공학석사학위논문

Space-efficient implementation of a compressed trie

공간효율적인 트라이 구현

2016 년 2 월

서울대학교 대학원

컴퓨터공학부

신 정 수

Space-efficient implementation of a compressed trie

지도교수 Srinivasa Rao Satti

이 논문을 공학석사 학위논문으로 제출함

2015 년 12 월

서울대학교 대학원

컴퓨터공학부

신 정 수

신정수의 석사학위논문을 인준함

2015 년 12 월

위 원 장 : Byung-Ro Moon (인)

부위원장 : Srinivasa Rao Satti (인)

위 원 : Bernhard Egger (인)

Abstract

Space-efficient implementation of a compressed trie

Jeongsoo Shin
Department of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

Trie is the typical data structure for keyword searching algorithms. The algorithms are classified into two ways. One is the array representation, the other one is the succinct representation. In array representation, one can access the child node using the one dimensional array which stores the pointers from the node to its child. This shows fast lookup time, but takes a lot of space. In succinct representation, one separates a tree structure and index array, and represents a tree structure by succinct representation. This shows slower query time than array representation, but takes less space. In this thesis, we first use the degree sequence and its variants to implement the space-efficient trie on the small fixed alphabet. The experimental result shows that for small alphabet ($|\Sigma| \leq 10$), our implementation gives fast query time and less space usage compared to the exist implementations.

Keywords : Trie, Array representation, Succinct representation, Compression algorithm
Student number : 2014-21791

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information retrieval is most important thing as the information is digitalized. A trie [1] is a popular data structure to represent a string set. The term comes from reTRIEval by Edward Fredkin. The trie is used in text indexing [9], spell checker [10], IP address lookup [11], information retrieval systems [13,14,15,16] and so on [17,18,19]. Trie consists of node and edge. Normally the node has a link pointer and the edge is labeled with a singular character or a string. The leaf node indicates the end of the keyword. The child nodes of a node have a same prefix. The **lookup(S)**, search the string S from the trie, begins by checking a root node and follows a character of the searching keyword sequentially. The searching operation is completed when the last character is ended with the leaf node. Figure 1 is an example of an array-based trie for the set $S = \{ by, sad, sells, the \}$. Insertion, deletion and lookup on the trie are fast, but it wastes large space when the array is sparse. The most common way to compress the

trie is to use a list structure. Each node has a label, a pointer to the child node and a pointer to the sibling node. The leaf node has a label with null pointers. Figure 2 shows an example of a list-based trie for the set S. The list-based trie saves the space by using the null pointers, but the **lookup** time is slow if the length of a key is long.

