Fast Full Text Search Using Tree Structured [TS] File

Takashi SATO

Dept. Arts and Sciences (Computer Science)
Osaka Kyoiku University
4-698-1 Asahigaoka, Kashiwara, 582 Japan

e-mail: `sato@cs.osaka-kyoiku.ac.jp`

Abstract. The author proposes a new data structure (TS–file) in order to make a fast search for an arbitrary string in a large full text stored in secondary storage. The TS–file stores the location of every string of length L (the level) in the text. Using this, we can efficiently search for, not only strings of length L but also those shorter than or longer than L. From an analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length l_k is shorter than or equal to L, and $2(L - l_k + 1)$ otherwise. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS–file, a compressed TS–file is introduced in order to lower storage costs for practical use without losing search speed. The experimental results on compression using UNIX online manuals and network news show that the space overhead of the TS–file against the text searched is from 17% (when $L = 3$) to 212% (when $L = 12$) which is small enough for practical use.

Key words: data storage and indexing, gram based index, full text search, no false drop, TS–file

1 Introduction

The capability to search for strings which are not specified in advance is required more and more recently in the various ways of processing online data such as documents, articles, books, manuals, news, dictionaries and so on. When a text becomes huge, methods which search the full text directly[1]–[4] are not practical. So auxiliary data structures are used in order to speed up the search[5]–[8]. A signature file[9],[10] is a typical data structure for such purposes and it is widely used in practical applications[11]–[13]. However, when we consider the recent status of secondary storage which is rapidly increasing in space per drive and decreasing in cost per bit, faster and more flexible string searches are needed more than those which require less space.

In this paper, we propose a new data structure called a *TS–file* (Tree Structured file) and a set of algorithms using this in order to make arbitrary string searches especially fast. In a previous paper, using a compressed data file we proposed an algorithm which is efficient when the length of the search string is rather long[14].

The method in this paper is most suitable when the length of the search string is rather short. The basic ideas of the TS–file is to store the location of every string of length L in the text. Using a TS–file, not only strings of length L but also those shorter than or longer than L can be searched efficiently.

A retrieval system using transposed files based on single characters, pairs of adjacent characters and longer strings of adjacent characters has been reported for searches of Japanese text[15]. Since the size of the Japanese character set is large, multiple data structures are provided for these combinations of character classes. This system is analogous to our method for each $L = 1, 2, \cdots$, however, our method prepares only one data structure and it has a unique L value.

Since one can find arbitrary strings using the TS–file alone, the proposed method is more accurate than the one using signature files or PAT tree[7] by which one can only know the possibility of existence. The proposed data structure is not made from a word by word index stored as an inverted index[16]. In other words, it does not depend on any specific language styles, for example, in which words are separated by blanks and so on. So, the proposed method is applicable to a wide variety of pattern matches which includes bit strings and genetic information.

From the analysis of search cost, the number of accesses to secondary storage in order to find the first match to a key is two when the key length l_k is shorter than or equal to L, and $2(L - l_k + 1)$ otherwise. This is far less than in the case of signature file search. The proposed algorithm is one of the fastest for arbitrary string searches. And the time required to find all matching patterns is proportional to the number of matches, which is the lowest rate of increase for these kind of searches. Because of the high storage cost of the basic TS–file, we introduce a compressed TS–file by (1)making the data structure a tree to remove unused slots (*null* pointers), and (2)storing differences between adjacent elements if possible in order to lower storage costs for practical use without losing search speed.

Experiments using UNIX online manuals (up to 6.1Mbyte) and network news (up to 500Mbyte) as source text show that the searches are very fast and their time is less than 200msec in most cases and several hundred milli seconds even when keys have many matches or their length is much longer than the level. The overhead of storing the TS–file compared to text size is 30% when $L = 4$ and 66% when $L = 6$ for UNIX online manuals and 47% when $L = 4$ and 212% when $L = 12$ for network news. These overheads are small enough for practical use.

2 Definitions

The alphabet is denoted by Σ. σ $(=|\Sigma|)$ denotes the size of the alphabet. The TS–file stores the location of every string of a given length (called a *gram*) in the text. This length is called the *level*, L. A string sought is a *key*, k, whose length is l_k. A key is constituted of characters c_i. So a key is denoted by $k = c_1 c_2 \cdots c_{l_k}$ $(c_i \in \Sigma)$. The length of the *text* searched is n characters. The text is assumed to be large compared with the size of main memory and is stored on secondary storage. Data are transferred to and from the main memory in blocks of size B words. The units of memory are *word*, *half word* and *character*. The number of characters per word is w. In typical cases, 1 word = 4 byte and 1 character = 1 byte so that $w = 4$.

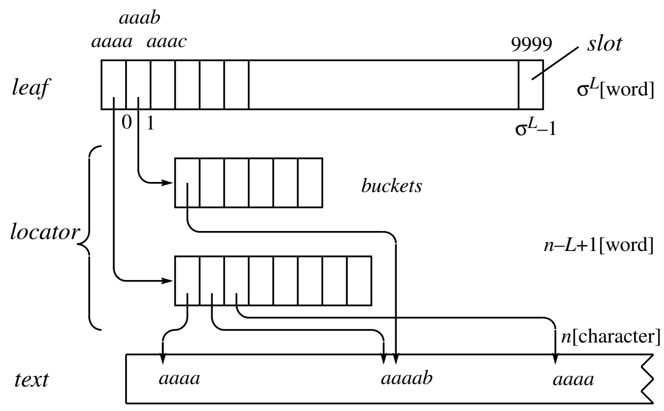


Figure 1: TS-file: Basic Structure ($L = 4$).

3 Basic TS–file and Its Storage Cost

3.1 Basic Data Structures

The basic TS–file consists of a *leaf* and a *locator*. Because the number of combinations of strings with length L is σ^L, the leaf has addresses of L digits in the σ-ary system $(0, 1, \cdots, \sigma^L - 1)$(see Fig. 1). We call each leaf address a *slot*. Each slot has a pointer which points to a bucket in the locator or *null*. Each bucket has pointers which point to locations in the text. The bucket which is pointed to by a slot stores the locations where the strings corresponding to the slot are found in the text. If there is no corresponding string in the text, the pointer in the slot is *null*.

3.2 Storage Cost

The size of the leaf is σ^L words assuming 1 word/slot. The size of the locator is $n - L + 1 \simeq n$ words assuming 1 word/pointer because there are $n - L + 1$ strings of length L in the text. Summing these up, the storage cost becomes

$$\sigma^L + n \quad \text{[word]}. \tag{1}$$

When slots used (i.e. not *null*) are sparse, we collect only used slots. Then slots become two words each because each slot should contain a slot value also. In this case a leaf is at most $2n$ words because the number of slots does not exceed n. Taking account of this collection of used slots, the storage cost becomes less than

$$min\{2n, \sigma^L\} + n \quad \text{[word]}. \tag{2}$$

If we can store information without having to align word boundaries, we can store data in every bit. The slot value is expressed in $L\lceil\log_2 \sigma\rceil$ bits and the pointer is expressed in $\lceil\log_2 n\rceil$ bits, so the above cost becomes as follows.

$$min\{n(L\lceil\log_2 \sigma\rceil + \lceil\log_2 n\rceil), \sigma^L\lceil\log_2 n\rceil\} + n\lceil\log_2 n\rceil \quad \text{[bit]}. \tag{3}$$

4 Search Algorithm and Its Cost

This section shows concrete algorithms based on the data structure introduced in **3**. We obtain a search cost by estimating the number of block transfers between main memory and secondary storage.

