A Structure-Shared Trie Compression Method

Thanasan Tanhermhong **Thanaruk Theeramunkong** **Wirat Chinnan**
Information Technology program, Sirindhorn International Institute of technology,
Thammasat University
P.O Box 22 Thammasat Rangsit Post Office,
Pathumthani 12121, Thailand
thanasan@kind.siit.tu.ac.th ping@kind.siit.tu.ac.th winnie@kind.siit.tu.ac.th

Abstract

Trie, a well-known searching algorithm, is a basic and important part of various computer applications such as information retrieval, natural language processing, database system, compiler, and computer network. Although a merit of trie structure is its searching speed, the naïve version of trie structure size may not be acceptable when the key set is very large. To reduce its size, several methods were proposed. This paper proposes an alternative approach using a new trie structure called, *structure-shared trie* (SS-trie). The main idea is to reduce unused space using shared common structure and bit compression. Three techniques are used: (1) path compression, (2) structure sharing, and (3) node compression. A number of experiments are conducted to investigate trie size, sharing rate, and average depth of the proposed method in comparison with binary search, naïve trie, PAT, and LC-trie. The result shows that our method outperforms other methods.

1 Introduction

Trie structure is a data structure frequently used for searching a datum from sources of data, such as natural language dictionaries, databases, tables in compilers and so forth. One of the merits of trie structure is its searching efficiency. The searching time in trie structure depends only on the length of keyword, while that of other structures involves the number of words (or entries). However, a problem of an original trie is that it wastes a lot of space. Some memories in the trie is reserved even though they are not used.

According to trie uses an array for fast accessing to the needed data, the size of array in trie is fixed to the maximum possible numbers of alphabets (or cardinality of alphabets). This introduces a lot of explicit useless space for a large cardinality of alphabets through small average branching factor. In order to solve this problem, the cardinality of alphabets is reduced to two for binary trie, ten for 10-ary trie, and so forth. Although the problem of explicit useless space is reduced, the number of nodes and transitions are increased, which leads to slow searching speed.

During the past few decades, a number of ideas were proposed to reduce the size of trie structure such as PAT (Morrison 1968), Double Array Trie (Aoe 1989), and LC-trie (Nilsson and Karlsson 1999, Zandieh 1999). However, it is still in a question whether there is more efficient compression method than these approaches. The idea of reducing branching factor of trie may have other ways to solve the space usage problem. In this paper, we propose a new idea called structure-shared trie (SS-trie) to reduce the size of trie structure and preserve quality of retrieval speed.

The SS-trie is an overhead-reducing version for a 256-ary trie. It uses the combination of three techniques: (1) path compression, (2) structure sharing, and (3) node compression to solve trie problem. SS-trie uses the path compression technique to discard the node with only one child. Then the second technique, Structure sharing, is applied to solve explicitly useless space by sharing frequently used node structure. For unshared structure, SS-trie uses the node compression to compress explicitly useless space into bit vectors. With the combination of these techniques, it can reduce size of trie structure more than naïve trie structure but still preserving the quality of searching speed.

For the rest of this paper, section 2 explores some previous works related with trie compression. Section 3 illustrates the main idea of SS-trie. In section 4, the effectiveness of SS-trie is shown through a set of experiments. Finally conclusion and discussion are given.

2 Previous Works

Searching algorithm is a basic and important operation for computer applications. In the past, a lot of searching algorithms were introduced. It can be classified into two types: indexing approaches and non-indexing approaches. Non-indexing approaches are slow, but there is no need to have a preprocess for indexing and indexing space are not requited. Efficiency of non-indexing approach is measured from searching speed. For indexing approaches, they give faster searching speed, but need a indexing process and indexing space. Efficiency of indexing approach is measured based on both searching speed and indexing space.

Trie is a type of indexing algorithm. The indexing algorithms normally use the concept of search tree (and hashing). For instance, inverted file algorithm (Harman, Fox, Baeza-Yates, and Lee 1992) uses one or two levels of a search tree. Naïve trie (De La Briandais 1959, Fredkin 1960) is a completed search tree. Searching speed normally depends on appropriate level or the depth of search tree. There is a trade off between searching speed and indexing space.

There were some well-known algorithms to reduce the size of trie such as bst trie (Clampett, 1964), L trie (Ramesh, Babu, and Kincaid 1989), ternary trie (Bentley and Sedgwick 1997), two-trie (Aoe, Morimoto, Shishibori, and Ki-Hong 1996), DAWG, DAT (Aoe 1989), PAT (Morrison 1968). These algorithms can roughly be classified to five main techniques.

