Yugoslav Journal of Operations Research
9 (1999), Number 2, 235-256

EFFICIENT IMPROVEMENT OF BRAIN-THARP'S ALGORITHM

Dejan SIMIĆ

Mihajlo Pupin Institute, "IMP-Računarski sistemi", d.o.o.
Volgina 15, 11060 Belgrade, Yugoslavia

Dušan STARČEVIĆ

Faculty of Organizational Sciences
Jove Ilića 154, 11000 Belgrade, Yugoslavia

Abstract: In this paper we present possible improvement of the best known perfect hashing algorithm. A comparison of the proposed algorithm with other known important perfect hashing algorithms in addition to Brain-Tharp's algorithm is also given. The Brain-Tharp's algorithm is the best known in the special class of algorithms that can be used to form ordered minimal perfect hash functions for very large word lists in terms of function building efficiency, pattern collision avoidance and retrieval function complexity. However, building a perfect hash function by the Brain-Tharp's algorithm is still extremely slow. In this paper we analysed important features of Brain-Tharp's algorithm and proposed three solutions to improve the total processing time of the packing phase. The proposed techniques are validated empirically. The improvement factor is close to 2 on the example of the standard UNIX dictionary.

Keywords: Algorithms, file structures, perfect hashing, physical design.

1. INTRODUCTION

A hashing algorithm or hash function is a means of calculating the disk address of a block containing a given record from the value of its key [17]. Thus, the software technique of direct access or key to address transformation is called hashing. Besides being a classical problem in computer science, hashing is applied in a wide range of applications such as compilers (symbol tables and code convertor tables), file organizations [14] (large databases), data mining [13], hardware applications [15], and so on. Perfect hashing and minimal perfect hashing are especially interesting and useful classes of hashing with a guaranteed single disk access retrieval. Perfect hashing is an injection or one-to-one type of mapping, that is, a different address corresponds to each key [11, 12, 3, 2, 5]. If there are no unused addresses then it is minimal perfect hashing [9, 10, 4, 8]. Hashing algorithms, which preserve the order of records after hashing, are called order preserving or ordered hashing algorithms.

The first perfect hashing algorithm was introduced by Sprugnoli [19]. Two techniques - the quotient reduction method and the remainder reduction method were developed. The hash function h for the quotient reduction method is

$$h(w) = \left\lfloor \frac{w + c_1}{c_2} \right\rfloor, \tag{1}$$

and for the remainder reduction method is

$$h(w) = \left\lfloor \frac{(c_1 + w * c_2) \bmod c_3}{c_4} \right\rfloor, \tag{2}$$

where w is the word to be hashed, and c_1, c_2, c_3, and c_4 are constants calculated by the algorithm. Sprugnoli's algorithm is unique in the fact that it does not require auxiliary tables at retrieval time - only the four constants must be stored. Without segmentation Sprugnoli's algorithm is able to handle small word sets (10-12 words).

Cichelli created a minimal perfect hashing algorithm [6] that can handle up to 50 words, wherein no two words can have the same length and the same first and last letters. The form of his minimal perfect hash function is:

$$\begin{aligned} h(w) = {} & key\_length(w) + \\ & associated\_value\_of\_the\_first\_letter(w) + \\ & associated\_value\_of\_the\_last\_letter(w) \end{aligned} \tag{3}$$

Although Cichelli's algorithm is very simple, efficient and machine independent, the main disadvantage is its exponential time complexity $O(c^n)$ [6], where n is the number of words to be hashed.

Sager introduced a minicycle algorithm [16] with polynomial time complexity $O(n^6)$ and thus improved Cichelli's algorithm. Sager's algorithm is able to handle up to 512 words. The form of Sager's perfect hash function is:

$$h(w) = (h_0(w) + g \bullet h_1(w) + g \bullet h_2(w)) \bmod N\ , \tag{4}$$

where w is the word to be hashed, h_0, h_1, and h_2 are three quickly computable pseudorandom functions, N is a nonnegative integer greater than or equal to n which is the number of words to be hashed, and g is a function to be determined by the algorithm.

Karplus and Haggard generalized Cichelli's algorithm using graph theory methods [10]. Their algorithm can hash word sets of up to almost 700 words, but the words in the sample lists seem to be carefully chosen. Experimental results have shown that time complexity for Karplus-Haggard's algorithm is $O(n^{1.5})$.