4.1 Search Algorithm

The algorithm is explained in terms of three cases according to the relation between l_k and L.
(A) $l_k = L$

(1) Obtain a slot address for k by

$$s = \sum_{i=1}^{l_k} \mathrm{ord}(c_i)\sigma^{L-i}, \tag{4}$$

where 'ord' represents an arbitrary function which maps each character uniquely onto $0, 1, \cdots, \sigma - 1$.

(2) Find a bucket of the locator which stores pointers to the same strings as k by following the pointer in the slot obtained in (1).

(3) List the locations where k appears in the text by following the contents in the bucket found in (2).

(B) $l_k < L$

(1) Obtain lower and upper limit of slots (s_1 and s_2 respectively) since the slots to be searched are consecutive.

$$\begin{aligned} s_1 &= \sum_{i=1}^{l_k} \mathrm{ord}(c_i)\sigma^{L-i}, & (5)\\ s_2 &= \sum_{i=1}^{l_k} (\mathrm{ord}(c_i) + \delta_{i,l_k})\sigma^{L-i} - 1, & (6)\end{aligned}$$

where $\delta_{i,i} = 1, \delta_{i,j} = 0 (i \neq j)$.

(2) Obtain buckets of the locator which are pointed to by the pointers in the slots of between s_1 and s_2.

(3) List the locations where k appears in the text by following contents in the buckets found in (2).

(C) $l_k > L$

(1) Obtain $l_k - L + 1$ slots from the following equation.

$$s_j = \sum_{i=1}^{L} \mathrm{ord}(c_{i+j})\sigma^{L-i} \quad (j = 0, \cdots, l_k - L). \tag{7}$$

(2) Obtain buckets of the locator which are pointed to by the pointers in the slots $s_j (j = 0, \cdots, l_k - L)$

(3) Find candidate locations where k may appear by following contents in the buckets found in (2). Candidate locations for the appearance of the key k in the text are the offsets of the obtained buckets contents minus j.

(4) List the locations where k truly appears by intersecting the sets of candidates for each j.

When $l_k < L$, we compute the set sum of locations in the buckets pointed to by the slots corresponding to strings containing the key. When $l_k > L$, we have to compute the set product of locations in the buckets pointed to by the slots contained in the key.
[Example1] Fig. 2 (a), (b) and (c) show how to follow the pointers in leaves and locators of a $L = 4$ TS–file when the key is 'text', 'ftr' and 'search' respectively. Since the buckets of the locator are stored sequentially, they are drawn in one box and separated by double lines. □

4.2 Search Cost

We estimate the cost to execute the algorithms in **4.1**. Because the TS–file is on the secondary storage, the search cost becomes the number of block transfers (fetches) from secondary storage to main memory.
(A) $l_k = L$

One fetch is required to read a slot computed from equation (4). When the number of matches for the key is M, $\lceil M/B \rceil$ fetches are required in order to read all matches in the locator. Summing these up, we obtain the cost f^a_{eq}.

$$f^a_{eq} = 1 + \lceil M/B \rceil \tag{8}$$

The fetches required to find a first match is $f^1_{eq} = 2$ because only the first block of the locator has to be read.
(B) $l_k < L$

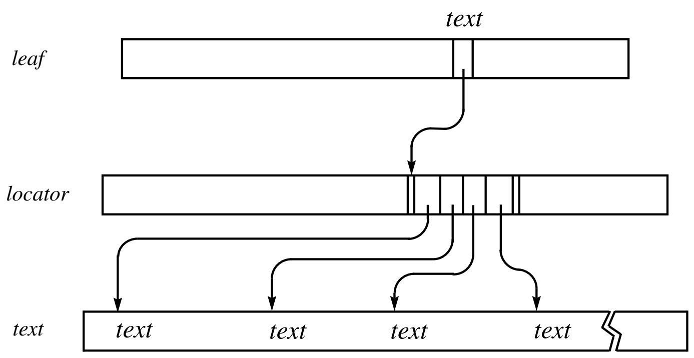
The algorithm fetches consecutive slots from s_1 to s_2 and buckets of the locator pointed to by these slots. Not only slots but also the buckets pointed to by the consecutive slots are expected to be stored in adjacent regions of secondary storage. Then the number of fetches to find all matches becomes

$$\begin{aligned} f^a_{lt} &= \lceil (s_2 - s_1 + 1)/B \rceil + \lceil \sum_{i=s_1}^{s_2} M_i/B \rceil, \\ &= \lceil \sigma^{L-l_k}/B \rceil + \lceil \sum_{i=s_1}^{s_2} M_i/B \rceil, \end{aligned} \tag{9}$$

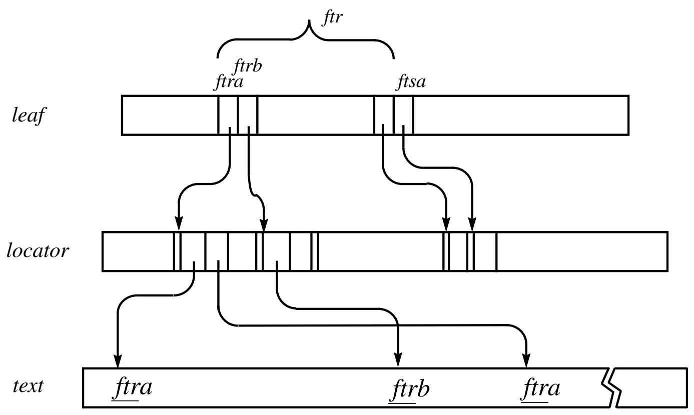
where M_i is the number of matches for slot i. The first term of this equation becomes quite small under the compression proposed in **6.2** because it comes from a sequential scan of the leaf. The fetches required to find a first match is $f^1_{lt} = 2$ because only the first block of the locator for the slot s_1 has to be read.
(C) $l_k > L$

Although the algorithm fetches $l_k - L + 1$ slots and the related parts of the locator, they are not necessarily adjacent on the secondary storage. So the number of fetches to find all matches becomes

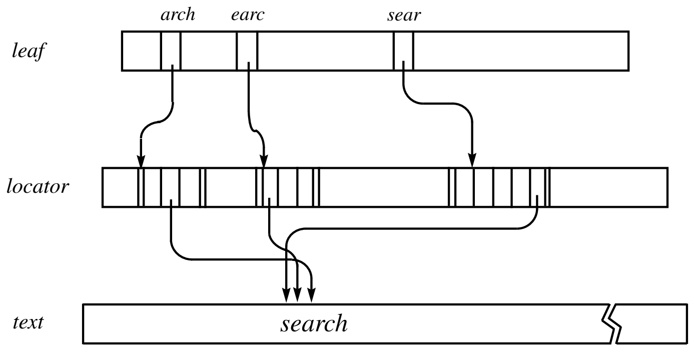
$$f^a_{gt} = l_k - L + 1 + \sum_{j=1}^{l_k - L + 1} \lceil M_j/B \rceil. \tag{10}$$



(a) $l_k = L, key =$ *"text"* $(l_k = L = 4)$.