The first technique is a combination of trie and binary search such as bst trie and L-trie. For *bst trie (binary search tree)*, binary search is used to search a child from sorted and contiguous children in each node in a trie. The merit is no waste of space but searching speed depends on the number of entries and keyword length. For *L-trie,* binary search is used instead of some sub trees in trie that have low children density. L-trie can improve both searching time and trie size (indexing size). Using binary search can reduce much on trie size but this method is slower than an original trie in the case that the needed data does not exist in data entries.

The second technique is to decrease an average unused space per node by discarding some low children density nodes such as PAT. *PAT (PATRICIA: practical algorithm to retrieve information coded in alphanumeric)* tries to get rid of unused space per node by changing cardinality of alphabets to 2 and discarding nodes whose branching factor is 1. Now the branching factor is exactly 2. On the other hand, PAT is similar to a perfect binary tree. PAT has an important property that number of nodes is exactly equal to number of entries minus 1.

The next technique is to adjust larger appropriate cardinality of alphabets for a trie or branching factor of each node. The examples of this type are binary search tree and LC-trie. *Binary trie* is a trie with cardinality of alphabets is 2. For binary trie, the size is smaller but the speed is slow down because of a trade-off between branching factor and depth. *LC-trie (level-compressed trie)* is an improved version of PAT. It improves PAT by changing branching factor of some nodes if some certain conditions are satisfied. LC-trie is faster than PAT because LC-tries increase branching factor and then the depths of trie become shallow. LC-trie size is smaller than PAT although branching factor increases.

Another technique is used in a well-known *DAT, double array trie*, uses the idea of sharing children between parents. Searching speed of DAT is quite, good but DAT construction is difficult to optimize. The last technique is to divide trie into two or more parts, using the appropriate data structure for each part. The example of this technique is two-trie. *Two-trie* divides a trie into the front part and the rear part.

There are other techniques for reduce usage space of trie such as lst trie, and DAWG. *Lst trie* is a trie that use list instead of array. Its size is smaller but the searching speed is very slow. DAWG is a trie that uses the concept of graph. It can share all transitions for both front and rear, respectively. In general, a DAWG has states with more than one outgoing transition after a state with two or more incoming transitions. Thus, there is no guarantee that can determine information for each key correctly.

3 Structure-Shared Trie

SS-trie is an overhead-reducing version for 256-ary trie using three techniques: (1) path compression, (2) structure sharing, and (3) node compression. The main idea is to reduce the unused space using sharing and compression of the explicit useless space into bits. SS-trie has a parameter i to identify level of the depth of sharable structure. We uses SS(i)-trie to represent SS-trie with the ith level of the depth of sharable structure. In the rest of paper, we use a term SS-trie for SS(1)-trie. When this structure combines with those three techniques, SS-trie can reduce trie structure size more than other algorithms. The detail of each technique is explained in the following section.

3.1 Path Compression

The first technique, so called *path-compression* (Morrison 1968), is a well-known technique that can increase retrieval speed and reduce trie structure size by removing internal nodes with only one child. From Fig 1, the circle is represented as an internal node and the rectangle is represented as a leaf or an external node. The internal nodes with only one child are shown in the shadow box. These nodes can be skipped while searching because there is only on way in the path. The number of nodes that have been skipped on each path is stored as the skipped value in the corresponding node as show in Fig 1.b. The path-compression binary trie is also known as PAT.

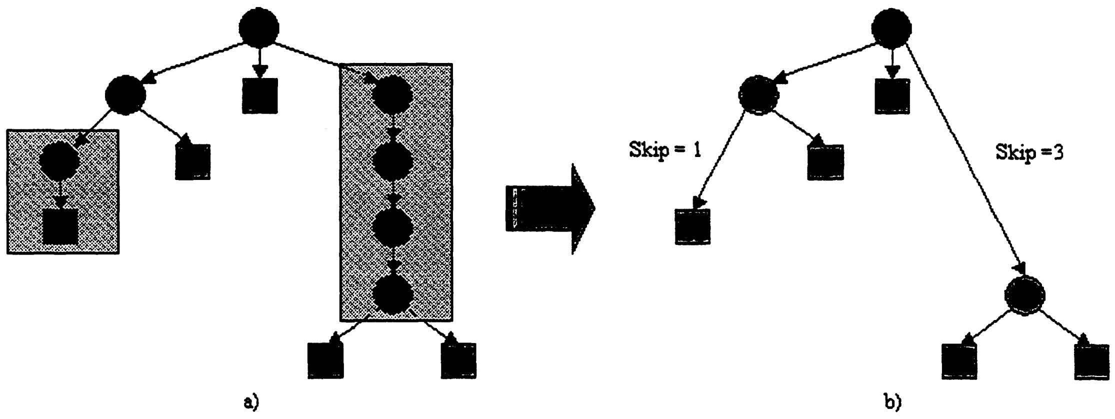

Figure 1. a) Naïve trie b) Path-compression version of trie showed in Fig 1.a.