Fox, Chen, Heath, and Datta [7] improved Sager's algorithm in such a way that minimal perfect hash function can be generated for word sets of 1000 words. Fox, Chen, Heath, and Daoud [8] have demonstrated a perfect hashing algorithm for very large word lists. The algorithm is $O(n)$ at build time and the hash tables needed at retrieval time are very small.

So far, in the class of algorithms that can be used to form ordered perfect hash functions, the best one is Brain-Tharp's algorithm [1] in terms of hash function building efficiency, pattern collision avoidance, and retrieval function complexity. It is important to note that unlike the algorithm introduced by Fox, Chen, Heath, and Doud [8], Brain-Tharp's algorithm produces an ordered perfect hash function, and it is much less expensive at retrieval time.

The rest of the paper is organized as follows. Section 2 introduces the problem of the efficient construction of ordered perfect hash functions. Section 3 describes Brain-Tharp's algorithm. Section 4 presents three possible solutions for significant improvement of Brain-Tharp's algorithm. Experimental results presenting the performance of the original BT and modified algorithms BT_BM, BT_IS and BT_HS are given in Section 5. A comparative analysis of the original BT algorithm with other known perfect hashing algorithms is shown in section 6. Section 6 also compares the BT algorithm to the BT_HS algorithm, which has better performance than other versions of the BT algorithm, using the example of a simulated dictionary with 100,000 words. In section 7 we conclude the paper.

2. PROBLEM DESCRIPTION

The terminology used in the paper is shown in Table 1. We assume that a collection of objects is given in a database and each object has a unique associated identifier k, which we will call a key. Each key is a character string having maximum length *Lmax* characters. This assumption is appropriate for keys that are words in any natural or artificial language [8]. The characters are elements of the alphabet Σ and all keys are elements of the universe of keys $U = \{k_1', k_2', k_3', \ldots, k_N'\}$, where N is some finite nonnegative integer. We assume also, that $A = \{k_1, k_2, k_3, \ldots, k_N\}$ is a set of actually used keys in a database, where n is a nonnegative integer and $n < N$. We access the objects via a hash function $h : A \to M$, which maps a set of keys onto a set of memory addresses. The set of memory addresses M is defined in the following way $M = \{address_1, address_2, address_3, \ldots, address_m\}$, where m is a nonnegative integer and $m \geq n$. The addresses in M are nonnegative integers. The objects are stored in the corresponding memory addresses. Bearing in mind the previous assumptions we give the following definitions for perfect, minimal perfect and ordered perfect hash functions.

Definition 1. *A perfect hash function is an injection* $h: A \to M$, *where* $m \geq n$ *[12].*

Definition 2. *A perfect hash function* $h: A \to M$ *is minimal if* $m = n$.

Definition 3. *If for any two keys,* k_i *and* k_j, *from A we have that* $i < j$ *such that* $h(k_i) < h(k_j)$ *then the perfect hash function h is order preserving.*

Table 1: Terminology

DEFINITION	NOTATION
The alphabet	Σ
i-th key value	k_i
Maximum length of a key (maximum number of characters)	$Lmax$
Universe of keys	U
A set of actually used keys	A
The cardinality of the universe of keys	N
The cardinality of the actually used set of keys	n
Hash function	h
i-th memory address	$address_i$
A set of memory addresses	M
The cardinality of the set of memory addresses	m
Average seek time	s
Average rotational latency	rl
Block transfer time between disk and primary memory	btt

It is well known that perfect hash functions are rare in the set of all functions. For example, Knuth [11] pointed out that only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses. Due to their definition the number of ordered perfect hash functions is smaller than the number of perfect hash functions. So, searching for ordered perfect hash functions is not easy and fast, especially for large sets of objects.

In this paper we address the problem of the efficient practical construction of an ordered perfect hash function for a given set of objects. So far, Brain-Tharp's algorithm is the best one in this class of algorithms. Our goal is to improve the total time needed to construct an ordered perfect hash function for a given set of objects.