(b) $l_k < L, key =$ *"ftr"* $(l_k = 3, L = 4)$.



(c) $l_k > L, key =$ *"search"* $(l_k = 6, L = 4)$.

Figure 2: Example of TS–file search

The fetches required to find a first match is $f^1_{gt} = 2(l_k - L + 1)$.

We have analyzed the cost in three cases from the relations between l_k and L; the search cost is independent of n as long as the number of matches (M, M_i, M_j) are constant. And we have to pay attention to the fact that slots are accessed sequentially when $l_k < L$, but randomly when $l_k > L$.

5 Comparison with Other Methods

In this section, we review signature files, which are widely used for full text searches, and PAT trees which provide fast retrieval times. Both are not based on word indices and can be retrieved by arbitrary strings. We then compare these with the TS–file.

5.1 Signature Files

In section **6**, we present a way of compressing TS–files without losing the features of a fast full text search. Since we have to handle many parameters there, we will compare our method with search using signature files from the viewpoint of search speed before section **6**. The signature file is a typical example of an auxiliary data structure which is used in order to make a full text search fast. In this section, we compare the method proposed in the previous sections with the method of signature files, and we show the former is much faster than the latter. The signature file is known as an effective method for fast search and has been studied extensively. Because there are many forms of signature files, we first outline the signature file which is the object of comparison with the method proposed. Next we estimate the search cost when the signature file is used and compare this with the results discussed in section **4**.

Since the TS–file can search for a string in a text which is not necessarily composed of words, we assume that the signature files with which we are comparing are also made from grams (i.e. strings of characters) and not words. We divide the text into logical blocks of D characters. For each logical block, we make a bit vector of length b bits. We make each bit vector as follows. The bit vector is bit string of all zeros initially. For each pair of adjacent characters from the beginning of the logical block, one obtains a number between $0, 1, \cdots, b-1$ by applying an appropriate hash function. The i-th position of the bit vector is set to '1' when the number is i. Because the number of such pairs is $(D-1)$, the hash function sets bits to '1' in this bit vector $(D-1)$-times. Although the hash function should be selected carefully so as to distribute '1's randomly and uniformly, we cannot avoid collisions. Combining the bit vectors of all blocks, we get a signature file. As we assume n is the length of the text, the number of logical blocks is $\lceil n/D \rceil$. We use α for the ratio of b to D $(\alpha = b/D)$. Then the size S in bytes of the signature file becomes

$$S = b/8 \times \lceil n/D \rceil \simeq \alpha n/8 \ \ \text{[byte]}. \tag{11}$$

In order to make the search fast, this signature file is sliced column wise when the bit vectors to be looked at are stored row wise. S is divided into b sub-files whose size is $S_b = S/b \simeq \alpha n/8b = n/8D$.

Next we consider a key search using the above sliced signature file. Since the key length is l_k, we compute the locations where '1' is set $(l_k - 1)$ times from the pairs of adjacent characters in the key. As we assume $l_k \ll D$, hash collisions are negligible, so the number of these locations is approximately $(l_k - 1)$. Searching the $(l_k - 1)$ sub-files which store the location in the bit vectors corresponding to the locations of 1's computed from the key, we return as candidates the rows which have 1's in all these sub-files. In this case the number of fetches f^a_{sg} of the signature sub-files becomes

$$\begin{aligned} f^a_{sg} &= S_b(l_k - 1)/B \\ &\simeq n(l_k - 1)/(8DB) \end{aligned} \tag{12}$$

We know that this search returns the numbers of logical blocks which may contain the key. So we have to access the text blocks directly and examine them in order to know whether the key truly exists or not and what are the offsets in these blocks if it exists.
[Example2] When $n = 10^8, B = 1024, D = b = 256, L = 6, l_k = 7, M_1 = M_2 = 100$, the numbers of fetches of the two methods are

$$\begin{aligned} f^a_{sg} &= 286 \\ f^a_{gt} &= 4. \end{aligned}$$

In this case, we see that the method proposed in this paper is about seventy times faster than the method using signature files. □
f^a_{gt} does not change with n, however, f^a_{sg} grows in proportion to n (i.e. $O(n)$).

5.2 PAT Trees

A PAT tree[7] is a Patricia tree constructed over all the possible strings (called sistring) formed by starting at a given position and continuing to the end of a text. A Patricia tree[17, 18] is a digital tree where the individual bits of the keys are used to decide on the branching. A zero bit will cause a branch to the left subtree, a one bit will cause a branch to the right subtree. Hence Patricia trees are binary digital trees. In addition, Patricia trees have in each internal node an indication of which bit of the query is to be used for branching. This may be given as a count of the number of bits to skip. This allows internal nodes with single descendants to be eliminated, and thus all internal nodes of the tree produce a useful branching, that is, both subtrees are non-null.

Patricia trees store key values at external nodes; the internal nodes have no key information, just the skip counter and the pointers to the subtrees. The external nodes in a PAT tree are sistrings, that is, integer displacements. For a text of size n, there are n external nodes in the PAT tree and $n - 1$ internal nodes.

Retrieval using a PAT tree follows edges from a root towards leaves by considering skip counts in nodes. The contents of leaves in a subtree which follow edges where the bit comparisons end are candidate places for the key. Since there may have been bits skipped which should have been compared, we have to access the text and confirm whether we can find the key at that place or not. This method can not avoid false drops.

The depth of a leaf is the number of comparisons required to distinguish its sistring from others. The average depth is $d = \log_2 n$. It is natural to assume that PAT trees are stored in secondary memory because the texts being searched are large and therefore also stored in secondary memory.

We store skip counts of a complete binary tree of e levels [*] and pointers from nodes at the lowest level of the tree into a one block space in memory. If we assume these are represented in a one word space, the size of data in one block is $2^{e+1} - 1$[word]. Then the number of levels which fit into a block is at most $e = \log_2(B+1) - 1$. When we cache one block which stores the root of a PAT tree in main memory, the average number of block accesses to a leaf in order to find a pattern which may match with the key is at least $\lceil d/e \rceil - 1$. Including one access to the text for confirmation, this becomes $\lceil d/e \rceil$.

The whole subtree lying under a point where the bit comparisons end has to be searched in order to find all patterns which match with the key in a text. If we assume that the point is at level t, the average number of nodes in the subtree becomes 2^{d-t}. Dividing by the number of words in a block, the number of block accesses becomes $2^{d-t}/B$ [†]

[Example 3] When $n = 2^{30} (\simeq 10^9)$, $B = 1024$, $d = 30, e = 9$, the number of disk accesses to find a pattern which matches the key is 4, and the number of accesses to find all patterns when $t = 15$ is 32. $\square$

We compare these results with the TS–file when $l_k \leq L$. The number of accesses to find one matching string is two in the case of the TS–file. To find all strings matching a key, we scan a part of the locator of the TS–file. It contains the same data as the leaf of a subtree of the PAT tree which starts at a point at level t but its size is half that of the PAT subtree. Based on this, we estimate that the search speed of the TS–file is twice as fast as that of the PAT tree in this case. On the other hand, when $l_k \gg L$, a PAT tree becomes advantageous in terms of the search speed.

