A Data Structure Using Hashing and Tries For Efficient Chinese Lexical Access *

Yat-Kin LAM and Qiang HUO
Department of Computer Science, The University of Hong Kong, Hong Kong, China
(Email: h9914679@graduate.hku.hk, qhuo@cs.hku.hk)

Abstract

A lexicon is needed in many applications. In the past, different structures such as tries, hash tables and their variants have been investigated for lexicon organization and lexical access. In this paper, we propose a new data structure that combines the use of hash table and tries for storing a Chinese lexicon. The data structure facilitates an efficient lexical access yet requires less memory than that of a trie lexicon. Experiments are conducted to evaluate its performance for in-vocabulary lexical access, out-of-vocabulary word rejection, and substring matching. The effectiveness of the proposed approach is confirmed.

1. Introduction

A lexicon is needed in many applications such as contextual processing in OCR, handwriting recognition, and speech recognition; spelling correction; information retrieval; computer games; and many others. Finding a good organization of the lexicon to address different application needs has been a topic of extensive researches for several decades. Popular data structures include tries, hash tables and their variants (e.g. [1, 3, 4]).

Although ideas behind these techniques are quite general, some data structures that are suitable for western languages like English with just 26 letters may not be suitable for oriental languages like Chinese with more than 5000 characters. For example, variants of 26-way tree [7] and compressed tries [4] are useful in English as only 26 flags are needed in a node to represent the 26 possible characters, but for Chinese, more than 5000 bits are needed for each node, which greatly increase the space required for storing the lexicon. Some other techniques have to be used. In [2], a specific trie data structure was used to store a Japanese lexicon. Each node in the lexicon tree consists of a character code and a pointer to a *child node pointer table* which is a variable size table storing the pointers to the child nodes. One of the advantages of trie is that it allows fast access to entries with common prefixes, which is very important when searching similar entries. One of the disadvantages is that if approximate string matching is performed, an assumption has to be made that the first letter of the query is correct, otherwise the whole lexicon will be searched.

In another recent attempt of finding a good data structure to store a Chinese lexicon [6], a hash table is used to store all 2-character prefixes in the lexicon. The advantage of such a table is that it allows a fast access to 2-character entries, and entries with the same 2-character prefixes, that indeed account for a large portion of entries in a typical Chinese lexicon.

By combining the advantages of trie and hash table as described above, a new data structure is proposed in this paper to store a Chinese lexicon. In the following, we describe the detail of the structure and show how it works in a series of experiments.

2. Our Approach

2.1. The Data Structure

The proposed structure consists of two tables and a number of tries. Entries in a Chinese lexicon is put into the structure according to their length as follows:

- All single character words are stored in a table of 32768 bins. Since the GB code of a character takes 2 bytes, and the bit 16 is always set to 1, bits 1-15 are used as the bin address for the word. Each bin corresponds to a 2-byte value which may or may not be a valid Chinese character. Therefore each bin requires a *valid bit* to indicate whether it represents a valid single-character Chinese word.

- All 2-character words are stored in a separate table of 65536 bins. The bin address is calculated from GB codes of the 2-character word according to a predefined addressing scheme. Should there be other words contesting for the same bin, they are kept in a linked

*This work was supported by a grant from the RGC of the Hong Kong SAR (Project No. HKU7145/03E).

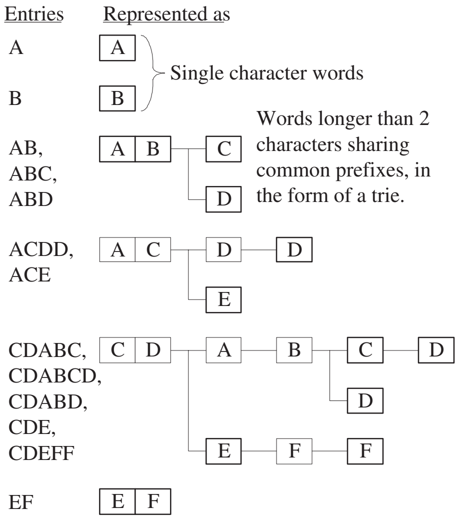

Figure 1. Managing entries of different kinds for the proposed structure.

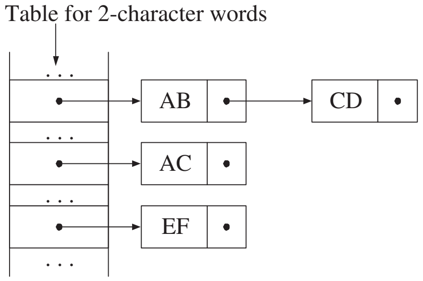

Figure 2. Linked list of entries in the table for 2-character words in the proposed structure.

list. The list can be sorted according to usage frequency of the words, which can improve the performance in practice. Note that we keep the number of bits used for each address within 16 bits, so that at most 2 bytes are needed to store each bin address. We will discuss and compare several hashing schemes in the experiment section.