3.2 Structure Sharing

Structure sharing, the second technique, is to share common structures in a trie.

3.2.1 General Concept

Structure sharing separates trie into 2 parts: (1) shared part and (2) unshared part. The *shared part*, called SP, is ith depth-limit tree structure for SS(i)-trie. It is used to store a frequently used structure in the trie as shown in Fig 2.b. A technique, applied to increase sharing rate is applied to SP in order to achieve smaller size. The details of this technique are described in the next section. The *unshared part*, called USP, is a list-trie like structure that each node stores its children with/without its structure identifier (id) as shown in Fig 2.a.

To illustrate structure-sharing concept, Figure 2 shows a SP and USP part of SS(1)-trie of a trie structure as shown in Fig 1. Assume that the keyword "bbc" is retrieved. Initially, we are starting at "root" of the USP part as shown in Fig 3.a. In this node, its structure id is 0. This means that there is sharing structure in the SP part as shown in Fig 3.b. The 0th structure identifies that "b" means the second link of "root" to "node 1". Here, the structure id is 0 again. Then the next character "b" is the second link of "node 1" to "node 2". From node 2, its structure has no id. This means that this structure is a unique structure (or unshared structure). The last character "c" means the transition from node 2 to node 3. Finally, the string "bbc" is found at node 3.

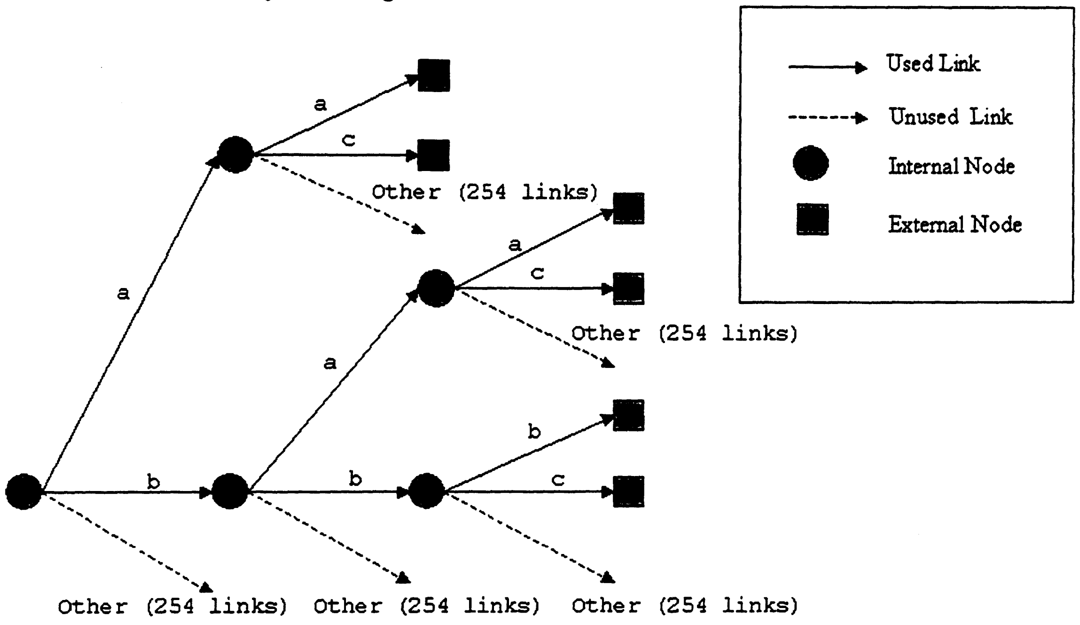

Figure 2. Unused spaces of naïve trie structure.