The problem with a PAT tree is its high construction cost. In order to make a PAT tree, all the sistrings in the text have to be sorted. When n sistrings whose average length is $n/2$ are sorted in main memory whose work size is p blocks by multi-way merge sort, the number of disk accesses is $f^s_{PAT} = (n/Bw)^2 \lceil \log_p (n/Bw)^2/2 \rceil$ [‡]. On the other hand in the case of a TS–file it is $f^s_{TS} = 2(nL/Bw)\lceil \log_p (nL/Bw) \rceil$.

[Example 4] When $n = 2^{25}, B = 1024, w = 8, L = 12, p = 2048$,

$$\begin{aligned} f^s_{PAT} &\simeq 50,332,000 \\ f^s_{TS} &\simeq 131,000. \end{aligned} \tag{13}$$

The sort used in making a TS–file is about 400 times faster than that for a PAT tree. $\square$

6 Compression of the TS–file

Although we can search for arbitrary strings fast by the method explained in the earlier sections, the TS–file often becomes too large to store in practice. So we want to compress the data structure without losing the features of fast full text searches. We explain compression methods for the locator and the leaf respectively in the following.

[*] in a PAT tree a level is a set of nodes which are at the same depth from the root.

[†] This is the case when every block is 100% filled up. If we use $\log_e 2 \times 100\%$, an average value reported in the literature [7], the number of accesses becomes $2^{d-t}/B \log_e 2$.

[‡] In general, the number of disk accesses required to sort x blocks of data by p–way merge sort is $2x\lceil \log_p x \rceil$.

6.1 Compression of the Locator

6.1.1 Using Block Numbers for Locations

In this paper, we are not concerned with searches making use of the text structure (for example SMGL or Hyper text) [19],[20]. When we search a large text, however, it is rather rare that it have no structure. Usually texts have logical blocks at least. So, numbering each logical block in sequence, we adopt the method in which the numbers are output as the result of the search instead of the locations (pointers) where the key appears in the text. Because the number of pointers in one logical block is (the number of characters in the block $-L+1$), the number expressed should decrease considerably if we use the logical block numbers instead of pointers as output results. Using block number, a half word may be enough in most cases even if the pointer is one word in these cases. For example, in section **7** we use a UNIX online manual whose size is 6.11 Mbyte. Although we use 4 bytes (1 word) to express pointers, 2 bytes (half word) is enough to express the number of logical blocks (manual pages) which is 2707 in total. So we halve the amount of storage. Moreover, since the same strings often appear in the same logical block, duplicated logical block numbers for the same string should be removed. If we assume c is the average number of duplications, the compression rate α_c by this compression method in storing the locations where the key appears becomes

$$\alpha_c = 1/2c. \tag{14}$$

When the locations are stored by block number, $-j$ in the algorithm of **4.1**(c)(3) should be removed, and the result gives only the possibility of existence. Now we have to access the logical blocks of the text whether the key is truly there or not. However, we have confirmed experimentally that if L is large enough ($L \geq 6$), false drops are very rare in practice.

6.1.2 Run-length Encoding

According to **6.1.1**, each bucket of the locator contains the numbers of the logical blocks which include a given string of length L. If the differences between adjacent numbers are small, we can store them in less storage space by sorting these numbers and storing their differences. This method is called *run-length encoding*[16].

For example, if 7bits is used to express the difference and 1 bit to express whether the next data is differenced or not, 1 byte (=8bits) is enough to store a block number when the difference is less than 128. We can halve the memory required for the locator compared with the case when 2 bytes is used for each logical block number. Assuming that p_d is the probability of being able to store by difference, the compression factor α_d by this method becomes

$$\alpha_d = 1 - p_d/2. \tag{15}$$

When we apply both **6.1.1**,**6.1.2** methods, the total compression rate for the locator becomes $\alpha_c\alpha_d$.

6.2 Compression of the Leaf

6.2.1 Using a Tree Structure

Although in **3.1** we prepared slots for every sub string of length L, the results of the experiments in **7** show that the usage rate of slots is not high. As we stated in **3.2**, we can remove slots that are not used in order to save space. If these slots are simply removed and the leaf is compacted, we can no longer compute, using just the sub string, the slots where the desired pointer to the locator is stored. So another auxiliary data structure should be added in order to access slots quickly. Firstly, in each slot we store not only a pointer to the locator but also a slot value computed from the sub string. Though each slot size increases from 1 word to 2 words, assuming a slot value can be expressed in 1 word, considerable compression is expected as a result because slots which have *null* pointers can be removed. If u_s is the usage rate of the slots, the compression factor becomes

$$\alpha_s = 2u_s. \tag{16}$$

Secondly, we prepare the *root*, a data structure which stores the values of regularly spaced slots (interval=b_f) of the leaf in order to guarantee fast accesses to the required slots of the leaf. In order to make leaf access only one block, b_f is set as

$$b_f = B/2 \tag{17}$$

from the fact that the physical block size is B (transfer unit between main and secondary memory) and each slot is 2 words (one for the slot value and one for a pointer to the leaf). Using this data structure, two fetches are required to access the leaf, as long as accessing the root is one fetch. Since the slots of the leaf are accessed in only one fetch for the basic TS–file, one more fetch is required in this case. But from the fact that physical data accesses are not required when we retrieve keys successively because the root is cached in main memory, the influence on search performance is negligible. If b_c is the number of blocks that can be used as the root cache, the whole root can be put in cache when the usability of slots u_s is

$$u_s \leq b_c b_f B/2\sigma^L = b_c B^2/4\sigma^L. \tag{18}$$

Although in most cases, a two-level structure (root and leaf) is enough, we can make the root into a multi-level structure, i.e. a tree structure, if we can not put the whole root in main memory. Fig. 3 shows this tree structure.

6.2.2 Compression of Slots

When we adopt the structure for the leaf proposed in **6.2.1**, each slot is composed of a slot value and a pointer to a bucket of the locator. Among these, the former is compressed to 1 byte by the run-length encoding as **6.1.2**, when the difference of neighboring slot values is less than 128. Because buckets are stored in the order of corresponding slot value, the pointers which point to buckets can be also compressed by run-length encoding.

Assuming p_s and $p_p (0 \leq p_s, p_p \leq 1)$ are the possibilities of storing differences for slot values and pointers respectively, the compression factor by this method becomes

$$\alpha_p = 1 - (3/8)(p_s + p_p). \tag{19}$$

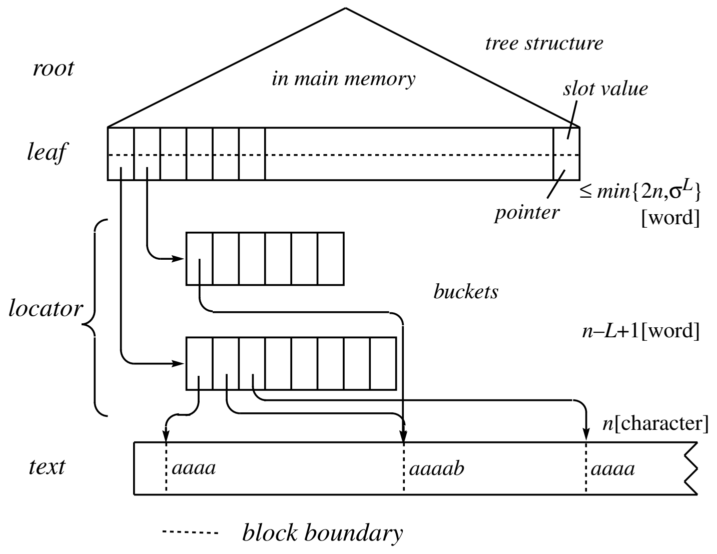


Figure 3: Compressed TS–file.