- Any word longer than 2 characters will be divided into a 2-character prefix and a suffix for the rest of the word. The prefix is stored in the bin table for 2-character words. Common prefixes share the same node, with all possible suffixes forming a trie linked to the prefix.

Fig. 1 illustrates how different entries are managed by using a 13-word lexicon (A, AB, ABC, ABD, ACDD, ACE, B, CDABC, CDABCD, CDABD, CDE, CDEFF, EF) as an example. As the 2-character patterns are stored in the table for 2-character words, collisions are resolved by using a linked list. In this example, the prefixes to be stored are “AB”, “AC”, “CD” and “EF”. Suppose “AB” and “CD” contest for the same bin, the table for 2-character words will have the entries as shown in Fig. 2.

Table for single character words

A value of 1 means the character is a valid entry

. . .	A	B	C	D	E	F	. . .
. . .	1	1	0	0	0	0	. . .

Table for 2-character patterns

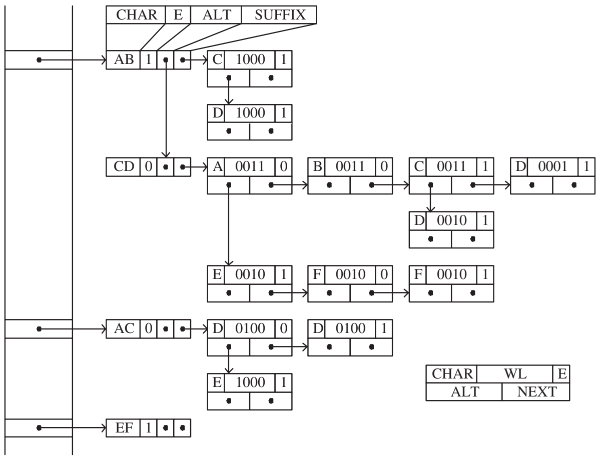

Figure 3. Using the proposed data structure to store a 13-word lexicon.

Apart from nodes for single-character words, there are other two types of nodes to be used in the structure: the nodes for 2-character patterns and the trie nodes. Each node for 2-character pattern has the following four fields

- **CHAR**: stores GB codes of 2-character word;
- **E**: ending bit, sets to 1 if the 2-character pattern is a valid word;
- **ALT**: points to an alternative node, which contests the same bin in the table of 2-character words;
- **SUFFIX**: points to a trie of suffixes.

The trie node is the same as the one used in a trie structure reported in [5]. By using the above 13-word lexicon as an example, the proposed data structure to store the lexicon is shown in Fig. 3.

2.2. Exact Matching

To search a word from the structure, different steps are used for entries of different lengths:

- To search for a 1-character word, the table for single character words is checked to see if the word is valid.

- To search for a 2-character word, the table of 2-character words is accessed. The 2-character word is searched along the linked list at the bin, following the ALT link of each node, until a match is found. If none of the nodes matches the 2-character word or the bin in the table is in fact empty, the search fails. Otherwise if a node is found, the E field of the node is checked: if it is 1, the word is found, else the search fails.

- To search for a word longer than 2 characters, the table of 2-character words is accessed, finding the 2-character prefix of the word the same way as finding a 2-character word, except that the E field will not be checked. If the 2-character prefix is not found, the search fails. Otherwise, the SUFFIX link of the node is followed to a trie of suffix; the remaining suffix of the word is searched within the trie, using a normal trie search technique (e.g. [5]).

2.3. Substring Matching

A substring matching algorithm will consider all possible substrings in the query. A straightforward method is to perform lexical access at each position of the query. For example, if the query is “abcd”, lexical access will be performed on sub-queries “abcd”, “bcd”, “cd” and “d”. In each lexical access, all prefixes will be considered. For example, in the process of lexical access for “bcd”, matches for “b”, “bc” and “bcd” will be found. In this way, all substrings in the query will be matched.

In the proposed data structure, sub-queries are searched in a multiple step process. For example, in substring matching on the sub-query “bcdefgh”, the first character “b” is checked against the table for single character words to see if it can be a valid word. Next, the first two-character word “bc” is searched in the table for 2-character words. If it can be a word, it is put in the return list. If it can be a prefix, the search will be continued on the trie attached to the 2-character prefix. The suffix “defgh” will be looked up in the trie. In each of the matched node in the trie, the end-of-word flag is checked to see if the prefix can be a word. In this example, the nodes representing “d”, “de”, “def”, “defg” and “defgh” will be accessed.

