final Hash table stores the sum of the empty sequence before the current empty sequence. This algorithm adopt Sunday jumps algorithm of Single Pattern Matching. With slightly and reasonable space cost increasing, iDAT reduces the average time complexity of the dynamic insertion process in Trie Tree. Practical results shows it has a good operation performance.

Keywords: Double-Array, Trie Tree, Time Complexity, Word Segmentation Dictionary.

1 Introduction

Presently the matching algorithms based on dictionary is still the method used by the dominant search engine company. The foundation of Chinese word automatic segmentation is dictionary, and its structure is directly related to the speed and efficiency for word segmentation. Automatic word segmentation is basis for Chinese information processing system, which leads to further syntax and semantic analysis of Chinese text[1]. Lexicon will directly influence the segmentation speed. The data structure of dictionary is mainly through the indexed methods, which include index table, inverted lists, hash tables and search tree[2].

A maximum matching algorithm is presented in paper[3]. The nearest neighbor matching algorithm document is put forward in paper[4] based on the first word Hash algorithm. In paper[5], it presents the dictionary organization method and algorithm to combine first word Hash and entire word binary search, and this further improve the segmentation speed. Since there are so many Chinese words, it's hard to use the Hash table to control the data distribution, and reduce the conflict. There are 6768 commonly used Chinese characters in GB-2312, each Chinese characters can be mapped uniquely to 1-6768[6]. So we can use Double-Array Trie Tree as the data structure of the Chinese word segmentation dictionary. A linear table based Trie Tree is presented in paper[7], and the double array Trie Tree is an improved version.

The searching efficiency of Double-Array Trie Tree is *O(n)*, n for matching character length. It has a good search performance, and weak insert performance. Its insert performance is still $O(cm^2)$ even after tuning. Here m is the character set size, constant C. For the study of Chinese dictionary based on Double-Array Trie Tree, method for processing node with more branch first to improve space utilization in paper[8].

As we mentioned before in paper[2], there's a method to arrange the conflict nodes into the Hash table without redistribute node to improve the efficiency of the insertion process. Yet the Hash conflict is inevitable, and the use of Hash will increase the number of search. A optimization method based on genetic algorithm and Sherwood double array Trie Tree is purposed in paper[8]. It improves the space utilization rate, and it also avoid the local optimal solution for the algorithm.

In this paper, we will propose an improved algorithm-iDAT, which is based on Double-Array Trie Tree for Chinese Word Segmentation Dictionary. iDAT optimize the efficiency of inserting together with the ability of search performance as for Double-Array Trie Tree.

2 Double-Array Trie Tree

2.1 Double Array Trie Tree

Trie Tree is essentially a deterministic finite state automata, each node represents a state. Its state transferred according to the different input variables.

Double array uses two arrays as base[] and check[] to implement Trie Tree. Assume the input character is *c*, and Double-Array Trie Tree changes from state *s* to state *t*, it fits for the following conditions.

$$\text{base[s]+c=t} \tag{1}$$

$$\text{check[t]=s} \tag{2}$$

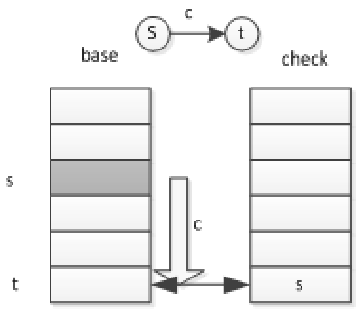

Fig. 1. Structure of Double array Trie Tree

For Double-Array Trie Tree shown in the Fig.1, s and t is the array index. With input *c*, the state *s* transfer to state *t*, so we have t=base[s]+c, and check[t]=s. So we can say that check array keeps record of the translated state for state *t*.

2.2 Insert Processing

Assuming that each state corresponds to an array index. For state *s*, if base[s] and check[s] are both 0, *s* stands for an empty place(Note: check can be when the node is idle). Assume t_1, $t_2 \ldots t_n$ is the suffix state begin with *s* , $c_1, c_2 \ldots c_n$ is respectively corresponding to the input state transition. *base*[*s*] is defined by the following process:

If there's a base[s], where $base[s]+c_1=0$, $base[s]+c_2=0$, $\ldots base[s]+c_n=0$, the base[s] can be accepted. The suffix state t_i can be stored in the $base[s]+c_i$.

If the new t_x suffix state appears, and the corresponding array for $base[s]+c_x$ is not empty, then we need to redo the above process, and recalculate value for *base*[*s*].

To determine the value of *base*[*s*], the entire array need to be traversed. The initial value of *base*[*s*] is determine by finding the empty node.

2.3 Insert Optimization

To avoid traversing arrays for an empty node from the very beginning, empty state can be used to construct the empty state sequence based on double linked table structure, as follows:

Assume $r_1, r_2 \ldots r_{cm}$ are the array index of the empty state for double array state

$$check[0] = -\ r_1 \tag{3}$$

$$check[\ r_i\ ] = -\ r_i + 1\ ; \qquad (i \in [1, cm\text{-}1]) \tag{4}$$

$$check[\ r_{cm}\ ] = 0 \tag{5}$$

$$base[0] = -\ r_{cm} \tag{6}$$

$$base[\ r_1\ ] = 0 \tag{7}$$

$$base[\ r_i + 1] = -\ r_i\ ; \qquad (i \in [1, cm\text{-}1]) \tag{8}$$

From the above definition, when the check value is negative, it indicates that the location is empty.

