Research on Dictionary-Based Word Segmentation Algorithms Using Trie Structure

Boxing Zhang
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: 1824595833@qq.com

Qinlong Kang
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: 1142926763@qq.com

Xin Jing
School of Computer Science and Engineering
Xi'an Technological University
Xi'an, China
E-mail: jingxin@xatu.edu.cn

***Abstract*—This study investigates dictionary-based word segmentation algorithms, which are essential in Natural Language Processing (NLP). Chinese word segmentation poses significant challenges due to the lack of clear word delimiters in the language. This paper explores the advantages and limitations of dictionary-based segmentation algorithms, focusing on how data structures such as Trie and Double-Array Trie (DAT) can enhance segmentation efficiency. An analysis of Trie and DAT structures leads to an optimization achieving constant-time state transitions. This paper evaluates and compares various segmentation algorithms, including full segmentation, forward maximum matching, backward maximum matching, and bidirectional maximum matching. The inherent limitations of dictionary-based segmentation, particularly its dependence on dictionaries and poor disambiguation capability, are also discussed.**

Keywords-Word Segmentation; Trie; Natural Language Processing; Double-Array Trie

I. INTRODUCTION

Word segmentation [1] is a fundamental in Natural Language Processing (NLP). It forms the basis for tasks such as word vector encoding, part-of-speech tagging, syntactic parsing, and text analysis. Unlike English, where words are clearly separated by spaces, Chinese words appear as continuous strings. Consequently, any NLP task involving Chinese must address the issue of segmenting text into individual words.

Popular approaches to Chinese word segmentation [2] include statistical-based methods and dictionary-based techniques. Statistics-based segmentation methods are more expensive and slower. Dictionary-based segmentation is one of the simplest and most frequently used methods. It only requires the construction of a dictionary and a strategy for matching words against it. Dictionary lookup essentially involves string matching, where a target string is compared to entries in the dictionary based on specific rules. This approach can be categorized into Full segmentation algorithms, Forward maximum matching algorithms, Backward maximum matching algorithms, and Bidirectional maximum matching algorithms.

Though dictionary-based segmentation is not complex, and its disambiguation performance is poor, it has the advantage of being fast. The key to leveraging this advantage lies not in the segmentation algorithm itself but in the underlying data structure supporting the dictionary. Dictionary-based segmentation efficiency depends

DOI: 10.2478/ijanmc-2025-0005

heavily on the underlying data structure used to represent the dictionary. Traditional approaches relying on linear scans or hash-based lookups may struggle to achieve the speed and scalability required for modern NLP applications. In this context, advanced data structures like Trie and Double-Array Trie (DAT) provide significant advantages:

Efficient matching: Tries enable prefix-based matching with time complexity proportional to the length of the query, making them well-suited for word segmentation tasks.

Optimized memory usage: DAT structures further enhance Tries by reducing memory overhead and enabling constant-time state transitions, which is critical for handling large-scale dictionaries.

The study aims to address the problem of low efficiency in traditional Chinese word segmentation by integrating Trie and DAT structures into dictionary-based algorithms. Specifically, the research seeks to:

Analyze the theoretical advantages of Trie and DAT in the context of Chinese word segmentation. Design and implement segmentation algorithms leveraging these data structures. Evaluate the performance of the proposed methods through extensive experiments, focusing on speed, memory usage, and scalability.

Combining theoretical insights with practical experiments comprehensively explores the optimization of Chinese word segmentation using advanced data structures. The findings aim to contribute to the broader field of NLP by offering efficient solutions to a critical preprocessing step in text analysis.

II. Data Structure

A. Trie

Trie, also known as a prefix tree, is a tree-like data structure [3] that represents a deterministic finite automaton (DFA), where each node corresponds to a state representing the prefix of a string. Moving from a parent node to a child node signifies a state transition. A search is completed when the terminal state is reached, or no further transitions are possible. The key idea of a Trie is leveraging shared prefixes to save time at the cost of additional space, minimizing redundant string comparisons, thus reducing query time and improving efficiency.

Trie is particularly suited for tasks such as statistics [4], sorting, and storing large amounts of strings. Each edge in the Trie corresponds to a character, and a path from the root node to a leaf node forms a complete string. Trie structure do not directly store strings at nodes. Instead, they treat a word as the path from the root to a specific node, marking that node as the word's endpoint.

For example, in the Trie shown in Fig. 1, each string is represented by a path. Searching for a word involves following the path starting from the root node. If the search reaches a node with a special marker, it indicates that the string exists in the set; otherwise, it means the string is not present.

1) Construction

A Trie works like a dictionary, with its directory structure mimicking real-life dictionary organization.