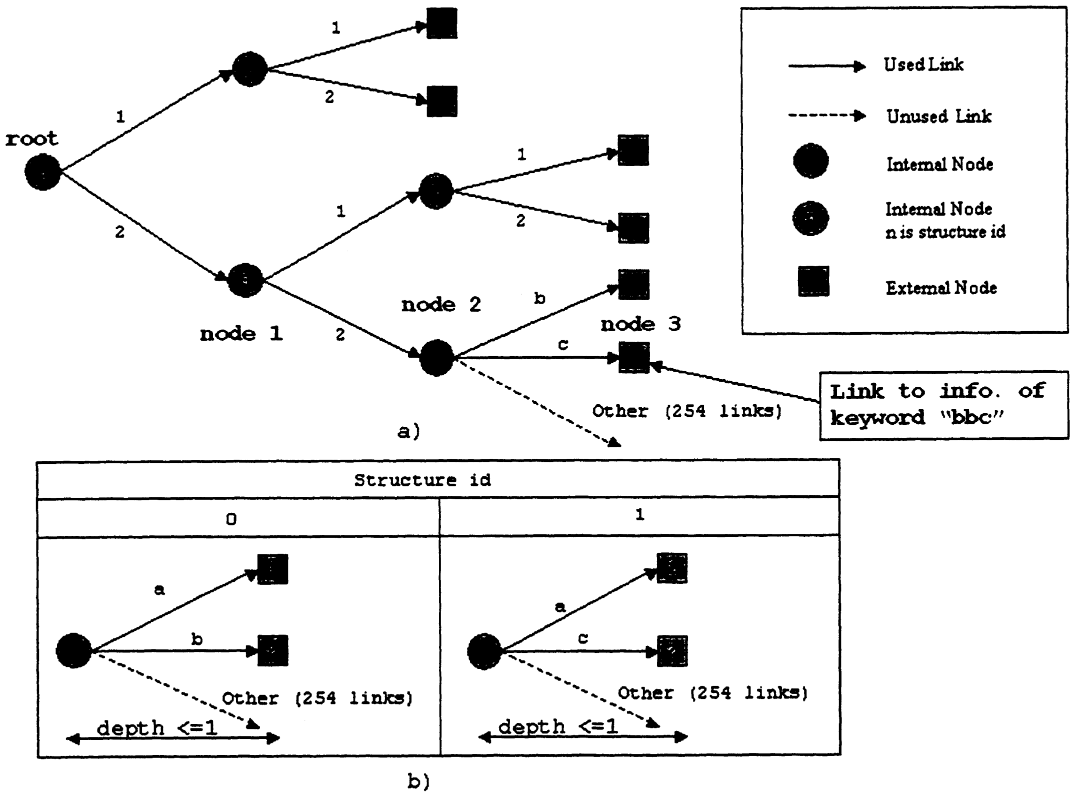

Figure 3. A SS(1)-trie of trie showed in Fig 2. a) USP part b) SP part

3.2.2 Sharing Rate Improvement

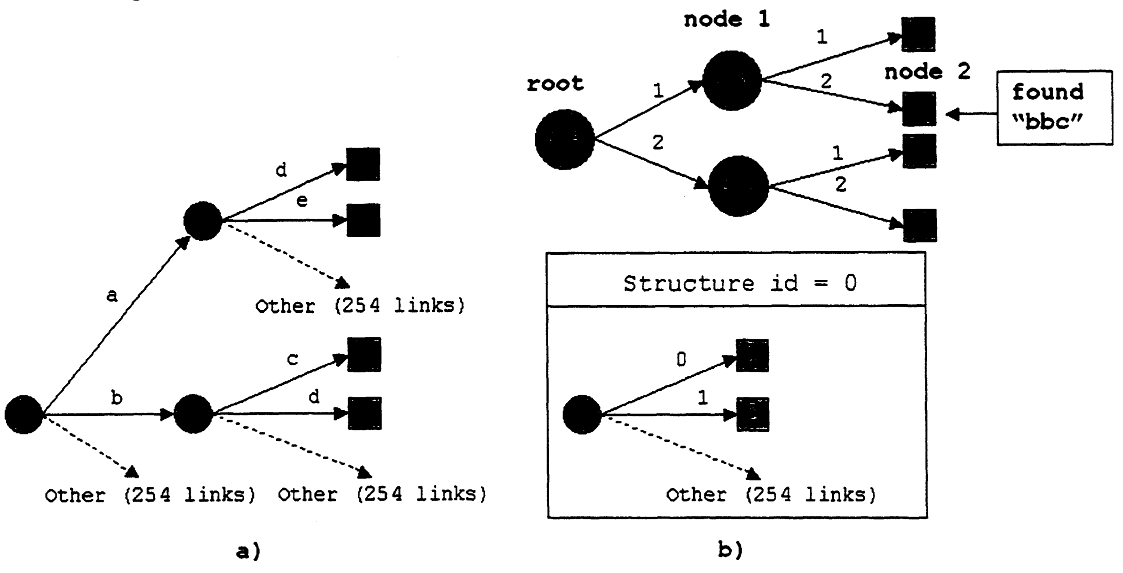

Figure 4. Sharing Rate Improvement using different between transitions.