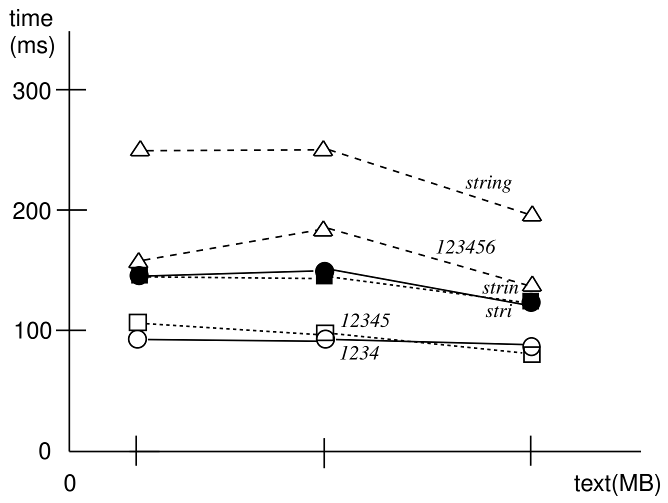
If both **6.2.1**,**6.2.2** are applied, the size of the leaf becomes $\sigma^L \alpha_s \alpha_p$ and the root becomes $\sigma^L \alpha_s / b_f$.

7 Experimental Results

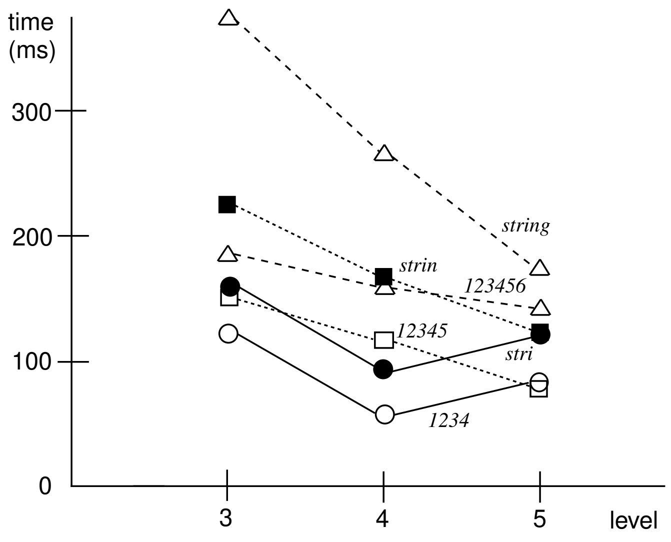
In order to confirm that the proposed data structure and algorithm provide very fast searches and that the compression methods proposed are effective we did experiments and measured the parameters appearing in **6**. The computer mainly used is a SUN Microsystems Sparc Server 630 (28.5MIPS) and the text searched is a UNIX online manual. In **7.3** where second stage compression is applied, we also show the results using network news, whose total size is 100, 300, 500Mbyte, as a source text. Each search time is measured under the condition that the whole root is cached and no parts of the leaf and the locator are cached in main memory initially.

7.1 Basic Structure

Experiments were done for the basic TS–file of **3** first (see Table 1). Fig. 4(a) shows the relationship between text size and search time, and (b) shows the relationship between level and search time. To make the alphabet size small, upper-case letters are converted to lower-case and letters other than numerical or alphabetical are all converted to blanks. That is, $\Sigma=\{0,1,\cdots,9,a,b,\cdots,z,\sqcup\}$ and $\sigma = 37$. Although this manual consists of 2707 separate files (called *manual pages*), we concatenated them into one large non-structured file. In order to change the text size in the experiments, four different size texts were made from the concatenation of 200, 500, 1000, 2707 pages of online manual respectively. As shown in Table 1, the maximum size of these texts is 6.11 Mbyte. Because $37^6 \leq 2^{32}$ and 1 word = 4 bytes in this case, the slot values can be expressed in one word if $L \leq 6$. The pointers to the locator are expressed in one word. The size of the locator and the leaf of each level are shown in Table 1. In the first experiment, we measured the search time for a key set of '1234',



(a) Text size – Time (Level: 5)



(b) Level – Time (Text: 6.1MB)

Figure 4: Basic

'12345', '123456', 'stri', 'strin', 'string'. Among these strings, '1234', '12345', '123456' are used as low selectivity examples, on the other hand, 'stri', 'strin', 'string' are used as relatively hight selectivity examples. The numbers of matches are shown in the 'unix manual' column of Table 2. Since no false drop occurs in basic TS–file search, we don't have to access the text in order to determine the locations where the key appears. Because the case $l_k < L$ is faster than the case $l_k > L$, L is preferably set somewhat larger than the average key length expected. The number of slots, however, grows exponentially with L, so we should take care that the leaf does not become too large by referring to the analysis of **3**. The size of the locator is not related to L. It is four times the text size because one pointer is four times longer than one character (=1 byte). In this experiment, however, it is less than three times as large because the pointers which point to characters which are neither alphabetic nor numeric are not stored in order to save storage space.

level		4				5				signature file			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
text(MB)		.751	1.89	2.80	6.11	.751	1.89	2.80	6.11	.751	1.89	2.80	6.11
leaf(MB)		7.29	7.29	7.29	7.29	270	270	270	270	.092	.232	.343	.750
locator(MB)		2.16	5.43	8.05	17.5	2.16	5.43	8.05	17.5				
time (ms)	1234	69.7	78.1	73.3	62.3	93.2	89.0	93.9	89.8	.42s	.82s	1.2s	2.2s
	12345	111	104	157	119	112	90.0	95.9	81.1	.35s	.67s	.98s	1.9s
	123456	205	159	234	166	163	197	187	133	.34s	.59s	.83s	1.7s
	stri	84.3	88.9	83.4	93.6	148	120	149	120	.39s	.77s	1.1s	2.2s
	strin	191	171	173	170	148	109	146	121	.41s	.78s	1.1s	2.1s
	string	278	262	267	269	257	214	241	184	.41s	.78s	1.1s	2.1s

Table 1: Experimental Results 1 – Basic Structure

	unix manual (page#)				news (MB)		
	200	500	1000	2707	100	300	500
1234	3	5	6	13	125	516	771
12345	2	3	3	8	51	136	223
123456	2	3	3	6	24	87	137
stri	57	151	248	601	9798	15184	21817
strin	40	95	143	376	867	3288	4993
string	40	95	143	376	867	3285	4988
database	–	–	–	–	1082	4540	6706
cryptograph	–	–	–	–	9	75	156

Table 2: Number of Matches

The search time measured is that for finding all addresses of matched strings. It is very fast as predicted in the analysis of **4** and it is less than 300 msec. In particular, it is faster, less than 150msec, when $l_k < L$. Search time does not increase with text size. The time required to search these texts using the signature files ($b = 256$) described in **5.1** is also recorded for reference. In the table 's' indicates that this is the only search measured in seconds. The size of the signature files is written between the rows for leaf and locator in this table. We can read the relationship between l_k, L and measured time qualitatively although, because the times measured are short and apt to include measurement errors, it doesn't necessarily agree with the analysis.