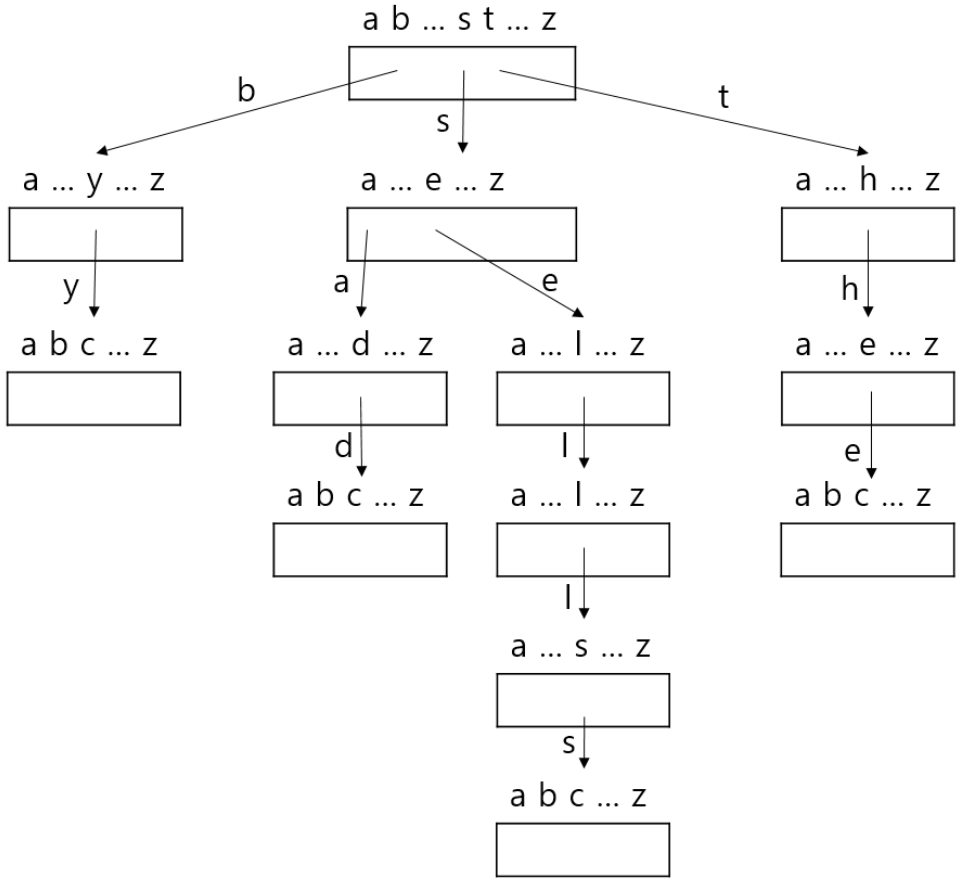


Figure 1 Array-based trie for $by, sad, sells, the$

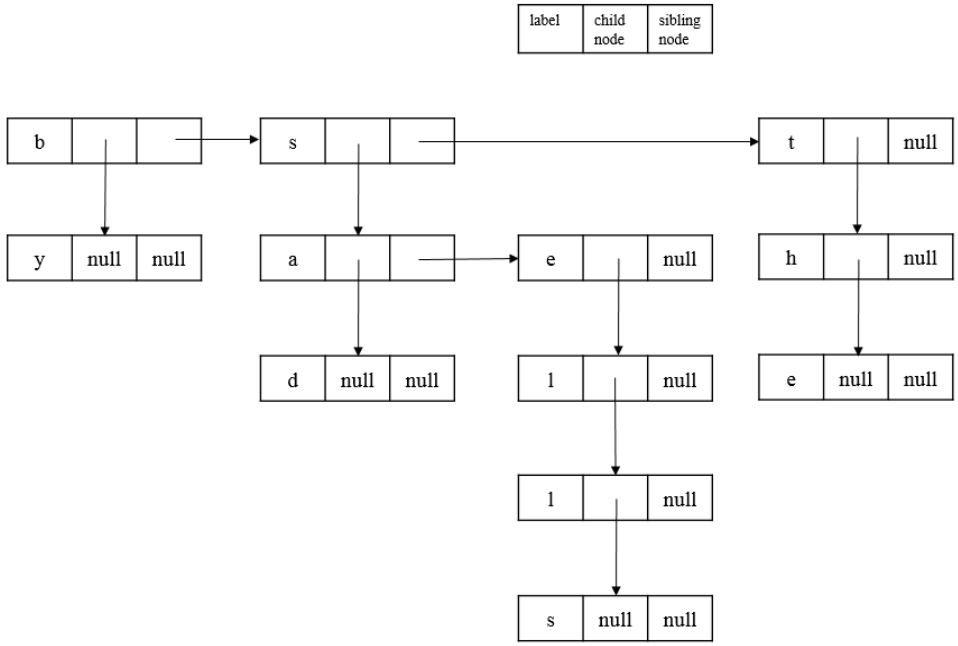


Figure 2 List-based trie for $by, sad, sells, the$

1.1 Previous Work

In the past, a lot of trie representations were introduced. It can be classified into two types. The one is the array representation and the other one is the succinct representation. The array representations such as ODA[2], CDA[3], and DALF[4] use two one-dimensional arrays. The arrays store the index of parent and child node, respectively. The succinct representations such as BP[5], DFUDS[6], and LOUDS[7] have a tree structure and a label. The tree structure is compressed by the various implementations and the label is stored into the normal array. So a practical

implementation is the most important part of the succinct representation. Shunsuke et al. [4] shows that the array representation is faster than the succinct representation because they can access the child node directly. However, the space usage is more than the succinct representation because they should stores the indexes of parent and child nodes. The performance comparison between succinct representations is conducted by Diego et al. [12]. Giuseppe et al [8] presents a new trie using DFUDS structure and path decomposition. It shows a fast query time and space-efficiency.

1.2 Contribution of this thesis

This thesis introduces some practical implementations of the space-efficient trie on the small fixed alphabet using degree sequence and its variants. For example, DNA sequence has $|\Sigma| = 4$ as entire alphabet set $\Sigma = \{ A, C, G, T \}$. In this thesis, we use succinct representation and for representing the tree structure part, we first use a degree sequence [7] and its variants which use small space in some cases. Using the original degree sequence, we can represent the trie with n nodes using $|\Sigma|n + o(n)$ bits and supports **lookup(S)** in $O(|S|)$ time. Our implementations show better performance in query time and less space usage compared to the current implementations of space-efficient tries if $|\Sigma| \leq 10$.

1.3 Organization of this thesis

This thesis is organized as follows. Chapter 2 explains the some array representations and the succinct representations. Chapter 3 introduces a degree sequence and its variants to represent the space-efficient trie on the small fixed alphabet. Chapter 4 shows the evaluation of our implementations. Chapter 5 concludes our paper.

Chapter 2

Preliminaries

In this chapter, we introduce the array representations and the succinct representations of the trie.

2.1 Array representation of trie

The simplest way to represent a trie uses arrays shown in Figure 1. Aoe [2] introduces the double-array representation which reduces the space usage of the traditional method. The original double-array, called ODA still takes large space when a trie is sparse. Compacted Double-Array [3] and Double-Array using Linear Functions [4] are the successor of the original double-array. They reduce the space usage of ODA. These representations are faster than the succinct representations, but the space requirements are higher than those of the succinct representations.

2.1.1 Double-Array

Double array [2] is using two one-dimensional arrays called BASE and CHECK. BASE is the base index of the child node and CHECK is the index of the parent node. The element of BASE and CHECK is a node in the trie. The structure is adopted by the successor such as CDA(Compacted Double-Array) [3] and DALF[4]. It is well known for the fast traversal. The time complexity for the **lookup** operation is $O(1)$ and the space complexity is $2(n+m)\log(n+m)$ bits when n is the number of nodes and m is the number of empty elements. The space usage of ODA is large when the trie is sparse. Figure 3 shows an example of ODA.

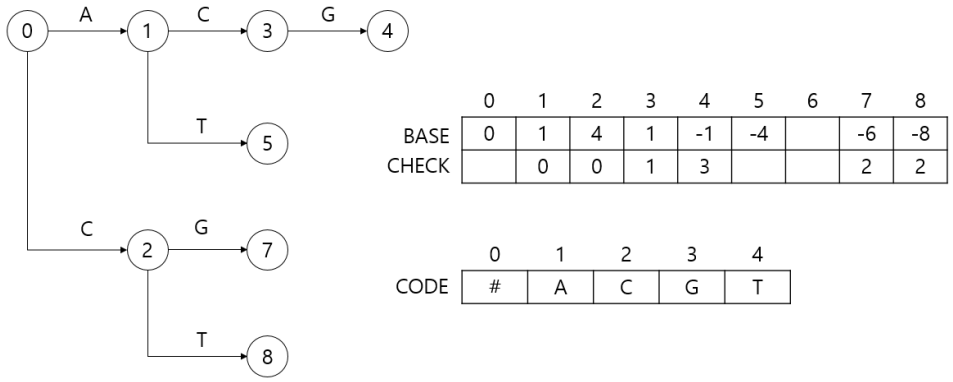


Figure 3 A double-array for ACG, AT, CG, CT

2.1.2 Compacted Double-Array (CDA)

CDA uses two one-dimensional array called BC and TAIL to represent a trie. BC contains the trie nodes and TAIL, the suffix of the string. BC consists of BASE and CHECK. Leaf nodes of BASE are linked to the suffix and other elements of BASE

indicates the offset to the child nodes. The **lookup** time of CDA is same as ODA and the space usage of ODA is reduced up to 40% for English keys.