3. Experiments and Results

3.1. Experimental Setup

In order to evaluate the performance of the proposed data structure, a Chinese lexicon consisting of 86995 words is used. The word length distribution of the lexicon is shown in Table 1. The average word length is 2.4 characters. Over 98% of the words are of length less than or equal to 4.

Table 1. Word length distribution in a Chinese lexicon used in experiments.

Total: 86995 words; Average length: 2.4					
Length	1	2	3	4	5
Entries	6123	54351	13536	11879	598
Length	6	7	8	9	10
Entries	292	139	66	3	8

To compare the performance of different data structures on exact matching and substring matching, the following three sets of experiments are carried out:

- The first experiment examines the performance of finding a query if the query is in the lexicon. The set of testing queries is in fact the whole lexicon (i.e. 86995 entries). The efficiency of such a basic operation affects the performance of many applications.

- The second experiment examines the performance of rejecting an invalid query if the query is not in the lexicon. As a good data structure, it should be able to reject invalid query efficiently so that applications or users can be alerted for the error and respond as soon as possible. The testing queries include 5000 invalid queries that are generated randomly according to the word-length distribution of the lexicon.

- The third experiment examines the performance of substring matching. The testing set consists of 5000 queries that are generated by concatenating randomly selected entries from the Chinese lexicon. Each query contains at least 2 valid lexical entries and at least 11 characters. The average length of queries is 11.9.

In the above experiments, the following three metrics are used for measuring the performance:

- the number of nodes accessed throughout the search,
- the number of character comparisons performed throughout the search, and
- the user CPU time taken to finish the search.

The CPU time is measured by running the experiments in a computer with a PIII-M 850MHz CPU (512k L2 cache) and 256MB SDRAM.

Table 2. A comparison of distributions of 2-character patterns using different hashing schemes.

	# of cells	Occupied cells (%)	Avg. chain len.	Max. chain len.
Greedy method, 16-bit address	65536	37977 (58%)	1.69	11
Greedy method, 15-bit address	32768	26600 (81%)	2.41	15
Greedy method, 14-bit address	16384	15663 (96%)	4.09	18
Division method, table size 49157	49157	26455 (54%)	2.42	16
Division method, table size 57347	57347	33216 (58%)	1.93	15
Division method, table size 61441	61441	28133 (46%)	2.27	16

Table 3. A performance comparison of the proposed data structure using different hashing schemes.

	Exact matching, valid entries			Exact matching, invalid entries			Substring matching		
Hash Function	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)
Greedy, 16-bit	2.2	3.8	0.00176	1.2	2.3	0.00185	30	35	0.00840
Greedy, 15-bit	2.8	4.8	0.00178	2.3	4.5	0.00192	41	53	0.00876
Greedy, 14-bit	3.8	6.8	0.00182	4.3	8.7	0.00199	58	87	0.00908
Div., 49157	2.8	4.8	0.00196	2.3	4.5	0.00199	42	56	0.00972
Div., 57347	2.6	4.3	0.00195	1.8	3.5	0.00198	37	44	0.00971
Div., 61441	2.9	5.0	0.00197	2.4	4.8	0.00202	41	53	0.00987

3.2. Effects of Different Hash Functions

In our approach, the hash function to calculate the bin address in the table of 2-character words is a very basic function for accessing the data structure. Such a function ideally should 1) be as simple as possible, otherwise a big overhead may be incurred, and 2) have a collision rate as low as possible, otherwise more nodes need be searched. We have tried the following two approaches:

- To choose n bits out of the 32 bit values of the GB codes of a given 2-character pattern as the bin address. For a given lexicon, a greedy algorithm is used to select n bit positions that lead to the most even distribution of 2-character patterns in the lexicon for each selected bit. In our experiments, three cases, $n = 16$, $n = 15$, $n = 14$, are tested.

- To use the division method by choosing an appropriate table size. The table size will be at most $2^{16} = 65536$. Using this method, three prime values, 49157, 57347 and 61441, are tested.

The distributions of 2-character patterns using the above methods are summarized in Table 2. Using the greedy approach, it is observed that the proportion of occupied bins increases as the number of address bits decreases. However at the same time, the average chain length of the occupied cells increases. With these different hash functions, the performance of the proposed structure in exact matching and substring matching is evaluated. The results are summarized in Table 3. It is observed that the performance depends much on the average chain length of the occupied cells. For example, using a 16-bit address, the average chain length is the shortest while the number of nodes accessed and number of character comparisons are the fewest. When less number of bits are used, the performance drops although the proportion of occupied cells increases. As for the division method, it is observed that with a table size of 57347, the structure performs the best among the three table sizes. There may be other table size that will give a better result, but the best setup can only be found by trial and error for every different lexicon.