Although a fast search is accomplished with this basic TS–file, the locator and the leaf which constitute the TS–file are quite large and storing them is burdensome.

7.2 Compression by Block Number and Tree Structure

For the first stage compression of the TS–file, page numbers, which are expressed in half word (=2 bytes), instead of locations are put in the locator, and a tree structure is made in order to remove unused slots of the leaf. Table 3(a) shows the experimental results in this case. The size of the locator, leaf and root are also measured in the table. The usage rate u_s of the slots and the average duplication count c of the same string in a logical block of the text which relate to the compression rate of the leaf is measured also. The size of the locator is decreased by 1/4 to 1/10 and the leaf is decreased as per equation (16). The manual pages don't have to be concatenated in this experiment. Each manual page corresponds to a logical block in **6** and the output is page numbers. Search time was measured for the same key set as in **7.1**. But it should be noted that when $l_k > L$ the time measured is until determining the possibility of existence. According to another experiment there are no false drops for the search strings in the table. The result of these experiments shows that the searches are quite a lot faster than those using the basic TS–file, because the amount of data accessed is reduced due to compression.

level		4				5				6			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
$u_s \times 100(\%)$		1.04	1.47	1.81	2.84	.0688	.104	.130	.214	.0034	.0057	.0072	.0122
c		3.06	2.95	2.92	2.74	2.28	2.21	2.22	2.12	1.90	1.84	1.87	1.82
root(kB)		.612	.864	1.06	1.67	1.49	2.26	2.82	4.64	2.76	4.54	5.77	9.80
leaf(kB)		156	220	272	426	381	578	722	1187	704	1163	1477	2509
locator(kB)		353	919	1377	3197	474	1232	1814	4128	570	1475	2151	4828
time (ms)	1234	48.1	58.5	78.4	63.5	61.4	61.2	68.6	64.8	61.1	52.5	72.9	81.2
	12345	48.2	80.1	82.1	66.0	62.7	60.9	68.5	63.8	61.5	52.1	62.2	91.8
	123456	49.0	80.9	77.1	73.6	59.3	83.2	69.4	65.3	69.5	50.3	69.0	89.6
	stri	83.2	94.2	97.9	116	90.2	107	111	115	108	77.3	118	124
	strin	97.1	122	126	151	90.2	116	108	109	114	76.6	126	138
	string	158	187	205	242	94.3	136	149	205	114	76.8	126	135

(a) Compression – 1

level		4				5				6			
page#		200	500	1000	2707	200	500	1000	2707	200	500	1000	2707
p_s		.902	.918	.931	.951	.773	.801	.811	.837	.633	.680	.690	.711
p_p		1.00	.972	.959	.937	1.00	.992	.988	.977	1.00	.997	.996	.991
p_d		.886	.923	.919	.912	.794	.846	.840	.824	.684	.746	.738	.717
root(kB)		.612	.864	1.06	1.67	1.49	2.26	2.82	4.60	2.75	4.54	5.77	9.80
leaf(kB)		44.8	64.1	79.2	124	128	189	234	380	273	431	544	907
locator(kB)		197	495	744	1739	286	710	1053	2426	375	925	1358	3096
space overhead(%)		32.3	29.6	29.4	30.5	55.3	47.7	46.0	46.0	86.7	72.0	68.1	65.7
time (ms)	1234	81.9	62.4	77.7	67.7	68.9	60.0	69.0	65.7	67.9	69.3	66.4	101
	12345	82.3	62.9	79.6	56.7	67.3	61.7	57.2	64.6	65.5	69.7	63.6	101
	123456	72.2	65.7	79.0	74.6	66.3	61.2	71.2	70.2	63.5	66.6	66.5	101
	stri	79.0	85.7	78.0	111	100	93.0	94.1	124	96.8	65.6	119	121
	strin	92.1	112	105	162	110	102	90.4	127	110	74.6	128	135
	string	126	160	187	241	111	106	161	219	111	74.7	128	135

(b) Compression – 2

Table 3: Experimental Results 2

Since a larger L increases the chance of $l_k \leq L$, it decreases search time. Moreover if $L \geq l_k$ we know the page numbers which contain the key without accessing the text because then there are no false drops. So a larger L is more advantageous as long as storage space permits it.

7.3 Compression by Run-length Encoding

For the second stage of compression, we did an experiment in which the locator and the leaf are compressed by run-length encoding (see Table 3(b)). Fig. 5(a) shows the relationship between text size and search time, and (b) shows the relationship between level and search time. The size of the compressed locator and leaf agree with the values which are computed from the equations in **6** with the original size and the compressing probability (p_s, p_p, p_d) measured. When $L = 4$ and the text is 2707 pages, the size of the TS–file (sum of the locator, leaf and root) is 1.86Mbyte which is 30% overhead against 6.11Mbyte (the text size). This ratio is 65.7% when $L = 6$ (see space overhead row of the table). Fig. 6(a) shows how TS–file is compressed by the first and second compression. We also show how search time changes by these compressions in (b).

level		4	6	12	sig
root(MB)		.0191	.115	1.20	—
leaf(MB)		1.23	10.6	111	—
locator(MB)		46.1	67.9	99.7	—
space overhead(%)		47.4	78.6	212	12.3
	1234	95.8	105	84.3	41s
	12345	184	115	83.5	36s
	123456	271	92.4	92.0	37s
time	stri	106	163	183	37s
(ms)	strin	206	133	167	38s
	string	331	134	165	37s
	database	363	290	86.6	38s
	cryptograph	611	505	78.0	13s

Table 4: Experimental Results 3
100MB (Compression–2)

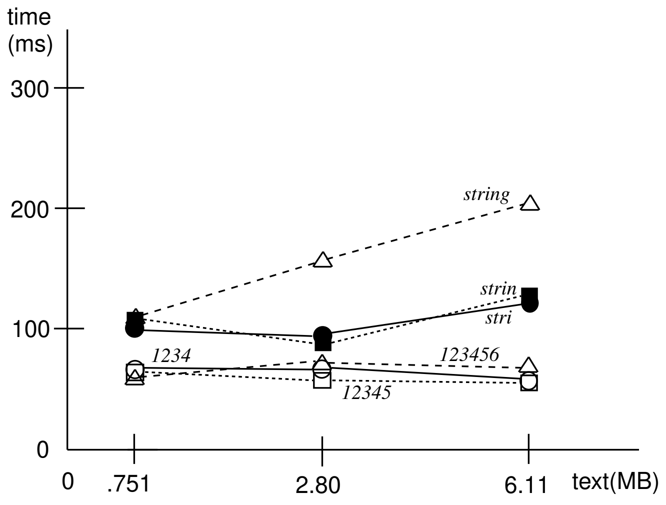
For this compression only, we also tried using 100Mbyte of network news as text (see Table 4). Fig. 7 shows the relationship between level and search time. Creating TS–files for the levels of 4,6 and 12, we measured the space and search time. Slots have twice the length, i.e. 8 bytes, for $L = 12$ only. we added 'database' and 'cryptograph' ($l_k = 8$ and 12 respectively) to the previous list of search strings. The number of matches are shown in the 'news' column of Table 2. 17.5% false drops were observed for the search string 'strin' and level $L = 4$; however, no false drop was observed for any other combinations of search strings and levels.