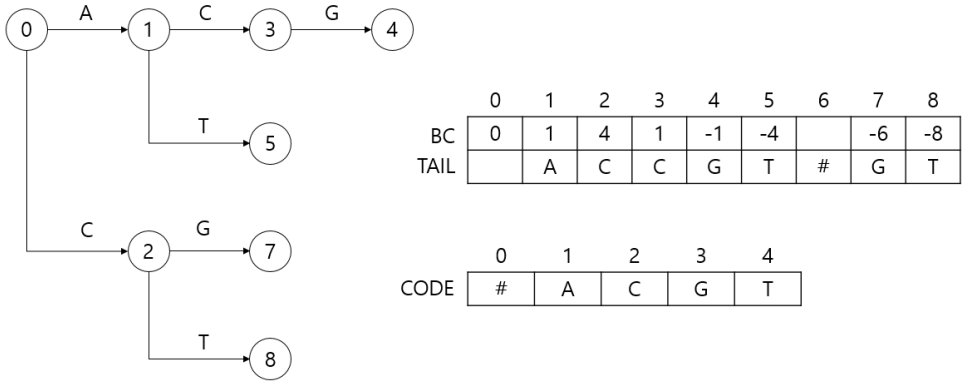


Figure 4 A compacted double-array for ACG, AT, CG, CT

2.1.3 Double-array using linear functions (DALF)

DALF [4] is originated from double-array [2]. DALF improves the space usage of ODA's BASE array by using a linear function. BASE is replaced with DBASE array which is calculated by the function. The space usage is reduced up to 44% supporting the almost same **lookup** performance of the original double-array.

2.2 Succinct representation of trie

The structure of a trie can be represented by the degree of a node. Jacobson introduced the first succinct representations of ordinal trees called LOUDS [7]. The child nodes are ordered in the ordinal tree. Jacobson's succinct representations use only $2n +$

$o(n)$ for the trie structure of n-nodes. Also it supports the primitive operations in $O(1)$. The primitive operations are as follows:

- **parent(x)**: returns the parent of node **x**
- **child(x, i)**: returns the **i**-th child of node **x**.
- **childrank(x)**: returns the rank of node **x** between siblings.
- **depth(x)**: returns the length of the path from the root to node **x**.
- **desc(x)**: returns the number of descendants of node **x**.
- **degree(x)**: returns the number of child nodes of **x**.
- **height(x)**: returns the height of the subtree rooted at node **x**.

The following operations are used for the bit vector.

- **rank(i)**: returns the number of 1's in the first $\boldsymbol{i}(1 \leq \boldsymbol{i} \leq n)$-th positions.
- **select(i)**: returns the position of the $\boldsymbol{i}(1 \leq \boldsymbol{i} \leq m)$-th 1.

One can support the **rank** operation on a bit vector of length n using $n + O\left(\frac{n \log\log n}{\log n}\right) = n + o(n)$ bits with $O(1)$ query time.

BP [5] and DFUDS [6] are the succinct representation using the degree sequence. Table 1 shows the comparison of the primitive operations between various succinct representations.

Operation	LOUDS	BP	DFUDS
parent	✓	✓	✓
child	✓	✓	✓
childrank	✓	✓	✓
depth		✓	✓
desc		✓	✓
degree	✓	✓	✓
height		✓	
rank	✓	✓	✓
select	✓	✓	✓

Table 1 Primitive operations supported $\boldsymbol{O(1)}$ time.

2.2.1 Level-order unary degree sequence (LOUDS)

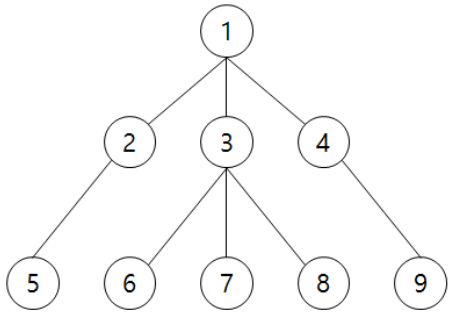
LOUDS [7] is the typical succinct representation of trie. It has a virtual node which is called super node encoded with 10. Each node is traversed by level order and encoded by the binary code. The number of child nodes is represented by the number of 1 and 0 is inserted to distinguish between nodes. A bit array is obtained by visiting each node following the level order. **rank** and **select** operations are introduced to support the **lookup** operation. A new bit vector is required for the operations. It guarantees $O(1)$ time complexity for the **lookup** operation.

2.2.2 Balanced parentheses (BP)

BP [5] is a typical parentheses representation. A pair of matching open and close parentheses called mates are used to represent each node. The preorder traversal sequence is used to encode each node. This representation supports left child, right child, parent, and size operations for binary trees. The length of the parenthesis is $2n$. One is for n open parentheses and the other one is for n closing parentheses. An auxiliary storage uses $o(n)$ bits to support standard operations in constant time. So the space complexity is $2n + o(n)$ bits and the time complexity for the **lookup** operation is $O(n)$.

2.2.3 Depth-first unary degree sequence (DFUDS)

DFUDS [6] is a kind of balanced parentheses representations. The node degrees are encoded in unary codes in depth-first order. For example, the degree d is represented by d (‘s, followed by a) and a dummy (is added at the beginning. As a result a balanced parentheses are obtained. The encoding method for each node is same as LOUDS, but the order is different.



BP : ((()) (() () ()) (()))
LOUDS : 1110 10 1110 10 0 0 0 0 0
DFUDS : 1110 10 0 1110 0 0 0 10 0

Figure 5 An example of BP, LOUDS, and DFUDS sequence of same tree

Chapter 3

Degree sequence and its variants

In this chapter, we introduce the degree sequences and its variants to represent the space-efficient trie on the small fixed alphabet before we describe these representations.

3.1 Degree sequence method

Each node has $|\Sigma|$ bits and each bit of the node stands for a predefined character. For example, DNA sequence has $|\Sigma| = 4$ as entire alphabet set $\Sigma = \{ A, C, G, T \}$. The first bit stands for A and the last bit stands for T. Each node is traversed by level order. If the trie has n nodes, we use $|\Sigma|n$ bits to represent a tree structure of a trie and $o(n)$ bits are required for the auxiliary structure for $O(1)$ time **rank** operation.

For the trie with alphabet of size $|\Sigma|$, we can support **lookup**$(a_1 a_2 \ldots a_k)$ operation uses as follows.

For i = 1 to k

1. If $S_{i-1} + p$-th bit in the bit vector is 0, we terminate where a_i is the p-th alphabet and we define $S_0 = 0$. If we terminate before the k-th loop, the $a_1 a_1 \ldots a_k$ is not in the trie.

2. Go to the position $S_i = |\Sigma| \times rank(S_{i-1} + p) + 1$.

Figure 6 shows an example which represents trie using a degree sequence. $S = \{ AAA, CGA, CGC, CGT, G, TT \}$. The degree sequence of S is 1111 1000 0010 0000 0001 1000 1111 0000 0000 0000 0000 0000 0000.