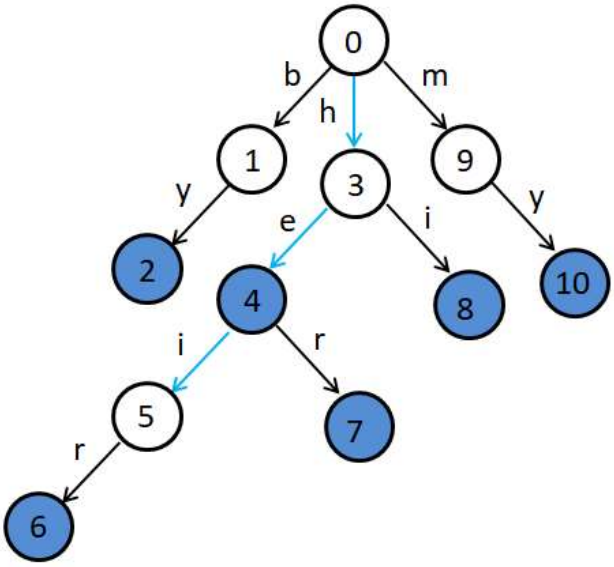

Figure 1. Trie

A string is essentially a path. To query a word, simply follow this path starting from the root node. If it reaches a node with a special marker, it indicates that the string exists in the set; otherwise,

it means the string is not present. The paths for string are shown in Table 1.

a) First, a root node is defined, which does not contain any value.

TABLE I. PATH DIAGRAM

String	Path
by	0—1—2
he	0—3—4
heir	0—3—4—5—6
her	0—3—4—7
hi	0—3—8
my	0—9—10

b) Then, using a for loop in a manner similar to Depth-First Search (DFS), each character of the string is checked sequentially to see if it exists in the Trie.

c) If a character does not exist, a child node is created and inserted.

d) If it does exist, the next subtree is retrieved in a DFS manner.

The construction process of the Trie can be broken down as shown in Fig. 2.

The overall steps are as follows:

a) First, define the root node, which does not store any characters.

b) When adding the first string, "by", to the Trie, since "b" does not exist in the Trie, it is added. Since the first character is not present in the Trie, none of the subsequent characters in the string will have been traversed either, so the following characters are added starting from the node indicated by "b".

c) After all characters of the first string are added to the Trie, the current node's isEnd is set to True, indicating that this node marks the end of the added string.

d) This process is repeated until all strings are added.

2) Lookup

Trie lookup involves matching a string by traversing the nodes in the Trie:

a) Start at the root node and search for the first character of the string.

b) If the character is not found, return a negative result, indicating that the string is not in the dictionary.

c) If the first character is found, proceed with a DFS from the matched node. If at any point during this traversal, the character is not found, as described in step b, the matching is considered unsuccessful.

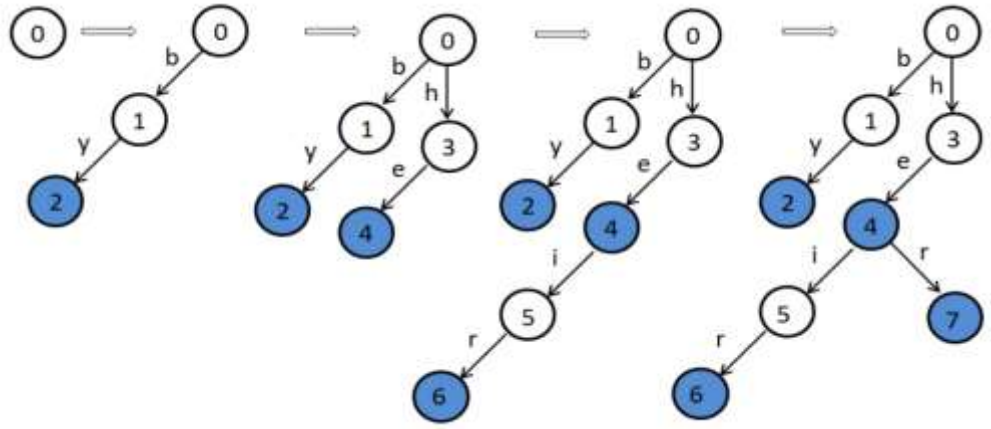

Figure 2. The construction process of the Trie

d) If all characters in the string are matched, check whether the final node is marked as a special terminal node, if isEnd=True, indicating that the full string is present in the dictionary.

In Fig. 1, the blue path shows the matching process for "hei", but since the last node (node 5) is not a terminal node, "hei" is a substring rather than a complete string in the Trie, resulting in a failed match.

3) Performance Evaluation

Constructing a Trie requires scanning all strings, resulting in a time complexity of O(n), where n is the total length of the strings. However, the Trie can be constructed incrementally, allowing for simultaneous querying. The time complexity for querying is O(k), where k is the average length of the strings. Thus, for frequent lookups within a set of strings, the Trie structure is highly efficient.

The key concept of the Trie is to trade space for time. In a typical binary tree, nodes store pointers to left and right children. However, in a Trie, nodes can have many children—up to 26 in the case of the English alphabet. To store these child pointers, an array of pointers (or indices) is used, where each pointer corresponds to a child node. For a set of strings with an average length of l, the worst-case memory usage for storing the pointers in a Trie would be proportional to 26^l, leading to significant memory overhead.