The ratio of the size of the TS–file to the text size is less than half (47.4%) when $L = 4$ and about twice (212%) when $L = 12$ (see space overhead in Table 4). Considering the average length of words is from five to seven, $L = 12$ is a sufficiently large level. The size of the root is 1.2Mbyte even when $L = 12$, which may easily be put in the main memory. From the fact that ordinary key word indices, which cannot be used for arbitrary string searches, often become bigger than the text and that secondary storage devices are increasing their space and decreasing the price per byte recently, the TS–file is sufficiently small for practical use.

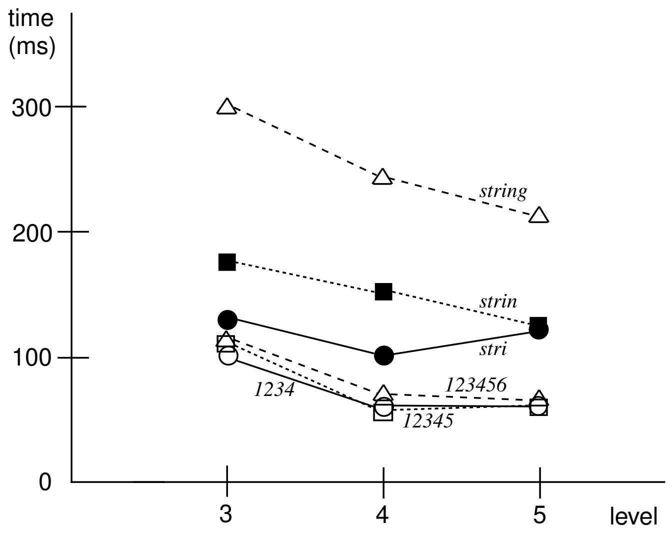
The search time measured is less than 200msec for the strings whose length is less than six when $L = 6$ and for all strings searched when $L = 12$. The search time is very fast for the arbitrary string searches of the 100Mbyte text.

Searches using signature files ($b = 256$) are also recorded for reference. Although the overhead in size (12.3%) is smaller than that of a TS–file, the search speed is very slow.

We also used 300 and 500Mbyte network news as a text with level $L = 12$. We use different computers in this case. A Silicon Graphics Challenge is used to make the TS–files, and a Silicon Graphics Indy R4600 (62.8 Specint92) is used to search for keys. We measured the time required to search for the first matches as well as that for all match (see Table 5). Fig. 8(a) and (b) show the relationship between level and search time for all and first match respectively. It is proved by this experiment that the search is fast even when the text size is quite large and its time depends only on the number of matches.

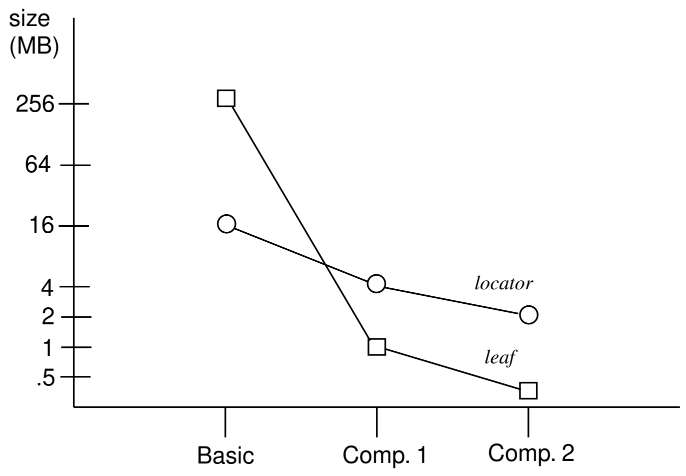


(a) Text size – Time (Level: 5).

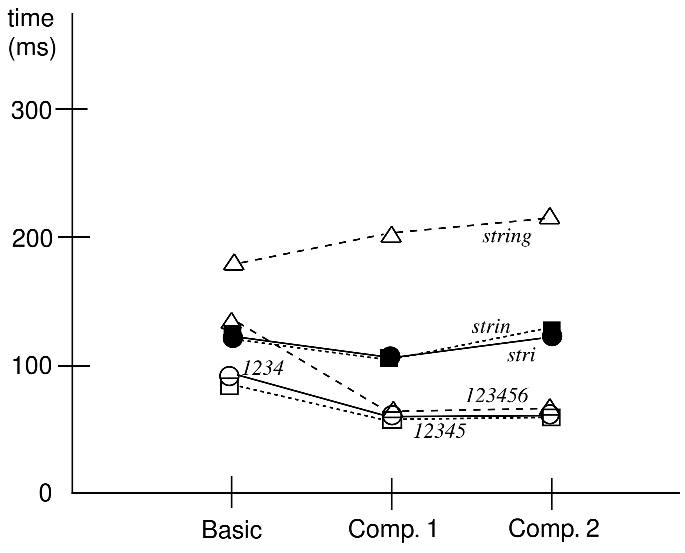


(b) Level – Time (Text: 6.1MB).

Figure 5: Compression



(a) Size of leaf and locator.



(b) Time (Text: 6.1MB; Level: 5).

Figure 6: Basic, Comp. 1 and Comp. 2

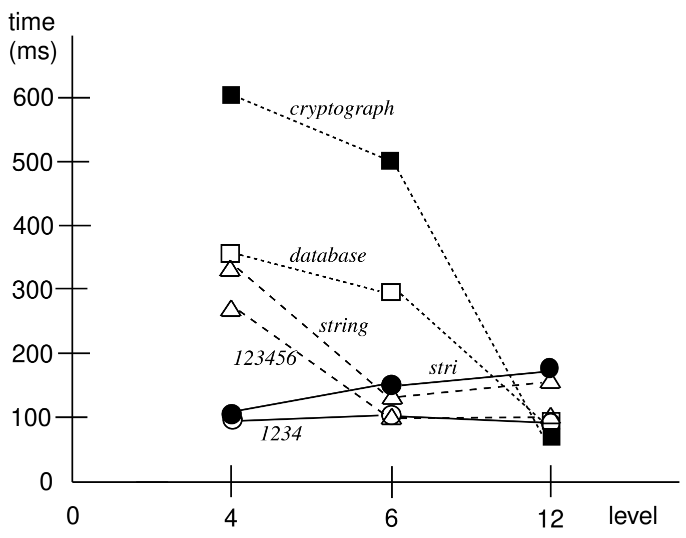


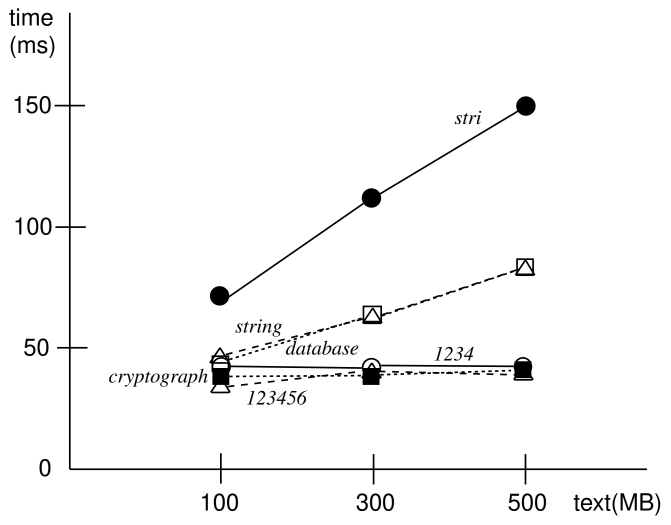
Figure 7: 100MB text Level – Time.