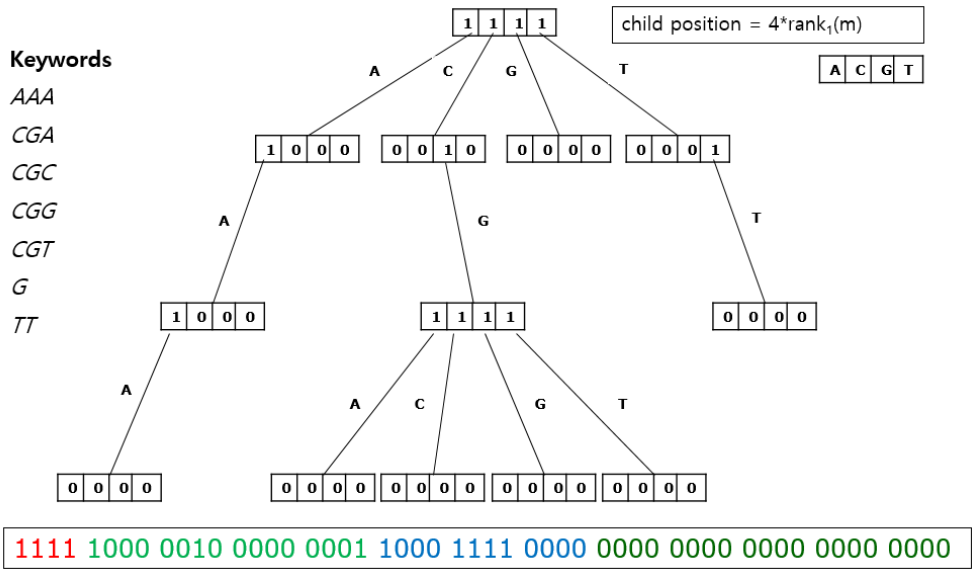


Figure 6 An example of degree sequence for $\Sigma = \{ A, C, G, T \}$

We know that the given character is presented when the node at the result of the formula is the non-leaf node. The node for the last character must be 0000 when the **lookup** is successful. For example, the searching keyword is CGA in the keyword set $S = \{ AAA, CGA, CGC, CGT, G, TT \}$. The bit position of first character C is 2, 3 for G, and 1 for A. By the **lookup** method, we can check the 2nd bit for C, 11th bit for G, and 25th bit for A. We know C, G, and A exist because the value of the bit position is 1.

3.2 Splitting method

The problem of the degree sequence is that the space usage for the leaf node is fat when the portion of the leaf node is overwhelming. So we introduce the splitting method. We remove the leaf node from the degree sequence and add a new bit vector (bit vector of node type) for each node. We can add '1' for the non-leaf node and '0' for the leaf node using level order traverse. Suppose that in the trie with n nodes, n' nodes are non-leaf nodes, then the space requirements is $(|\Sigma| + 1)n'$ bits for the non-leaf node and $n - n'$ bits for the leaf node. Also, we can support **rank** operation for each bit vector using $o(n)$ bits. For the trie with alphabet of size $|\Sigma|$, we can support **lookup**$(a_1 a_1 \dots a_k)$ operation uses as follows. Let $rank_d$ be the **rank** operation on the degree sequence for the non-leaf nodes and $rank_t$ be the **rank** operation on the

bit vector of the node types.

For i = 1 to k

1. If $S_{i-1} + p$-th bit in the bit vector is 0, we terminate where a_i is the p-th alphabet and we define $S_0 = 0$. If we terminate before the k-th loop, the $a_1 a_1 \ldots a_k$ is not in the trie.
2. Go to the position $S_i = |\Sigma| \times rank_d\left(S_{i-1} + p - \left(n - rank_t\left(\frac{S_{i-1}+p}{|\Sigma|}\right)\right)\right) + 1$.

This method is slower than the degree sequence because the number of **rank** operations is two times than the degree sequence. However, the space usage can be less than the degree sequence if the trie has many leaf nodes.

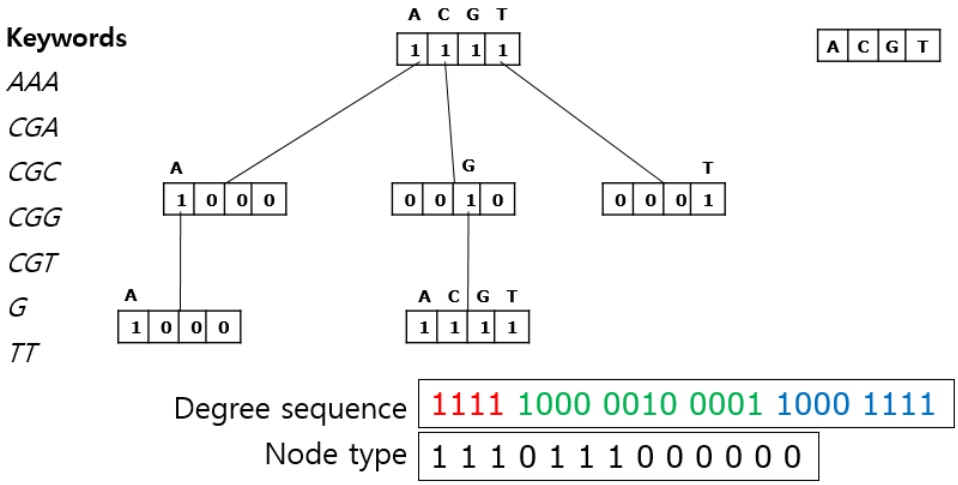


Figure 7 An example of splitting method for $\Sigma = \{A, C, G, T\}$

Figure 7 shows the example of the splitting method. This consists two types of the bit vector. One is for the degree sequence, the other one is for the node type which is '1' for the non-leaf node and '0' for the leaf node. This method uses two **rank** operations for **lookup** operation because the degree sequence for the leaf node is omitted. For example, the searching keyword is CGA in the keyword set $S = \{ AAA, CGA, CGC, CGT, G, TT \}$. The bit position of first character C is 2, 3 for G, and 1 for A on the degree sequence bit vector. By the **lookup** method, we can check the 2nd bit for C, 11th bit for G, and 21st bit for A in the degree sequence bit vector. We know $C,\ G, and\ A$ exist because the value of the bit position is 1 in the degree sequence bit vector.

3.3 Prefix code

This method is introduced by Benoit et al. [6]. Since the trie with $|\Sigma| = \sigma$ has 2^{σ} distinct patterns, we first count the number of occurrences of each patterns in the original degree sequence. After that, we use the prefix code to compress the degree sequence with least expected compressed sequence size. This method assigns small bits for the high frequency bit pattern in the same degree. Table 2 shows the prefix code for $|\Sigma| = 4$.