From the investigation of data, there are many structures similar to Fig 4.a. These kinds of data have some similarities such as that the difference between each transition results in the same value. If normal structure sharing technique is applied, there is no sharing between these kinds of data. Therefore, the relative transition technique is used to improve the sharing rate as show in Fig 4.b. As the keyword “ae” is retrieved, we are starting at “root” as shown in Fig 4. In this node, its structure id is 0 and starting alphabet is “a”. This means that there is sharing structure in the SP part. It calculates the path from the distant of that alphabet comparing with the path number given in the SP part. Therefore, the 0th path number of alphabet “a” is “a” and the 1st path number of alphabet “a” is “b”. From this, the 0th structure and alphabet “a” identifies the first link of “root” to “node 1”. Here, the structure id is 0 again and the alphabet is “d”. So it uses the same sharing structure in the SP part, then the next character “e” is the second link of “node 1” to “node 2”. From node 2, the string “ae” is found.

3.3 Node Compression

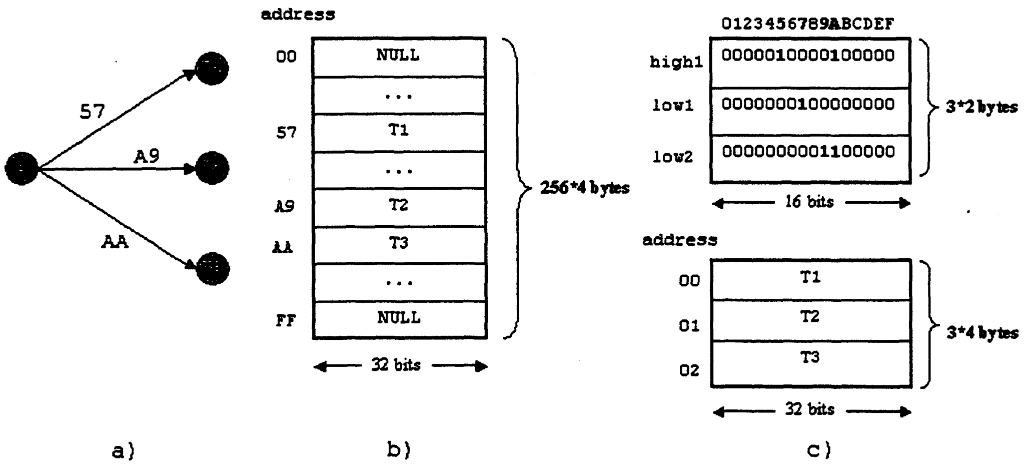

Figure 5. a) Tree representation of node S. b) Array representation of node S (1024bytes).
c) Bit vector representation of nodes S.

The last technique, *node compression,* is to compress four high bits and four low bits of the node transitions from Fig 5.a to bit vectors as shown in Fig 5.c. Initially, the processes of compression come from the trie structure in Fig 5.a and the array of its address in Fig 5.b. This array shows the transition between nodes via address mapping. From this Figure, there are three node transitions from S to T1, T2 and T3. The addresses of these transitions are 57, A9, and AA respectively. Then this technique compresses this array into a bit vector as seen in Fig 5.c. For high bits, 5 and A, is set to 1 in the "high1" in the Fig. After that, the low bit of 5, which is 7 is set in low1 and the low bits of A, which is 9 and A is set in the low2. From this compression, address location in Fig 5.c is calculated from the number of 1 prior to that transition; for example, transition AA to node T3 has 2 1's before it. The first bit comes from position 9 in low2 and the second bit comes from position 7 in low1. In this case, the address 02 keeps the transition node T3.

This technique can compress useless space into 4 to 34 bytes size of bit vectors and an average bit vector size is about 6 bytes.