This is particularly problematic in Chinese, where each node needs an array far larger than 26. Even with Java's UTF-16 encoding, a perfect hash requires each node to maintain an array of size 65,536. For a string of length n, the worst-case memory usage can grow exponentially, leading to Out of Memory (OOM) errors. Such a design is impractical for Chinese word segmentation.

What are potential solutions to this issue?

B. Double-Array Trie (DAT)

We apply a binary search strategy to all nodes except the root, enabling the first character match to be achieved with a time complexity of O(1). However, for subsequent nodes using binary search, the state transition complexity is O(log c), where c represents the number of child nodes. When c is large, the transition speed is still slow. In 1989, Jun-Ichi Aoe proposed the Double-Array Trie [5], a data structure with a constant state transition complexity.

1) Algorithm Principles

The DAT is a finite-state automaton [6] that consists of two integer arrays: base and check. Each element in the base array represents a node (or state), while the check array indicates the predecessor state of a node. Initially, all elements in the base and check arrays are set to 0, signifying that a state is unoccupied. If a state represents a complete word, its corresponding base value is set to a negative number (if the state represents a complete word and is not a prefix for any other word, its base value can be set to the negative value of its state position). A state transition is successful when the following conditions are met:

$$p = base[b] + c$$
$$check[p] = base[b]$$

Where *b* is the index of the current state, *c* is the value of the accepted character, and *p* is the index of the transition state. If these conditions are not met, the state transition fails. The state transfer process is shown in Fig. 3.

For example, to check if a transition to the state "西安工" is possible, we compute: "西安工" = *base*[西安] + "工" and verify whether *check*[西安工] == *base*[西安]， This check is performed with one addition and one integer comparison, allowing state transitions in constant time.

2) Construction

The construction of the DAT essentially involves traversing a regular Trie and assigning indices in the double arrays for each node while maintaining and updating the values of the arrays. The most complex part of the construction is determining the base value for each state. The base value of a state must ensure that there is enough space in the array for all its child nodes. The construction process can be outlined as follows:

Take Trie shown in Fig. 4 as an example.

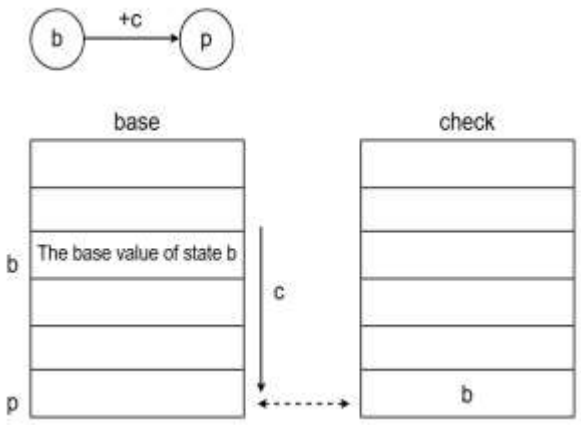

Figure 3. The state transfer process

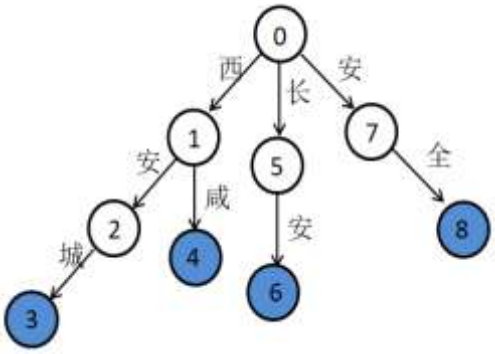

Figure 4. Example of Trie

To simplify, we manually encode the characters in the dictionary as follows, as in Table 2.

a) Initialize the Arrays, as in Table 3

Initialize *base* [0] = 1, with the check array set to all zeros.

b) Assign subscripts to nodes

The root node has child nodes: "西", "长", and "安". We assign positions to them such that *check* [*base* [0] + S_i.code] = 0, meaning the assigned positions are free.

For "西", $S_{西}$.code = 1, *base*[0] = 1, *base*[0]+ $S_{西}$.code = 2. If *check*[2] = 0, assign index 2 to "西", setting *check*[2] = *base*[0] = 1, linking the state represented by index 2 with its parent (the root node). For "长", assign index 6 similarly, setting *check*[6] = *base*[0] = 1. For "安", assign index 3, setting *check*[3] = *base*[0] = 1.

No conflicts occurred during this round of array maintenance, and the results are shown in Table 4.

c) Continued distribution

In the next round, treat the child node "西" as the new parent. First, check whether "西" is a leaf node; since it's not, find its children: "安" and "咸". For "安", $S_{安}$.code = 2, the task is to find a free position for insertion. The transition point is no longer the root node but its parent node "西". Following the rule for calculating the index *p* using the formula *base*[s]+c = p, where is the current node, we have: *base*[2]+$S_{安}$.code = p. There are two unknowns in this equation: the position of *base* [2] and the index *p* for insertion. Let's assume that the insertion index *p* is 4, i.e., p = 4. We can then deduce: *base* [2] = p - 2 = 2. The results at this stage are shown in Table 5.