3.4 Huffman encoding

Huffman code [20] is a well-known optimal prefix code that is used for lossless data compression. Since huffman code guarantees that always give a compression ratio close to its zero-th order entropy, This shows less space than the prefix code introduced in Section 3.3. Table 2 is huffman code for the degree sequence.

Node	Prefix code	Huffman code	# of nodes
0000	00	0101	100000
0100	010	11	840534
0010	011	10	828410
1000	101	00	692126
0001	1001	011	637161
0110	1100	0100101	8586
1100	1101	0100011	6824
0101	10000	0100010	6371
1010	10001	0100001	6306
0011	11100	01001101	5611
1001	11101	01001100	5118
1110	111100	01001001	4037
0111	111101	01001000	3747
1101	111110	01000001	3510
1011	1111110	01000000	2598
1111	1111111	0100111	11133

Average bits for prefix code : 3.2
Average bits for huffman code : 2.37
Average bits for degree sequence : 4
Average bits for splitting method : 4.87

Table 2 Prefix code and huffman code for input set '10 to 60'

Chapter 4

Evaluation of Implementations

We now present results from our implementations. We evaluate the effectiveness of our algorithms, including the space usage and **lookup** time.

4.1 Experimental environment

The code was written in C++. The test machine is Intel(R) Xeon(TM) E7450 6 core CPU @ 2.40GHz, 3 3MB shared L2 cache, and 12MB L3 cache, running Fedora Linux (kernel version: 4.1.6) and g++ 5.1.1 with optimization level 2.

4.2 Data sets

We use data sets with three different alphabet size, 4, 13, and 26. Table 3, Table 4, and Table 5 show the experimental data sets. ‘10 to 20’ and ‘10 to 60’ are from RNA

sequences from NCBI[1] and they are processed to regulate the length of the keyword. 'Set 1' and 'Set 2' are randomly generated with $\Sigma = \{ A, C, G, T \}$. 'Wordnet13' and 'Wordnet26' are from English words of WordNet-2.3.09[2]. 'Wordnet26' is extracted only alphabet from the reference and we remove the duplicated keywords. We convert from 'N-Z' to 'A-M' to regulate the length of keywords for 'Wordnet13' and we remove the duplicated keywords. In 'Wordnet13' and 'Wordnet26', $|\Sigma| = 13$ and 26 respectively. To evaluate the effect of performance on various space efficient tries while the size of the alphabet is increased, we generate the data sets consist of uniformly random-generated keywords with same sizes. For each set of keywords with size k, the set consists of possible permutations of alphabets of size k. In this experiment, we restrict a value of k as 4 to 26. The length for each keyword is 10 and the number of keywords is 100,000 for Figure 13.

	10 to 20	10 to 60	Set 1	Set 2	Wordnet13	Wordnet26
Alphabet size	4	4	4	4	13	26
Average depth	11.85	39.55	15.73	16.29	11.91	11.75
Maximum depth	20	60	42	40	64	64
Number of leaves	100,000	100,000	300,000	300,000	113,559	116,723
Number of nodes	409,632	3,162,072	642,154	648,780	529,498	562,013
File size (MB)	1.22	3.86	4.78	4.94	1.4	1.4

Table 3 Properties of data set

[1] National Center for Biotechnology Information : http://www.ncbi.nlm.nih.gov
[2] WordNet 3.0 : http://wordnetcode.princeton.edu/3.0/WordNet-3.0.tar.gz

Alphabet size	Average depth	Maximum depth	Number of leaves	Number of nodes	File size (MB)
4	4	4	256	341	0
5	5	5	3,125	3,906	0
6	6	6	46,655	55,986	0
7	7	7	300,000	432,406	2.3
8	8	8	300,000	798,883	2.6
9	9	9	300,000	1,185,282	2.9
10	10	10	300,000	1,560,194	3.2
11	11	11	300,000	1,925,920	3.5
12	12	12	300,000	2,281,223	3.8
13	13	13	300,000	2,627,073	4.1
14	14	14	300,000	2,964,868	4.3
15	15	15	300,000	3,297,848	4.6
16	16	16	300,000	3,627,239	4.9
17	17	17	300,000	3,955,031	5.2
18	18	18	300,000	4,281,346	5.5
19	19	19	300,000	4,605,684	5.8
20	20	20	300,000	4,929,305	6.1
21	21	21	300,000	5,251,283	6.3
22	22	22	300,000	5,571,478	6.6
23	23	23	300,000	5,889,674	6.9
24	24	24	300,000	6,205,845	7.2
25	25	25	300,000	6,521,104	7.5
26	26	26	300,000	6,834,386	7.8

Table 4 Properties of data set for $|\boldsymbol{\Sigma}| = \boldsymbol{4}, \ldots, \boldsymbol{26}$

	10 to 20	10 to 60	Set 1	Set 2
Degree 0	100,000	100,000	300,000	300,000
Degree 1	247,660	2,998,231	125,270	129,864
Degree 2	35,411	38,816	153,550	156,216
Degree 3	15,095	13,892	43,553	44,317
Degree 4	11,466	11,133	19,781	18,383

Table 5 Node degree of data set for $|\Sigma| = 4$

4.3 Experimental Result

4.3.1 Space usage

The implementation code for DALF is from the author of DALF, S. Kanda and others except our algorithms are from libtaiju[3]. We use SDSL[4] for **rank** operation.

Figure 8 shows the comparison of the space usage for $|\Sigma| = 4$. Our implementations, (degree sequence, splitting method, huffman encoding), use smaller space than other algorithms. DALF uses the largest space because of the properties of the array-based implementation. Compared to 3 methods we implemented, huffman encoding takes smaller space than others in all cases. In '10 to 20' and '10 to 60', only less than 30% and 5% respectively nodes are the leaf nodes. In this case, splitting method takes more space than the original degree sequence. In other two sets, about half of the nodes are leaf nodes. In this case, splitting method takes less space than the original degree sequence but their difference is not too huge because splitting method requires one extra rank index.