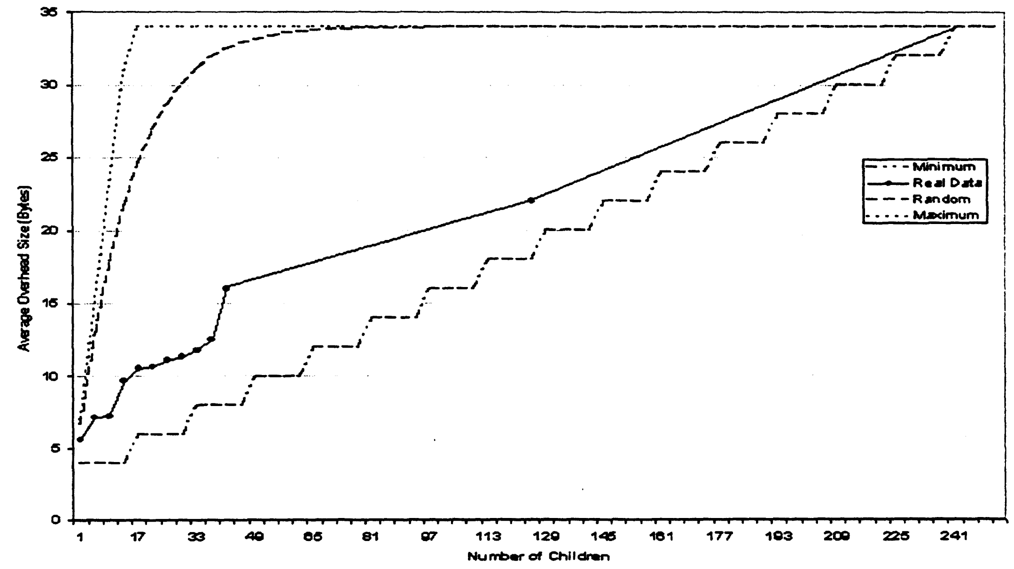

Figure 6. The effect of number of entries with overhead of node compression.

Figure 6 shows the overhead of SS-trie structure. There are minimum and maximum overhead in the Figure. For maximum overhead, it has two kinds of data: random data and normal data. Here, our approach is closer to minimum overhead than maximum overhead. In addition, it gets small increasement of overhead when the data set is growing. So our approach can preserve the searching speed as the naïve trie structure.

However, bit manipulation is very slow. So, in order to preserve quality of speed here, 64K-bytes or 32K-bytes (slower version) size lookahead table is applied. This lookahead table is a function to find the number of 1's before a certain position. The equation of this function is

$$f(x,y) = \sum_{i=15-y}^{15} \left(x \text{ and } \left(2^{i}\right)\right).$$

Where:

x is a 16 bits integer number.

y is a position indicator, range from 0 to 15.

For example $f((1011\underline{1}00101010001)_2,4) = 4$.

4 Experimental Results

To investigate the efficiency of SS-trie, three experiments are made. The word lists are automatically segmented and downloaded from Internet. They have total distinct words from the range of 1,000 to 100,000 words. The first experiment is made to evaluate effect of the number of entries on space complexity of SS-trie comparing with naïve trie, PAT, and LC-trie. The second experiment is for evaluating the effect of number of entries on the sharing rate of structure. The last experiment is made to study the effect of number of entries and keyword length effect on these trie. The next section describes the detail of each experiment.

4.1 Space Complexity

Through the trie structure, naïve trie is a n-ary trie but PAT and LC trie is a binary trie. So PAT and LC-trie yields the far better space reduction than naïve trie structure. Although our approach, SS-trie, is a 256-ary trie structure, it can reduce more space than the binary trie. The experiment is conducted to compare space complexity of trie structure. In this experiment, PAT, LC-trie, and SS-trie are compared together to investigate the memory consumption. This result of experiment shows the space complexity graph of trie structures in Fig 7.

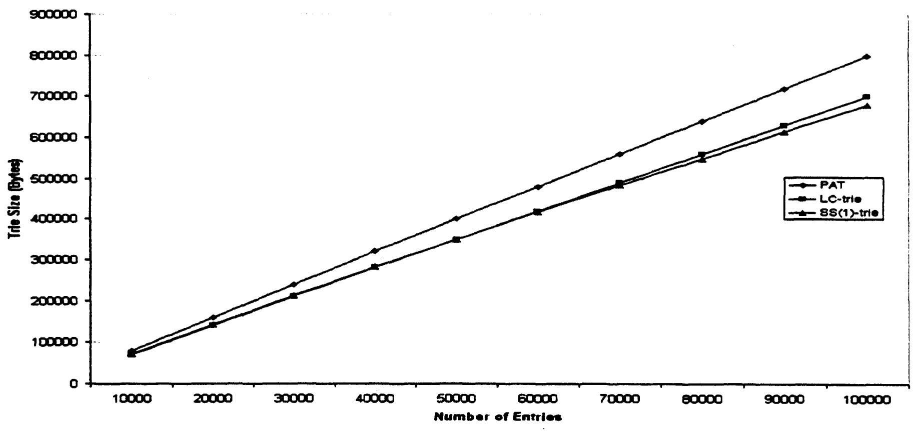

Figure 7. The evaluation of space complexity.

Fig 7 shows the memory consumption in creating the trie structure for PAT, LC-trie, and SS-trie. Here, SS-trie can reduce the useless space more than both PAT and LC-trie. For the small set of word lists, LC-trie is the best choice among experimental approaches. It can reduce the useless space more than other approaches. However, for the large set of word lists, SS-trie consumes less space than other approaches. From Fig 7, SS-trie beats the LC-trie when the word list is more than 70,000. The maximum size of structure using SS-trie is about 680,000 compared with PAT (800,000) and LC-trie (700,000). Therefore, the result concludes that SS-trie outperforms than other approaches for a large set of word lists.