TABLE II. ENCODE THE CHARACTERS

Character	Code
西	1
安	2
城	3
咸	4
长	5
全	6

TABLE III. INITIALIZE THE ARRAYS

Character											
Serial Number	0	1	2	3	4	5	6	7	8	9	10
base	1										
check	0	0	0	0	0	0	0	0	0	0	0

TABLE IV. START ALLOCATION

Character			西	安			长				
Serial Number	0	1	2	3	4	5	6	7	8	9	10
base	1										
check	0	0	1	1	0	0	1	0	0	0	0

Next, we process the second child node, “咸.” Based on the known information, we can calculate *check*[*base*[2] + S $_{咸}$.code] = *check*[6]. Since *check*[6] ≠ 0, this indicates a conflict because the position was already assigned to the character “长” during a previous step. To resolve this conflict, we shift the pre-allocated position by one slot (though it can be shifted by more depending on the algorithm’s rules). Now, *check*[7] = 0, so position 7 is allocated to “咸”.

To ensure that both *base*[s]+c = p and *check*[p] = *base*[s] are satisfied, we update the arrays. By backtracking, we compute *base*[2] = p - S $_{咸}$.code = 3. Additionally, the state of the previously inserted node “安” must be maintained. Thus, we verify if *check*[*base*[2]+ S $_{安}$.code] = *check*[5] = 0. If this condition is satisfied, “安” is inserted at position 5, and both *base*[5] and *check*[5] are updated accordingly. The array at this point is shown in Table 6.

d) Iteration

And so on, Similarly, it is worth noting that when a node is a leaf node, the corresponding base value is set to -1, as shown in Table 7.

It can be observed that the key challenge in constructing the DAT lies in handling state conflicts. In the process of constructing the DAT, conflicts are inevitable. These conflicts often arise when multiple words share common characters. For example, in the words “西安城”, “长安”, and “安全”, the character “安” is common across all of them. While these words can share a common prefix in the Trie, issues arise when the suffixes contain identical characters, or when suffixes overlap with prefixes. In such cases, a new node must be constructed, which will inevitably cause a conflict. Once a conflict occurs, the base value of the parent node must be adjusted to ensure that all child nodes can find a free position for insertion. This also necessitates the reconstruction of any previously constructed child nodes.

TABLE V. CONTINUE ALLOCATION

Character			西	安	安		长				
Serial Number	0	1	2	3	4	5	6	7	8	9	10
base	1		2								
check	0	0	1	1	0	0	1	0	0	0	

TABLE VI. REALLOCATION PROCESS

Character			西	安		安	长	咸			
Serial Number	0	1	2	3	4	5	6	7	8	9	10
base	1		3								
check	0	0	1	1	0	3	1	3	0	0	

TABLE VII. FINAL RESULT

Character			西	安	安	安	长	咸	城		全
Serial Number	0	1	2	3	4	5	6	7	8	9	10
base	1		3	4	-1	5	2	-1	-1		-1
check	0	0	1	1	2	3	1	3	5	0	4

As a result, the construction time for a DAT can be quite long, and sometimes inserting a single new word may require reconstructing the entire Trie. Additionally, the order in which words are inserted can lead to different conflict scenarios. Typically, when constructing DAT, all the first characters of the words are built first, followed by the child nodes for each word. In this way, if a conflict arises, it can be isolated to a single parent and its immediate child nodes, thus avoiding the need for widespread reconstruction of the Trie.

III. ALGORITHM

A. Full Segmentation Algorithms

1) Algorithm Concept

The Full Segmentation Algorithms aim to identify all possible words in a segment of text. The logic behind implementing naive full segmentation algorithms is simple: simply traverse the continuous sequences in the text and check whether each sequence exists in the dictionary. The algorithm concept is illustrated in Fig. 5.

The core idea of the Full Segmentation Algorithms involves two for loops that traverse every possible continuous sequence in the text for comparison. The first comparison starts with the first character and then sequentially adds each following character to form a string. If the string exists in the dictionary as a valid word, it is considered a word and added to our list of segmented words. The second loop starts from the second character and combines subsequent characters to form new strings, continuing this process until the last character of the text is reached.