[3] Taiju library : https://code.google.com/p/taiju

[4] Succinct Data Structure Library : https://github.com/simongog/sdsl

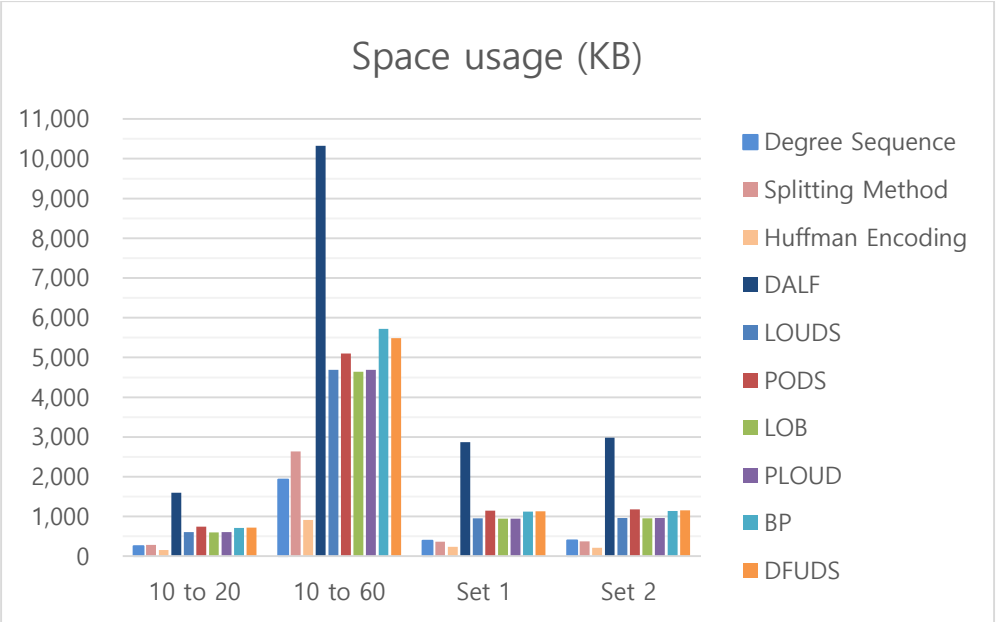


Figure 8 Space usage of the experimental result for $|\Sigma| = 4$

Figure 9 shows the comparison of the space usage for $|\Sigma| = 4, 13, 26$. Huffman encoding shows smaller space than other algorithms. The space usage for degree sequence and splitting method increases proportionally to the alphabet size because the total space usage increases linearly by the alphabet size (other succinct implementations are only affected from the number of nodes).

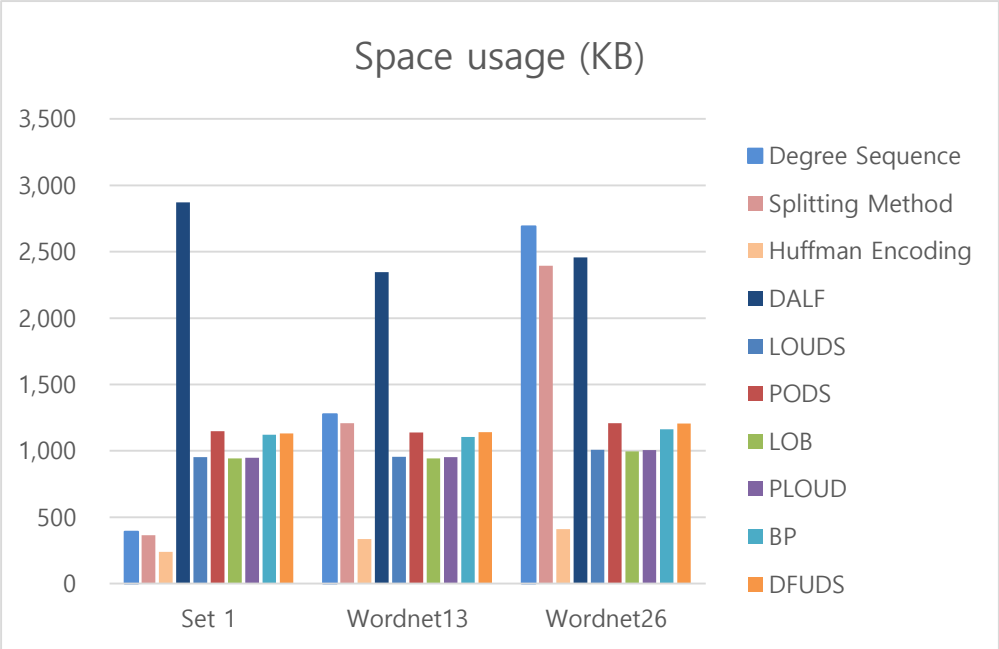


Figure 9 Space usage of the experimental result for $|\Sigma| = 4, 13, 26$

Figure 10 shows the space usage of space-efficient tries (degree sequence, splitting method, huffman encoding, DALF, and LOUDS) for $|\Sigma| = 4, \dots, 26$. Degree sequence with huffman encoding shows smaller space than other algorithms. Original degree sequence and splitting method show smaller space than LOUDS if $|\Sigma| \leq 10$, and even take larger space than DALF if $|\Sigma| \geq 22$. These results are obtained by the same reason as result in Figure 9.

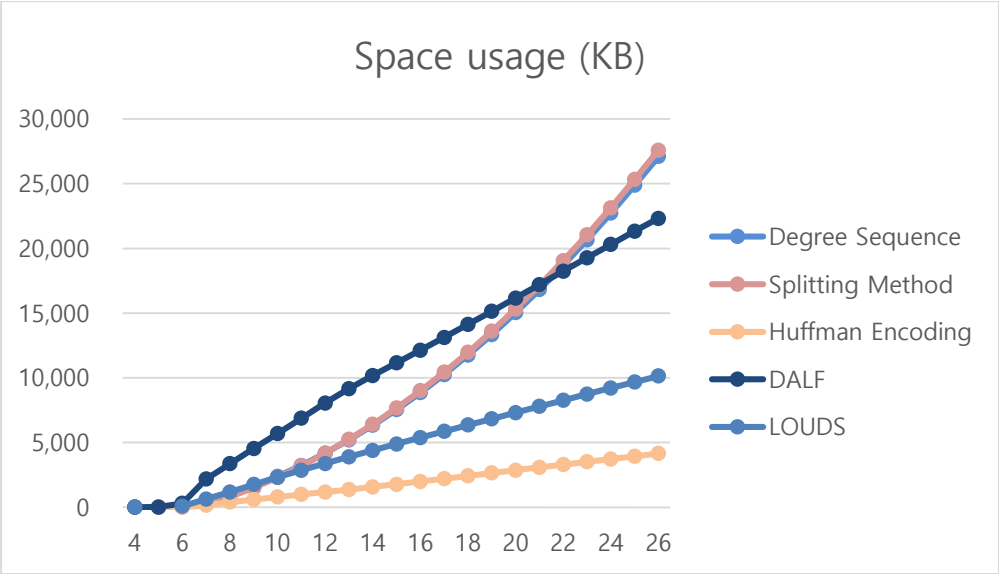


Figure 10 Space usage of the experimental result for $|\Sigma| = 4, \ldots, 26$

4.3.2 Lookup time

Figure 11 shows the comparison of the **lookup** time for $|\Sigma| = 4$. Degree sequence and splitting method are faster than other succinct representations because the other succinct representations use both **rank** and **select** operations for **lookup** operation. Splitting method is slower than degree sequence because two **rank** operations are required to search a single character. DALF is the fastest algorithm in this result because it can access the child index directly.