4.2 Sharing Rate

From the word list of previous experiment, we can measure the sharing rate by using these 3 parameters: (1) Distinct Structures (2) Repeated Structure (3) Distinct Structure Among Repeated Structure. The first parameter indicates the total distinct structure in the word list. For the second structure is the total repeated structures, which have the same structures more than one in the word list. The last structure is the distinct structure of all repeated structure. This parameter is calculated from the total distinct structure, which retrieved from the repeated structure. For example, if "a, b, c, and d" is represented as structure, assume that there are the set of structures "aabbcd". The distinct structure from this set is 2, which comes from c and d. For repeated structure is 4 from two of a and two of b but the distinct structure among repeated structure is 2 from a and b. The result of the sharing rate of those 3 parameters is shown in the Fig 8.

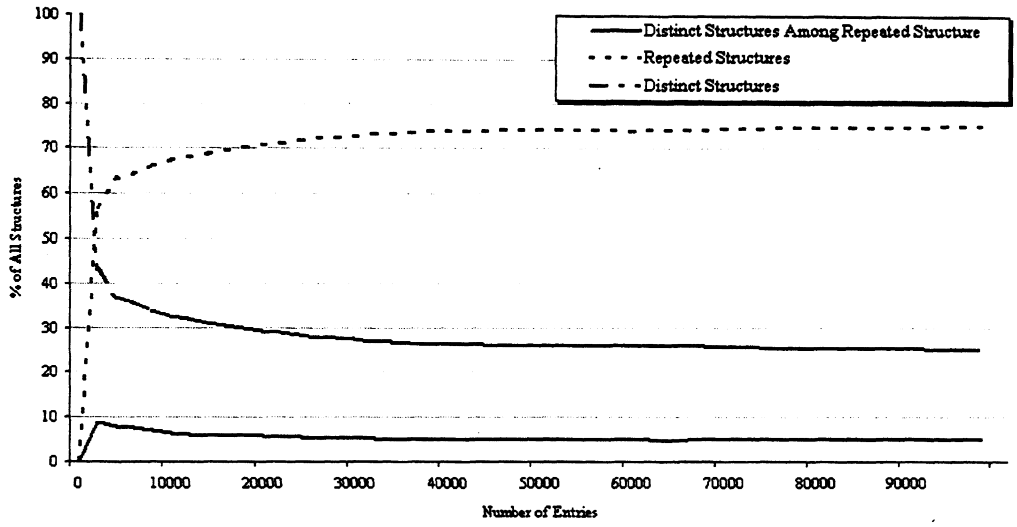

Figure 8. The Sharing rate graph.

In Fig 8, the graph of distinct structure is not proportional to the repeated graph. For the large set of word lists, the repeated structure is increasing while the distinct structure is decreasing. Thus, the sharing structure technique is work well for the large set of information. Moreover, the experimental result shows that the sharing structures are about 75% in the average case. With the information from graph, there are only 5% are totally distinction among the repeated structure. Therefore, our technique can compress the information from 75% to 5% and reduce the usage space of trie structure.

4.3 Average Depth

This experiment is conducted to compare the average depth of trie structure. In this experiment, PAT, LC-trie and SS-trie are compared together to investigate the average depth. The Fig 9 shows the graph of average depth of each approach.

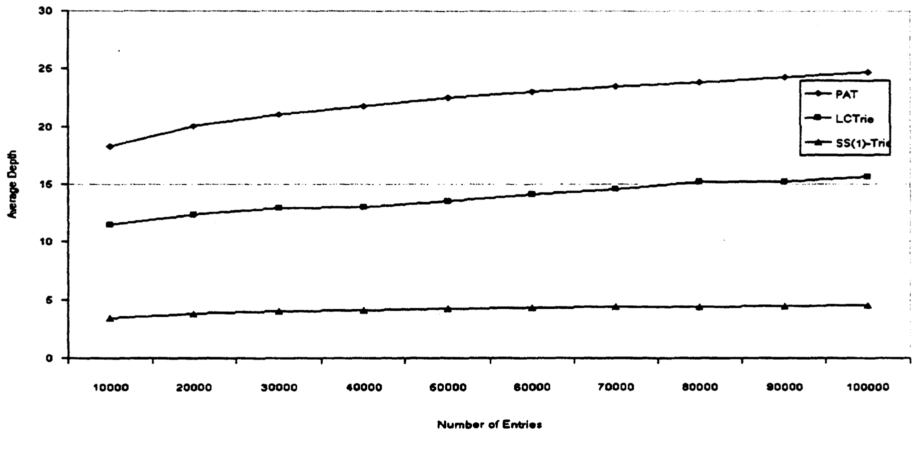

Figure 9. Average depth of each experimental approach.