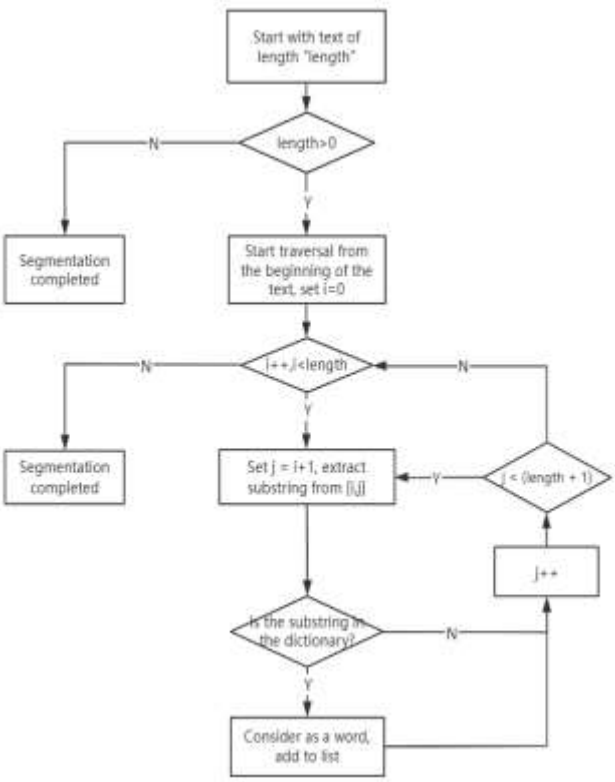

Figure 5. Full Segmentation Algorithm Flow

In the code, an ordered set TreeMap [7] is used, with a complexity of O(log n). The test results are shown in Fig. 6.

As shown, the Full Segmentation Algorithm outputs all individual characters and words from the text. However, this is not the desired outcome for Chinese word segmentation. In practice, what we need is a meaningful sequence of words. For example, we expect "西安工业大学" to be segmented as a single word, rather than fragmented as "西安+工业+大学". This problem occurs for two reasons:

a) The dictionary does not contain "西安工业大学" as a continuous word.

b) The Full Segmentation Algorithm does not account for the fact that longer words typically express meanings that are closer to the actual context.

Thus, we can solve these issues by expanding the dictionary and incorporating the Longest Matching Algorithm.

2) Modifying the Dictionary Word Repository

We can use the mini core dictionary that comes with HanLP. This dictionary is a plain text file that can be opened directly with a text editor. The format after opening is shown in Fig. 7.

The dictionary format in HanLP is a space-separated table. The first column contains the words, and the next two columns represent the word type and corresponding word frequency (the frequency is derived from a corpus). During full segmentation, it was observed that the dictionary does not contain "西安工业大学" as a continuous word.

By manually modifying the dictionary, "西安工业大学" is added to the dictionary repository, as shown in Fig. 8.

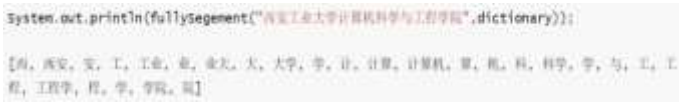

Figure 6. Test results

Now, let's retest the Full Segmentation results, as shown in Fig. 9. As the results clearly show, "西安工业大学" appears in the segmentation output. This validates our solution to the first problem, which involved modifying the dictionary. However, in real applications, it is not feasible to manually create a dictionary based on the text to be segmented. Instead, segmentation is performed based on an existing dictionary, which inherently limits the ability to recognize new words.

Figure 7. The format

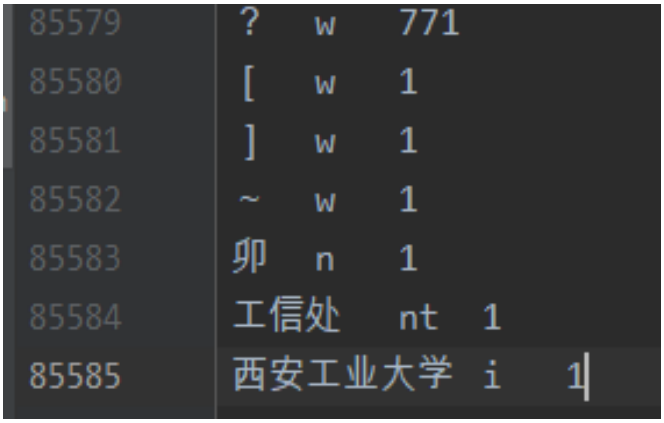

Figure 8. added to the dictionary repository

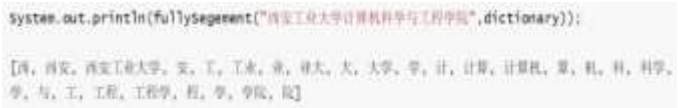

Figure 9. Results After Dictionary Modification

B. Maximum Matching Algorithm

While Full Segmentation Algorithms can capture all the words present in the dictionary, many of the words identified, particularly single characters, are often meaningless. To obtain a more meaningful sequence of words, we introduce the Maximum Matching Algorithm [8],

establishing the rule that "longer words take priority". Specifically, when traversing and matching words starting from a particular index, the longer word is preferred. Depending on the strategy, this leads to three variants: Forward Maximum Matching Algorithm (FMM), Backward Maximum Matching Algorithm (BMM), and Bidirectional Maximum Matching Algorithm (BIMM).

1) Forward Maximum Matching Algorithm

FMM begin by scanning the text from index 0 in a forward direction. In contrast to the logic of the Full Segmentation Algorithm, a new variable *longestword* is introduced to record the longest matched word. The longest word is then added to the word list. The algorithm concept is shown in Fig. 10.