From the above results, we recommend to use the hashing scheme of 16-bit bin address derived by the greedy approach. For the Chinese lexicon we are using, those 16 bits are bit 1, 2, 3, 4, 5, 9, 10, 11 of the GB code of both characters.

3.3. A Comparison with Other Data Structures

The performance of the proposed structure is also compared with that of other data structures in exact matching and substring matching. Structures compared include the sequential list, trie, and the structure used in [6]. Sequential list and trie are the basic and common structures used

Table 4. A performance comparison of different data structures.

	Exact matching, valid entries			Exact matching, invalid entries			Substring matching		
Data Structure	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)	Avg. # of nodes accessed	Avg. # of char. comp.	CPU time per query (in ms)
Seq. list	15.5	21.0	0.00248	16.5	21.8	0.00325	260	340	0.0439
Trie	16.3	16.3	0.00211	16.1	16.1	0.00301	190	190	0.0160
Struct. in [6]	5.3	10.2	0.00194	7.8	15.5	0.00229	87	148	0.0117
Proposed	2.2	3.8	0.00179	1.2	2.3	0.00184	30	35	0.0084

Table 5. A comparison of memory requirement of different data structures for a Chinese lexicon.

Data Structure	Sequential list		Trie	Structure	Proposed
	fixed width array	index table	(binary search)	in [6]	structure
Memory Requirement	1.7MB	660KB	770KB	760KB	730KB

in many applications, while the structure used in [6] is an existing structure designed to store Chinese lexicon in literature. Table 4 summarizes the experimental results. Binary search is used in sequential list searching. It is also used in searching in the trie structure among alternative nodes. It is observed that the proposed structure achieves the best performance in terms of the abovementioned three metrics in all cases. The proposed structure can reject invalid queries very efficiently because the table of 2-character patterns acts as a hash table. Although trie does not help much on processing a single query in comparison with sequential list, it does help a lot on substring matching because common prefixes among substrings of the query are matched at the same time.

Table 5 compares the memory requirement of different data structures estimated by using our implementations for the Chinese lexicon mentioned previously. It is noted that there are two possible ways to store a lexicon as a sequential list. The first method is to store the lexicon as an array with fixed width. The other method is to store the entries sequentially. Then a lookup table is used to track the position of all entries. The proposed data structure requires less memory than that of a trie.

4. Discussions and Conclusion

In this paper, we have proposed a new data structure that is suitable to store a Chinese lexicon. It combines the use of tries and hash table and takes advantages of both approaches in storing Chinese lexicons. A simple hashing scheme is suggested for the hash table after comparing it with some other methods. Experimental results show that the proposed structure allows a fast access of in-vocabulary entries, a fast substring matching as well as an efficient rejection of out-of-vocabulary entries. The structure will be a good choice for storing Chinese lexicons in many applications as mentioned in the beginning of the paper.

References

[1] E. Fredkin, "Trie Memory,", *Communications of the ACM*, Vol. 3, No. 9, pp.490-499, 1960.

[2] M. Koga, R. Mine, H. Sako and H. Fujisawa, "Lexical Search Approach for Character-String Recognition," in *Document Analysis Systems: Theory and Practice*, S.-W. Lee and Y. Nakano (Eds.), pp. 115-129, 1999.

[3] D. E. Kunth, *The Art of Computer Programming: Vol. 3, Sorting and Searching*, 1973.

[4] K. Maly, "Compressed Tries," *Communications of the ACM*, Vol. 19, No. 7, pp. 409-415, July 1976.

[5] S. N. Srihari, J. J. Hull and R. Choudhari, "Integrating Diverse Knowledge Sources in Text Recognition," *ACM Transactions on Office Information Systems*, Vol. 1, pp. 68-87, January 1983.

[6] P. K. Wong and C. Chan, "Postprocessing Statistical Language Models for a Handwritten Chinese Character Recognizer," *IEEE Trans. on Systems, Man, and Cybernetics – Part B: Cybernetics*, Vol. 29, No. 2, pp.286-291, 1999.

[7] C. J. Wells, L. J. Evett, P. E. Whitby, R. J. Whitrow, "Fast Dictionary Look-up for Contextual Word Recognition," *Pattern Recognition*, Vol. 23, No. 5, pp. 501-508, 1990.