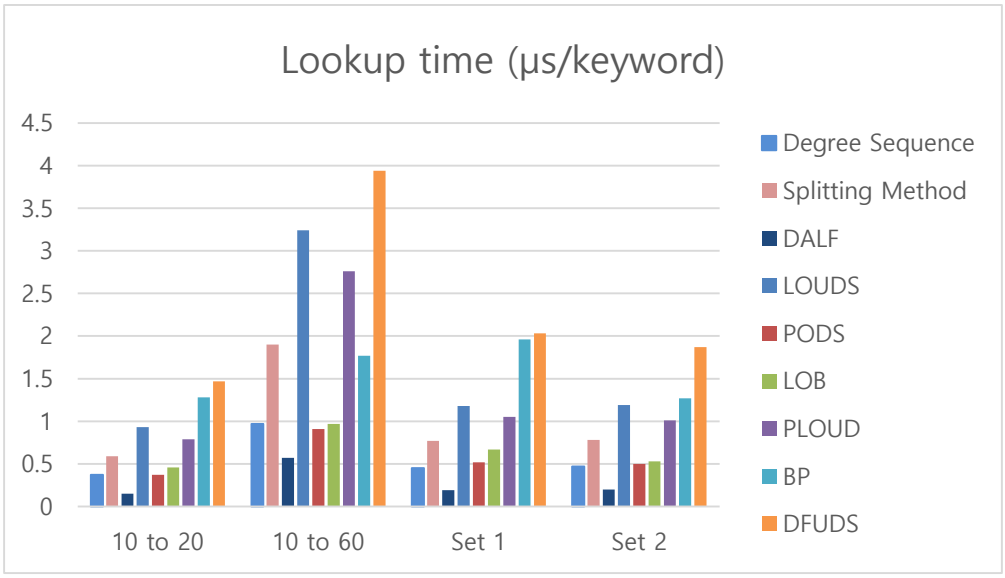


Figure 11 **Lookup** time of the experimental result for $|\Sigma| = 4$

Figure 12 shows the comparison of the **lookup** time for $|\Sigma| = 4, 13, 26$. The lookup performance of our implementations is same regardless of the alphabet size and we use same number of **rank** operations if the length of the keywords are same. BP is only affected from the alphabet size because BP can't find a keyword by the tree structure and it should refer the label. Degree sequence supports faster **lookup** operation than other succinct representations by the same reason as above.

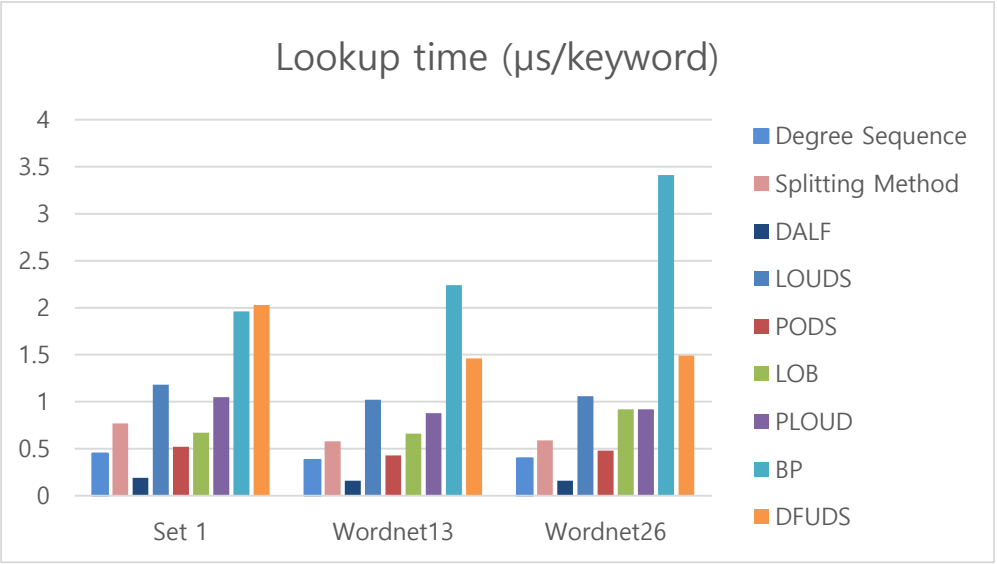


Figure 12 **Lookup** time of the experimental result for $|\Sigma| = 4, 13, 26$

To evaluate the effect of the alphabet size, we also performed another experiment. Figure 13 shows the lookup time of space-efficient tries (degree sequence, splitting method, DALF, and BP) for $|\Sigma| = 4, \ldots, 26$. The lookup performance of BP increases linearly by the alphabet size. Original degree sequence shows the best performance. Splitting method shows better performance than BP.