In the FMM, the first *for* loop retrieves the longest string starting from the first character. The second *for* loop then iterates over all possible combinations starting from the first character. If a word exists in the dictionary, the algorithm checks whether the current word is longer than the previous longest match. If it is, the previous match is replaced with the current one. The longest word found in the first round of matching is then added to the list. To ensure the segmentation results, when concatenated, match the original text, the second round of traversal begins from the next character after the longest matched word, and the process repeats until $i >$ text.length(). The implementation result is shown in Fig. 11.

Observation of the output successfully addressed the two main issues encountered with the Full Segmentation Algorithm. However, a new issue arises. In the text "西安工业大学计算机科学与工程学院", the valid word segments are "工程" and "学院". Using the Forward Segmentation Algorithm, based on the principle that "longer words take priority", the segmentation results include "工程学" and the single character "院", which are clearly incorrect. The error arises because "工程学" takes precedence over "工程" due to its length.

To eliminate this ambiguity, the process can start from the end of the text and traverse forward to find the longest match.

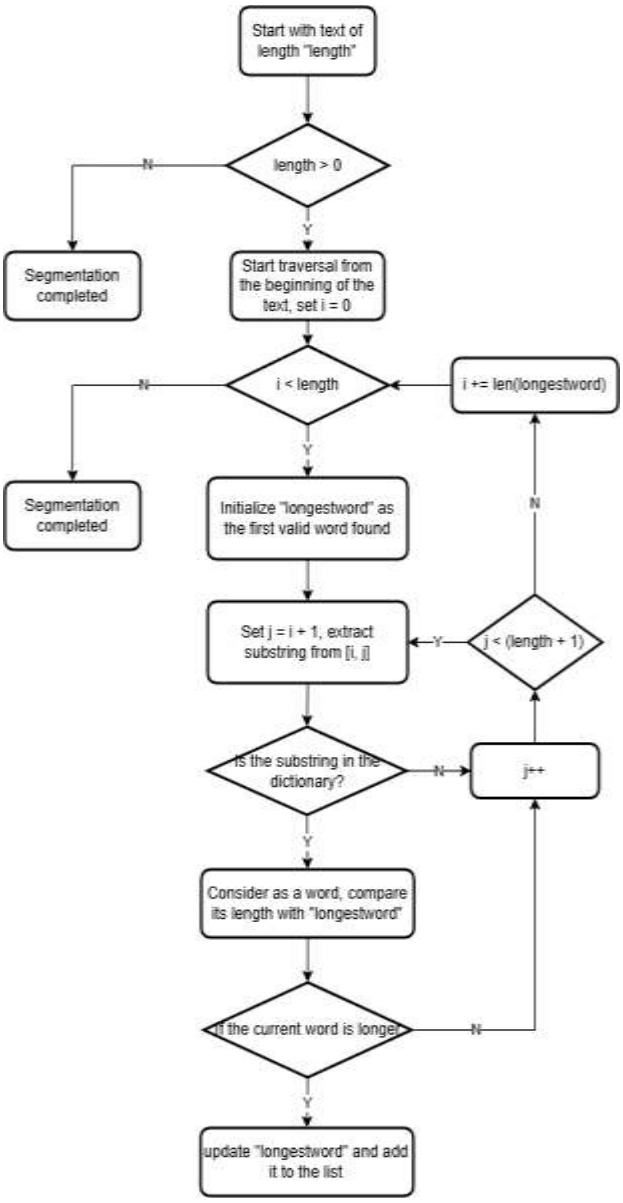

Figure 10. FMM Concept

```
system.out.println(foward_longest_seg("西安工业大学计算机科学与工程学院",dictionary));
输出:
    [西安工业大学, 计算机, 科学, 与, 工程学, 院]
```

Figure 11. FMM Segmentation Results

2) Backward Maximum Matching Algorithm

BMM [9] follows the same concept as the FMM, but instead starts from the last character of the text and traverses forward. It's important to note that does not mean reversing the text; the segmentation results are still ordered according to the sentence's original sequence. As the algorithm concept has already been described.

Fortunately, the issues caused by the FMM do not appear in the BMM. However, if we use the phrase “醒目的大树” as the segmentation target, the result is problematic:[醒,目的,大树].

This demonstrates that the BMM is not perfect either, and it’s difficult to definitively say whether the FMM or BMM performs better.

3) Bidirectional Maximum Matching Algorithm

By applying both the FMM and BMM to segment certain texts, we see that sometimes forward matching performs better, and sometimes backward matching is superior. There are also cases where neither algorithm successfully resolves the ambiguity.

According to Professor Sun Maosong from Tsinghua University, in about 90% of Chinese sentences, the segmentation results of the FMM and BMM are identical. However, in 9% of sentences, the two algorithms produce different results, and one of these results is always correct. Only in 1% of cases do both algorithms fail to produce a correct segmentation result.

This raises the question: is it possible to design a rule-based strategy that selects the correct segmentation result from the two algorithms? In response, the BIMM [10] was proposed, combining both FMM and BMM with the following strategy:

a) Perform both FMM and BMM. If the number of words differs, return the result with fewer words.

b) If the word counts are the same, return the result with fewer single-character words.

c) If the number of single-character words is also the same, prioritize the result from BMM.