Under the array based double linked table structure, we can choose the value for initial state *base*[*i*] by directly traverse empty nodes.

This reduces the time compared with the traversal of the entire array. It also optimizes the node deletion operation with double linked table structure.

3 Improved Algorithm

The Double-Array Trie Tree saves the space compared with the traditional Trie Tree. When this insert processing encountered conflict, it need to determine the maximum sub-prefix base value, and this is time costly. To solve this problem, we propose a iDAT algorithm to optimize its insertion time complexity with the same search efficiency.

3.1 Definitions for iDAT

We have the following definitions for iDAT.

1. $N = \{n_1 n_2 ... n_l\}$ is the base(or check) array node set, n_i denotes the i node, i is corresponding the array index, l is the array size, and *l=N_Length*;

2. $R = \{r_1 r_2 ... r_m\}$ is the all empty node set in N, r_j is the j-th empty node, m is empty node length; m=*R_Length*;

3. T is the maximum sub-prefix under insert state, $S = \{S_1 S_2 ... S_k ... S_{s\_length}\}$ is T's child nodes character set. Since *base*[*s*]+*c*=*t*, the character sequence in S set, which was created by traversal N, is in ascending order. So we have $S_k > S_{k-1}$ and its length is *s_length*;

4. Assume $S^* = \{S_1^* S_2^* ... S_k^* ... S^*_{s\_length}\}$ is all the inserted child nodes character set. *pos*($\overset{*}{S_1}$) is the index for the latest insert character $\overset{*}{S_1}$;

5.Assume *START* as the last character to be inserted position in *Set S*. Since *base*[r_i+1] = - r_i, we have *START* = -*base*[*pos*($\overset{*}{S_1}$)];

3.2 Hash Table

Assume the length of Hash table *D=N_Length/10*, and mapping function as *Hash*(*t*)=*t%D*. We use linked list to solve the node conflict.

step 1: For the empty node counting, we have a temporary variable *tem=0* to initialize the Hash table *ht*[*D*];

step 2: After double array dictionary is initialized, we traverse the empty node from the *check*[0] position;

step 3: For the empty nodes corresponding array index, a Hash transform is done sequentially.

step 4: Let *ht*[*Hash(i)*]=*tem*; *tem* value increased by 1,here *i* is the current empty array index.

Here count is sum of all the empty node before the array position.

3.3 Skip Function

To improve and accelerate the algorithm, we design a skip function *isAccept*(*S*,*START*) to determine whether to skip certain base.

The detail work flow is listed as,

step 1: Assume $\Delta s = s_{length} - s_1$, Δs stands for intervals between s_{length} and s_1. So the insertion position for s_1 should be *START* - Δs. Try to find whether *check*[*START* - Δs] is less than 0. If it is less than 0, the location is empty and enter step2. Otherwise it jump out and returns false;

step 2: Calculate $\Delta count = hash(START) - hash(START - s\_length)$ -1,Here *Hash(START)* is the empty position number before *START*. $\Delta count$ is the empty node numbers between the head node and tail node in Set S. IF $\Delta count < s\_length$ -2 , jump out and returns false. Otherwise enter step3;

step 3: $base^0[T] = START - S_{s\_length}$, here $base^0[T]$ is the initial values of base[T]. Let's record the value and return true

3.4 Tree Construction

The construction steps are described as,

step1: initialization the dictionary according to the Double-Array Trie Tree optimization flow described before. For the initialized array, let's construct empty node linked table as formula (3)-(8). Then create Hash table.

step2: Assume insert word is Ts_x, base[T]+ s_x is not empty. Let's traverse the check table, and get the character set $S = \{S_1 S_2 ... S_k ... S_{s\_length}\}$,whose suffix is the maximum sub-prefix T.

step 3: Set *S* as input parameter for *isAccept(S,START)*. If the return value is true, it determines the initial value of base[T], and enter step4. Otherwise enter step5.

step 4: Process node inserting as described in iDAT algorithm. If the initial value for base is found, let's record the pos(s_1^*). Otherwise enter step5.

step 5: According to the formula *base*[r_i+*1*] = -r_i, change the value of *START*, and jump to step3.

step 6: If pos(s_1^*) is 0, let's update the Hash table as described in step1.

The skip function *isAccept(S,START)* is a convergence algorithm. Its ordinary time complexity is *O(*1*)*, and its worst-case complexity is *O(* cm^2 *)*(here *c* is constant,*m* is empty node number). Since the *isAccept* function has the ability to skip and jump quickly. It avoids some unnecessary compare, and also reduces the average time complexity of the algorithm.

To improve the average time complexity, we create a Hash table. It does cost some space. In order to minimize the number of maintenance for Hash table. iDAT achieves inserting from back to front.