The average depth of trie structure can be represented as the searching speed of trie structure. If trie structure has a large number of average depth, the searching speed of that trie is slow. From this graph, indicates that the average depth of SS-trie is 5 compared with PAT and LC-trie, which is 14 and 23 respectively. This result concludes that the searching speed of SS-trie is faster than both PAT and LC-trie. Therefore, our approaches can preserve the searching speed of trie structure.

5 Discussion

Our structure uses three techniques for compressing trie structure. Each technique has its own benefit. The first technique, path-compression, is used to reduce an average depth of trie. Then, the structure sharing, the second technique, can reduce frequently used node structures. For the last technique, node compression is applied to compress node size up to 128 times. With the combination of these three techniques, the result shows that SS-trie also beats the size reduction. Therefore, SS-trie is better than both binary and n-ary trie. Moreover, the Structure sharing technique can be use widely in most trie structure.

6 Conclusion

SS trie structure can reduce more space usage when comparing with naïve trie structure and preserving the quality of searching speed. This scheme presented is more practical than traditional trie structure for a huge key set. However, the shared part of this technique described here are one-depth tree structure. It is possible to share higher-depth structures. In addition, SS-trie requires the development of searching, updating and deleting time of trie structure. All of these remain in our future work.

Acknowledgements

This work has been supported by National Electronics and Computer Technology Center (NECTEC), National Science and Technology Development Agency (NSTDA) under the project number NT-B-06-4F-13-311. Thank you to Mr. Arunthep Saengsomboon and Ms. Duangrumol Phatharakittikul for revising our paper.

References

Aho, A. V., Sethi, R., Ullman, J. D. 1985. Compilers: Principles, Techniques, and Tools. *Addison-Wesley.*

Andersson, A, Nilsson, S. July 1993. Improved Behaviour of Tries by Adaptive Branching. *IPL,* 46(6):295-300.

Aoe, J, Morimoto, K, Shishibori, M, and Ki-Hong P. June 1996. A Trie Compaction Algorithm for a Large Set of Keys. *Transactions on Knowledge and Data Engineering* IEEE,8(3).

Aoe, J. September 1989. An Efficient Digital Search Algorithm by Using a Double-Array Structure. *IEEE Transactions on Software Engineering,* 15(9):1066-1077.

Bentley, J., and Sedgewick, R. 1977. Fast Algorithms for Sorting and Searcing Strings. *In the Eight Annual ACM-SIAM symposium on Discrete Algotihms, SIAM Press.*

Clampett, II. March 1964. A. Randomized binary searching with tree structures, Commutation of the ACM, 7(3):163-165.

Comer, D, and Sethi, R. July 1977. The Complexity of Trie Index Construction. *Journal of the Association for Computing Machinery,* 24(3):438-440.

Comer, D. September 1981. Analysis of a Heuristic for Full Trie Minimization. *Transactions on Database Systems* ACM, 6(3):513-537.

De La Briandais, R. 1959. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference,* 15, IRE, New York. Spartan Books, New York, 295-298.

De Maine, P. A. D, and Rotwitt, T. Jr. November 1971. Storage optimization of tree structure files representing descriptor sets. *Proceeding of SIGFIDET Workshop on Data Description, Access and Control* ACM:207-217.

Fredkin, E. September 1960. Trie memory. *Communication* ACM, 3(9).

Harman, D., Fox, E., Baeza-Yates, R. and Lee, W. 1992. Inverted Files. In *Information Retrieval: Data Structures & Algorithms,* Eds. Frakes W.B. and Baeza-Yates R. Prentice Hall, 28-43.

Knuth, D.E. 1973. The Art of Computer Programming (vol. 3). Addison-Wesley, Reading, Mass.

Lucchesi, C.L, and Knowaltowski, T. 1993. Applications of Finite Automata Representing Large Vocabularies. Software Practices and Experiences, 23(1):15-30.

Morrison, D. October 1968. PATRICIA- Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514-534.

Nilsson, S, Karlsson, G. June 1999. IP-Address Lookup Using LC-Tries. *Journal on Selected Areas in Communications* IEEE, 17(6):1083-1092.

Ramesh, R, Babu, A.J.G. and Kincaid, J. P. March 1989. Variable-Depth Trie Index Optimization: Theory and Experimental Results. *Transactions on Database Systems* ACM, 14(1):41-74.

Zandieh, M. December 1999. Evaluation of an LC-Trie Algorithm for IP-Address Lookups. Master thesis, Uppsala University.