The results indicate that while the BIMM successfully selects the best result in certain cases, it chooses incorrect results in others. As a result, its overall accuracy is sometimes even lower than that of the BMM. Therefore, rule-based segmentation algorithms are fragile and cannot guarantee an optimal result, the end result is just robbing Peter to pay Paul.

IV. Experimental Result and Analysis

Dictionary-based Segmentation Algorithms are limited by the dictionary.

A. Naive Implementation

The performance of the four segmentation algorithms was tested under the naive implementation.

1) Definition of Test Methods and Test Cases

a) Speed Test Method

The speed test method is shown in Fig. 12.

```
public static void evaluateSpeed(){
    start = System.currentTimeMillis();
            for (int i = 0;i < pressure;i++){
                back_longest_seg(text,dictionary);
            }
            costTime = (System.currentTimeMillis() - start) / (double)1000;
            System.out.println("[illegible]" + text.length() * pressure / 10000 /
                               costTime +"万字/秒\n");
}
```

Figure 12. Speed test method

First, use the *System.currentTimeMillis* method to record the program's start time and store it in the variable *start*. Then, use a *for* loop to repeatedly perform the segmentation operation *pressure* times. Within the loop, call the *back_longest_seg* method to segment the text *text*.

After the loop completes, record the end time and calculate the total time taken for the entire segmentation process, *costTime*, in seconds. This is calculated by subtracting the start time from the end time and dividing by 1000.

Finally, print the segmentation speed (i.e., the number of characters processed per second). The speed is calculated as *(text.length * pressure) / costTime / 10000*, with the result expressed in units of “ten thousand characters per second”.

b) Test Case

Text = “西安工业大学计算机科学与工程学院”.

Pressure = 100,000 iterations.

2) Test Results

The test results are shown in Fig. 13.

The results of these tests demonstrated that, under naive conditions, the FMM outperformed the other algorithms in terms of speed. While the

BMM showed competitive performance, the BIMM was slightly slower due to the overhead of combining both forward and backward passes. The Full Segmentation Algorithm was the slowest, given its need to evaluate all possible word combinations, significantly increasing its computational complexity.

Under these circumstances, the time complexity is O (log n).

B. Trie Implementation

We tested the performance of the Full Segmentation Algorithm and the FMM using the Trie structure.

1) Defining the Test Method and Test Cases

The testing method and test cases for this structure are similar to those used in the naive implementation.

2) Test Results

A Comparison of test results between the two structures is shown in Fig. 14.

C. DAT Implementation

The performance of the Full Segmentation Algorithm and the FMM were tested using the DAT structure.

1) Defining the Test Method and Test Cases

The testing method and test cases for this structure are similar to those used in the naive implementation.

2) Test Results

The test results of this method are shown in Fig. 15.

The comparison results of the three cases are shown in Fig. 16. The results showed a significant improvement in both algorithms' performance compared to their naive implementations. The DAT structure achieves constant-time state transitions. This reduces overall computational complexity, especially for FMM, where segmentation requires fewer comparisons and faster state transitions.

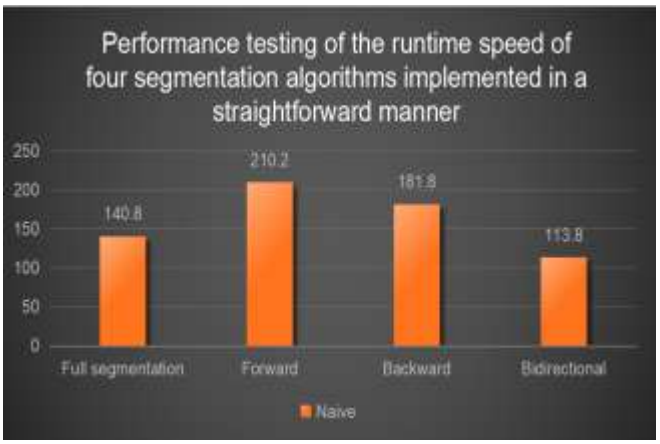

Figure 13. Test results（Naive）

Figure 14. Test results（Trie）

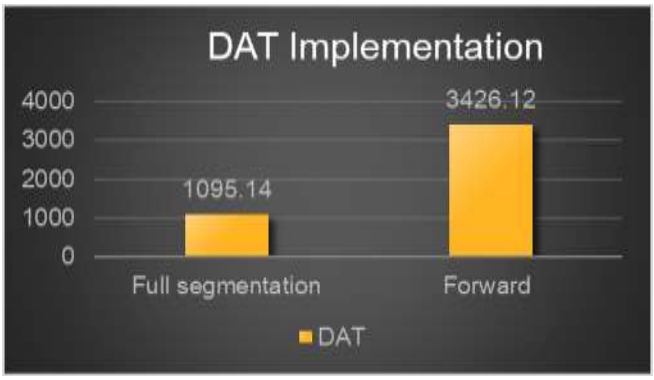

Figure 15. Test results（DAT）

During the state transition process, the time complexity approaches O (1). This is because, in the DAT structure, once the initial state is determined, the transition between states involves only a constant number of operations, such as index calculation and comparison. This efficiency makes the DAT highly suitable for large-scale

word segmentation tasks, as it ensures consistent performance regardless of the input size.