text(MB)		100	300	500
root(MB)		1.20	3.94	7.71
leaf(MB)		111	263	555
locator(MB)		99.7	248	442
space overhead(%)		212	172	201
first match (ms)	1234	33.2	33.1	37.3
	12345	36.6	37.1	36.8
	123456	31.0	37.6	37.2
	stri	40.2	39.2	38.1
	strin	34.6	38.7	41.9
	string	35.7	34.2	38.8
	database	39.1	40.6	37.3
	cryptograph	36.1	36.4	34.6
all match (ms)	1234	41.8	42.1	42.4
	12345	39.0	33.1	35.7
	123456	34.9	41.7	39.2
	stri	72.1	120	152
	strin	43.3	57.7	75.3
	string	47.9	63.1	78.1
	database	45.1	63.9	78.3
	cryptograph	39.6	39.8	41.8

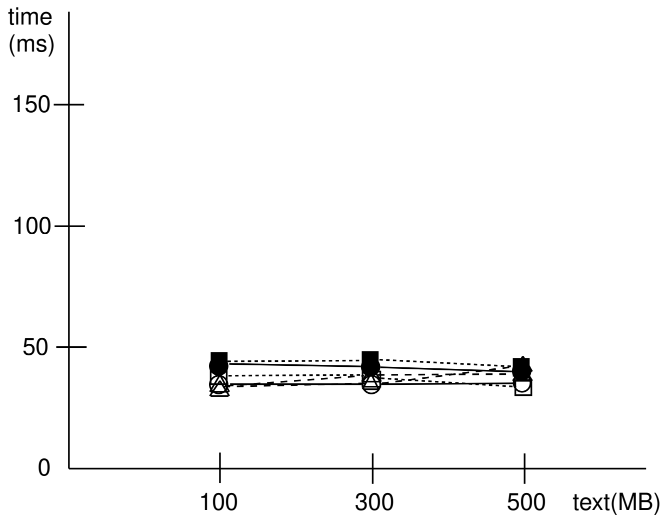
Table 5: Experimental Results 4
$L = 12$ (Compression–2)

8 Conclusions

In this paper we introduced the TS–file, a new gram based data structure for fast full text search, and we gave a set of concrete search algorithms using the TS–file. Because the proposed method can find an arbitrary string using the TS–file alone, the proposed method is more accurate than the one using signature files by which one



(a) all match.



(b) first match.

Figure 8: Text size – Time (Level: 12)

can only know the possibility of existence. We also showed that this method is much faster than searches using signature files by analysis. From the experimental results, we confirmed that sub string matches of rather short strings, which are common in practice, can be done very fast. We were able to reduce the size of the TS–file without losing search speed by introducing two stage compression methods by which the storage required became sufficiently small for practical use.

References

[1] D. E. Knuth, J. H Morris. and V. R. Pratt: "Fast pattern matching in strings", *SIAM J. Comput.*, **6**, 2, pp.322–350 (1977–06).

[2] R. S. Boyer and J. S. Moore: "A fast string searching algorithm", *Commun. ACM*, **20**, 10, pp.762–772 (1977–10).

[3] A. V. Aho and M. J. Corasick: "Efficient string matching : An aid to bibliographic search", *Commun. ACM*, **18**, 6, pp.333–340 (1975–06).

[4] R. M. Karp and M. O. Rabin: "Efficient randomized pattern-matching algorithms", *IBM J. Res. Develop.*, **31**, 2, pp.249–260 (1987).

[5] C. Faloutsos: "Access methods for text", *ACM Comput. Surveys*, **17**, 1, pp.49–74 (1985–03).

[6] J. Aoe: *Computer algorithms – String pattern matching strategies*, IEEE Computer Society Press (1994).

[7] G. Gonnet, R. Baeza-Yates and T. Snider: New indices for text: Pat trees, in *Information retrieval: data structure & algorithms* chapter 5, W. Frakes and R. Baeza-Yates Ed., pp. 66–82 (1992).

[8] H. Shang: *Trie methods for text and spatial data on secondary storage*, Ph.D Dissertation, Faculty for Graduate Studies and Research of McGill University (1995).

[9] M. C. Harrison: "Implementation of the substring test by hashing", *Commun. ACM*, **14**, 12, pp.777–779 (1971-12).

[10] C. Faloutsos: "Signature files: Design and performance comparison of some signature extraction methods", *Proc. 1985 ACM SIGMOD International Conference on Management of Data*, pp.63–82.

[11] D. L. Lee and C. Leng: "A partitioned signature file structure for multiattribute and text retrieval", *Proc. 6th IEEE International Conference on Data Engineering*, Feb. 1990, pp.389–397.

[12] T. Kinoshita, J. Aoe and T. Sato: "A method for speeding up a hash search using a substring test", *Trans. Inst. Electron. Inf. Commun. Eng. D–I*, **J73–D–I**, 5, pp.535–538(1990–05) [in Japanese].

[13] W. W. Chang and H. J. Schek: "A signature access method for starburst database system", *Proc. 15th International Conference on VLDB 1989*, pp.145–153.

[14] T. Sato: "Fast string pattern matching by using compressed data files", *Trans. Inst. Electron. Inf. Commun. Eng. D-I* , **J73–D–I**, 4, pp.451–452 (1990–04) [in Japanese].

[15] Y. Ogawa and M. Iwasaki: A new character-based indexing method using frequency data for Japanese documents, *In Proc. 18th ACM SIGIR Conf.*, Seattle, USA, July 9–13 1995, pp. 121–129.

[16] J. Zobel, A. Moffat and R. Sacks-Davis: "An efficient indexing technique for full-text database systems", *Proc.18th International Conference on VLDB 1992*, pp.352–362.

[17] D. Morrison: PATRICIA – Practical algorithm to retrieve information coded in alphanumeric, *J.ACM,*, **15**, 4,, pp. 514–534 (1968).

[18] D. Knuth: *The art of computer programming: Vol.3 sorting and searching*, Addison-Wesley, Reading, Mass., pp. 490–499 (1973).

[19] C. Clifton and H. Garcia-Molina: "Indexing a hypertext database", *Proc.16th International Conference on VLDB 1990*, pp.36–49.

[20] R. Sacks-Davis, T. Arnold-Moore and J. Zobel: "Database systems for structured documents", *Proc. Int. Symp. Advanced Database Technologies and Their Integration ADTI'94*, Nara, Japan, Oct. 26–28 1994, pp.272–283.

[21] T.Sato: "Fast Full Text Search with Free Word Using TS–file", *Proc. 19th ACM SIGIR Conf.*, Zurich, Switzerland, Aug. 18–22 1996, p.342.