4 Evaluations

The experiment try to compare our improved double array Trie Tree(iDAT) solution with the optimized double array solution proposed in paper[8].

The experimental environment: CPU Core i7, Memory 16Gb, Operating system is window7, Programming language is Java over Eclipse. The dictionary to be tested is the open Chinese lexicon provided by sogou (http://www.sogou.com/), which includes 157201 entries. After we load the test dictionary with Double-Array Trie Tree, the base (check) array size is 574464.

During the constructing of Double-Array based Chinese dictionary, we find through the actual simulation ,that there is a certain relationship among the success rate of insertion, double array empty node proportion, and insert node number.

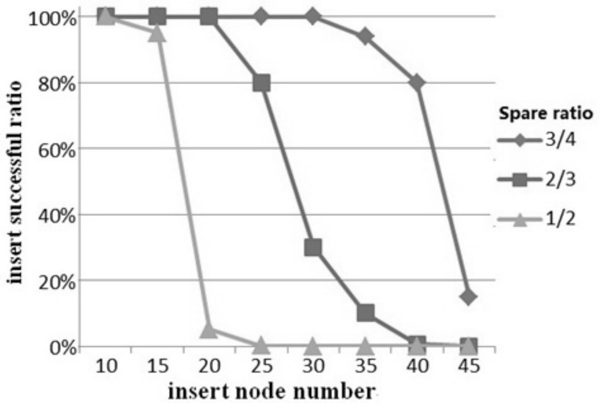

Fig. 2. Diagram of relation between insert successful ratio and insert node number

Fig. 2 shows relation between insert successful ratio and insert node number. Idle rate is the proportion of empty nodes in the array sizes. There are three array idle rate, which are 3/4, 2/3, and 1/2. The X-axis is the insert node number, and the Y-axis is the success rate. From Fig. 2, we can find that with constant idle rate, the success rate of insertion may have radical transition under certain insert values.

For the insertion algorithm, the word with a number of child nodes has the high priority. Based on the idle rate of array, it choose to allocate new space in the tail of word. This can help to skip the insertion position selection process for entire array, and can further improve the efficiency of the insertion algorithm.

From the simulation data, we find that the idle rate in array is about 2/3. So we use the red curve to mark them. When the node number exceeds 35, we allocate some new space in the end of the array.

The size of new space is determined by the mapping code range for the child nodes. The time cost comparison for iDAT and EDS is shown in table 2.

Table 1. Time cost comparison for iDAT and EDS

Insert entry \ Algorithm	EDS	iDAT
100	6.47	5.49
200	9.23	7.01
300	12.89	10.72
400	22.58	18.36
500	31.59	22.61
600	40.04	28.18

As can be seen from Table 1, the advantage of iDAT is not obvious for less inserting entry. Accompany the increasing for entry, the optimized solution will skip more unnecessary compare. It seem more effective.

The Hash table used in iDAT need extra space cost. The simulation result shows that ,when the size of hash table is about 1/10 to the total double array size, the performance can meet the requirements. Table 2 shows the analysis of the space cost of two methods.

Table 2. Space cost comparison for EDS and iDAT

algorithm	space cost
EDS	array size 574,464
iDAT	Array size 583,632 Hash table size 57,449

5 Conclusions

In this paper, we introduce iDAT, which is an improved algorithm for Chinese word segmentation dictionary based on double array Trie Tree. iDAT can implement fast mechanical word segmentation, such as maximum matching or reverse matching. Time cost for search in iDAT is almost the same with the original solution. But it is more effective than other solutions in insertion operation. In iDAT also can solve the space cost problem for traditional Trie Tree during Chinese word segmentation. Anyway, the idle rate for array is about 60% in the actual simulation process. Further research on the algorithm to optimize the space cost need to be done.

Acknowledgment. This paper is supported by the Opening Project of State Key Laboratory of Digital Publishing Technology.

References

1. Huang, C.N.: A review of ten years of Chinese word segmentation. Journal of Chinese Information 147, 195–199 (2007)
2. Zhao, H.Y.: A study on Chinese word segmentation based on Double-Array Trie Tree. Journal of Hunan University 22, 322–329 (2009)
3. Zhao, C.Y.: A word segmentation method based on the word. Journal of Soochow University 18, 44–48 (2002)
4. Chen, G.L.: An improved fast segmentation algorithm. Journal of Computer Research and Development 37, 418–424 (2009)
5. Li, Z., Xu, Z., Tang, W.: A full two points maximum matching in Computer Engineering and application of fast segmentation algorithm. Journal of Computer Science 38, 102–108 (2005)
6. Li, J.: A fast algorithm for query Chinese dictionary. Journal of Chinese Information 137, 97–101 (2006)
7. Wang, S.: Research on Double-Array Trie Tree algorithm optimization and its application. Journal of Chinese Information 138, 131–137 (2006)
8. Wang, S., Li, Z., Ke, X.: Based on improved genetic algorithm and Sherwood thought the Double-Array Trie Tree. Journal of Computer Engineering 78, 231–236 (2009)