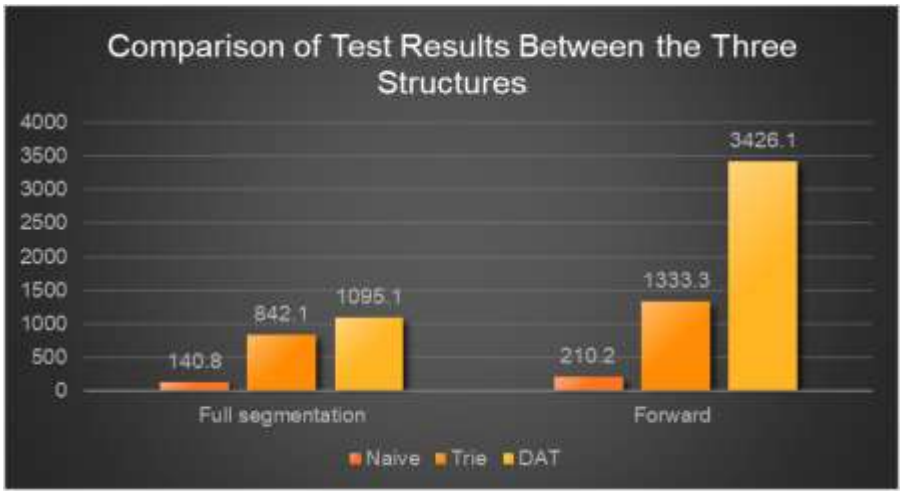

Figure 16. Comparison of test results between the three structures

V. Conclusions

The segmentation algorithms themselves are not inherently complex. The key to leveraging the natural advantages of fast segmentation lies not in the segmentation process itself, but in the data structure that supports the dictionary. By optimizing the data structure, the efficiency of string matching improves by orders of magnitude.

The DAT achieves constant-time complexity for state transitions, though the algorithm still has limitations:

When performing Full Segmentation on text of length *n*, the complexity can degrade to $O(n^2)$. This is because during full segmentation, the starting point constantly shifts to discover new matches. For example, suppose the dictionary contains all Arabic numerals. Scanning the text "123" results in 6 state transitions: 1, 12, 123, 2, 23, 3. Extending this to the text "123...n," the total number of state transitions becomes $n + (n-1) + \ldots + 1 = n(n+1)/2 = O(n^2)$. An Aho-Corasick Automaton can optimize the DAT by performing only a single scan to find all matches.

On the other hand, it is noted that dictionary-based segmentation heavily relies on the dictionary itself, leading to poor disambiguation and limited ability to recognize new words. Today, in the field of NLP, deep learning-powered statistical models are more prevalent in segmentation algorithms.

References

[1] Pak I, Teh P L. Text segmentation techniques: a critical review [J]. Innovative Computing, Optimization and Its Applications: Modelling and Simulations, 2018: 167-181.

[2] Liu C, Zhang Q, Feng J, et al. A Chinese word segmentation method based on dictionary and HMM [C]//Proceedings of the 2022 6th International Conference on Electronic Information Technology and Computer Engineering. 2022: 644-649.

[3] Sugahara R, Nakashima Y, Inenaga S, et al. Efficiently computing runs on a trie [J]. Theoretical Computer Science, 2021, 887: 143-151.

[4] Yeasin Emon R, Chanda Tista S. An Efficient Word Lookup System by using Improved Trie Algorithm [J]. arXiv e-prints, 2019: arXiv: 1911.01763.

[5] Bannai H, Goto K, Kanda S, et al. NP-Completeness for the Space-Optimality of Double-Array Tries [J]. arXiv preprint arXiv:2403.04951, 2024.

[6] Piedeleu R, Zanasi F. A String Diagrammatic Axiomatisation of Finite-State Automata [C]//FoSSaCS. 2021: 469-489.

[7] Scheibel W, Limberger D, Döllner J. Survey of treemap layout algorithms [C]//Proceedings of the 13th international symposium on visual information communication and interaction. 2020: 1-9.

[8] Pei J. A dictionary-based maximum match algorithm via statistical information for Chinese word segmentation [J]. International Journal of Electronics and Information Engineering, 2020, 12(1): 24-33.

[9] Li R. English Translation Intelligent Recognition Model Based on Reverse Maximum Matching Segmentation Algorithm [C]//International Conference on Innovative Computing. Singapore: Springer Nature Singapore, 2023: 342-349.

[10] Yan X, Xiong X, Cheng X, et al. HMM-BiMM: Hidden Markov Model-based word segmentation via improved Bi-directional Maximal Matching algorithm [J]. Computers & Electrical Engineering, 2021, 94: 107354.