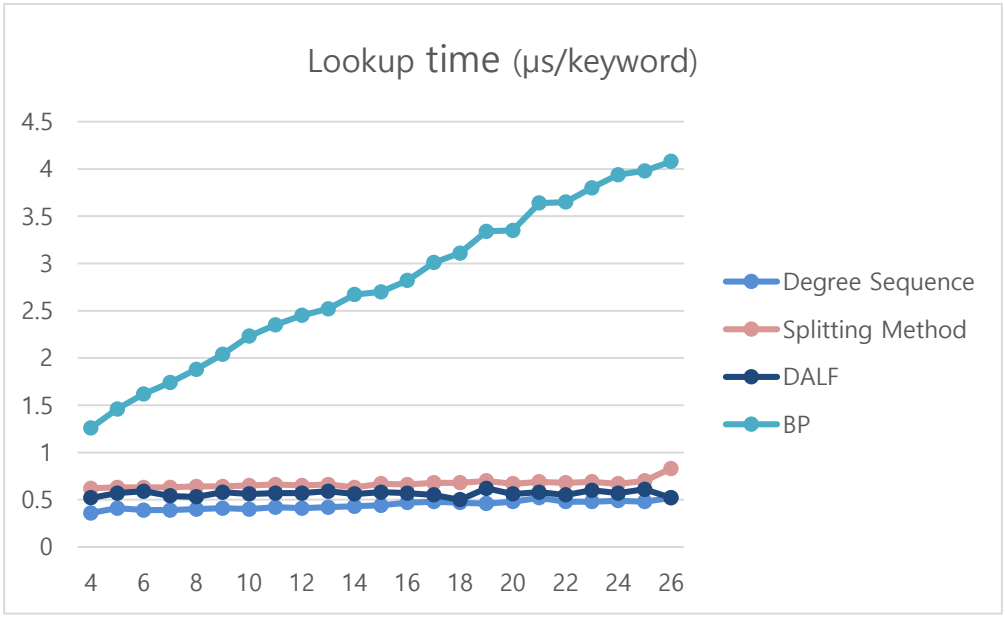


Figure 13 Lookup time of the experimental result for $|\Sigma| = 4, \ldots, 26$

Chapter 5

Conclusion

In this thesis, we implemented the space-efficient trie on the small fixed alphabet. We use succinct representation and for representing the tree structure part, we first use a degree sequence [7] and its variants which use small space in some cases. The space usage for degree sequence and splitting method increases proportionally to the alphabet size. Our implementations show better performance in space compared to the current implementations of space-efficient tries if $|\Sigma| \leq 10$. Degree sequence with huffman encoding shows smaller space than other algorithms. The lookup performance of our implementations is same regardless of the alphabet size. Degree sequence and splitting method are faster than other succinct representations. We end with giving some future works.

- Improving the performance for the long alphabet ($|\Sigma| > 10$).

- Reducing the space for the sparse data.

- Supporting the efficient **lookup** operation for huffman encoding.

Bibliography

[1] Fredkin E.: Trie memory. Communications of the ACM, 3:490–499, 1960.

[2] Jun-Ichi Aoe: An Efficient Digital Search Algorithm by Using a Double-Array Structure. IEEE Trans. Software Eng. 15(9): 1066-1077 (1989)

[3] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, Jun-Ichi Aoe: A compact static double-array keeping character codes. Inf. Process. Manage. 43(1): 237-247 (2007)

[4] Shunsuke Kanda, Kazuhiro Morita, Masao Fuketa, Jun-Ichi Aoe: Experimental Observations of Construction Methods for Double Array Structures Using Linear Functions. JSW 10(6): 739-747 (2015)

[5] Richard F. Geary, Naila Rahman, Rajeev Raman, Venkatesh Raman: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368(3): 231-246 (2006)

[6] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, S. Srinivasa Rao: Representing Trees of Higher Degree. Algorithmica 43(4): 275-292 (2005)

[7] Guy Jacobson: Space-efficient Static Trees and Graphs. FOCS 1989: 549-554

[8] Giuseppe Ottaviano, Roberto Grossi: Fast Compressed Tries through Path Decompositions. ALENEX 2012: 65-74

[9] Gonzalo Navarro: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms 2(1): 87-114 (2004)

[10] James L. Peterson: Computer Programs for Spelling Correction: An Experiment in Program Design. Lecture Notes in Computer Science 96, Springer 1980, ISBN 3-540-10259-0

[11] Jing Fu, Olof Hagsand, Gunnar Karlsson: Improving and Analyzing LC-Trie Performance for IP-Address Lookup. JNW 2(3): 18-27 (2007)

[12] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, Kunihiko Sadakane: Succinct Trees in Practice. ALENEX 2010: 84-97

[13] Marshall D. Brain, Alan L. Tharp: Using Tries to Eliminate Pattern Collisions in Perfect Hashing. IEEE Trans. Knowl. Data Eng. 6(2): 239-247 (1994)

[14] M. J. Nelson: A prefix trie index for inverted files. Inf. Process. Manage. 33(6): 739-744 (1997)

[15] Makoto Okada, Kazuaki Ando, Samuel Sangkon Lee, Yoshitaka Hayashi, Jun-Ichi Aoe: An efficient substring search method by using delayed keyword extraction. Inf. Process. Manage. 37(5): 741-761 (2001)

[16] Makoto Okada, Kazuaki Ando, Samuel Sangkon Lee, Yoshitaka Hayashi, Jun-Ichi Aoe: An efficient substring search method by using delayed keyword extraction. Inf. Process. Manage. 37(5): 741-761 (2001)

[17] Alfred V. Aho, Margaret J. Corasick: Efficient String Matching: An Aid to Bibliographic Search. Commun. ACM 18(6): 333-340 (1975)

[18] Jun-Ichi Aoe, Katsushi Morimoto, Masami Shishibori, Ki-Hong Park: A Trie Compaction Algorithm for a Large Set of Keys. IEEE Trans. Knowl. Data Eng. 8(3): 476-491 (1996)

[19] Venkatachary Srinivasan, George Varghese, Subhash Suri, Marcel Waldvogel: Fast and Scalable Layer Four Switching. SIGCOMM 1998: 191-202

[20] D.A. Huffman: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the I.R.E. 1952, 1098–1102

요약

Trie는 키워드 검색에 있어 가장 잘 알려진 자료구조이다. Trie는 크게 두 가지 범주로 나눌 수 있는데 그 중 하나는 배열을 이용한 방식이고 다른 하나는 간결한 표현 기법을 사용한 방식이다. 배열을 이용한 표현 기법은 자식 노드를 가리키는 포인터를 저장하는 일차원 배열을 이용하여 자식 노드에 접근하게 된다. 이러한 방법은 빠른 검색 성능을 보이지만 공간을 많이 차지한다. 간결한 표현기법은 트리 구조와 인덱스 배열을 나눠서 저장한 후 트리 구조를 간결한 구조로 표현하는 방법을 뜻한다. 이러한 방법은 배열을 이용한 표현기법보다 느리지만 공간을 적게 차지한다. 본 논문에서는 처음으로 작은 고정된 크기의 알파벳 상에서 정의된 trie를 구현하기 위해 degree sequence와 그 변형들을 사용하였다. 본 논문에서 제안한 구현은 작은 알파벳 사이즈에 대해 ($|\Sigma| \leq 10$) 다른 구현 방법들에 비해 적은 공간과 빠른 검색시간을 지원한다.

주요어 : Trie, 배열 기반 자료 구조, 간결한 자료 구조, 압축 알고리즘
학 번 : 2014-21791