3. BRAIN-THARP'S (BT) ALGORITHM

Brain-Tharp's (BT) algorithm is a perfect hashing algorithm, which can be used to create ordered minimal perfect hash functions for large word lists. The main implementation techniques of the BT algorithm are array-based tries and a sparse matrix packing algorithm [1]. The pseudocode of the BT algorithm is shown in Fig. 1.

```
Build the 2-D trie array.

Sort the columns of the 2-D array. The columns with the most entries should
be at the beginning of the sorted list, and other columns should fill the list in
decreasing order.

Loop through all columns containing 1 or more values.

        Attempt to place the current column at the first location of the 1-D array.

        While the new column's values collide with values already in the 1-D array,
        move the column one position down in the 1-D array and check for
        collisions again.

        Once a non-colliding position for the new column is found, place the
        column in the 1-D array and update the CLT (Column Lookup Table).

Save the final  CLT formed.
```

Figure 1: Pseudocode of Brain-Tharp's (BT) algorithm.
The CLT holds information about the starting location of each column in the 1-D array.

In the BT algorithm the input is a large word list, wherein no two words have exactly the same characters in all positions. The algorithm consists of 3 phases: a 2-D (dimensional) array-based trie creating, the 2-D array's column sorting, and packing of the sorted columns into the 1-D array. A very important characteristic of the BT algorithm is that in all phases processing is sequential. Therefore, in the next section of this paper in order to obtain an improved version of the BT algorithm we will introduce some form of parallelization.

3.1. Example

A simple example is now provided to illustrate the main features of the BT algorithm. Figure 2 shows a 2-D array-based trie for a word list containing the words ALAN, ALGORITHMS, CAT, COP, CORN, DEJAN, and DUSAN. The words are the keys. For the given example the words are alphabetically arranged. This is not necessary and the words can be ordered in any way.

	1	2	3	4	5	6
A	-2		1	3		
B						
C	-4					
D	-6					
E						6
F						
G			2			
H						
I						
J						
K						
L		-3				
M						
N						
O				-5		
P					4	
Q						
R					5	
S						
T						
U						7
V						
W						
X						
Y						
Z						
eow						

<u>Word List</u>

1 ALAN

2 ALGORITHMS

3 CAT

4 COP

5 CORN

6 DEJAN

7 DUSAN

Figure 2: An example of the 2-D array-based trie

	1	3	4	5	6	2
A	-2	1	3			
B						
C	-4					
D	-6					
E					6	
F						
G		2				
H						
I						
J						
K						
L						-3
M						
N						
O			-5			
P				4		
Q						
R				5		
S						
T						
U					7	
V						
W						
X						
Y						
Z						
eow						

Criterion for column sorting

the number of non-empty fields

Figure 3: Sorted columns of the 2-D array shown in Fig. 2

The trie shown in Fig. 2 is a matrix representation for an initial data structure of the BT algorithm. The elements of the 2-D array-based trie are pointers to columns and addresses of locations where words are stored. In the trie positive values point to the word list, while negative values point to the next column. The number of bytes needed for the address representation depends on the number of words in the given word list. For word lists of up to 32767 words 2 bytes are sufficient. For the sake of simplicity, the rows of the trie are denoted by the alphabet's 26 upper case letters and by eow which stands for "End of Word". The main disadvantage of the 2-D array structure is the low memory utilization. The memory utilization is only 8.8% for the trie array shown in Fig. 2.

The second phase starts with the columns of the 2-D trie and ends with the columns sorted. Figure 3 shows the sorted columns of the 2-D trie shown in Fig. 2 by the number of non-empty fields in descending order [21].

The time complexity of the third phase is $O(r^2)$, where r is the number of columns of the 2-D array. Once the columns are sorted, the third phase begins. The third phase starts the column packing into the 1-D array. The column packing is performed from the first location of the 1-D array using the first-fit strategy. The only constraint on the packing process is that no two values from different columns can occupy the same position in the 1-D array. After column packing into the 1-D array the relative distance between column elements is preserved. The variable part is the position of each column inside the 1-D array. In order to hold the starting position of each column within the 1-D array, a column lookup table (CLT) is used. The number of fields in CLT is equal to the number of columns in the 2-D array. Figure 4 shows the 1-D array and associated CLT for the 2-D array shown in Fig. 3.

Column Lookup Table

0	-2	1	4	-9	1

1-D Packed Trie

-2	1	-4	-6	3	6	4	2	5	-3									-5			7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

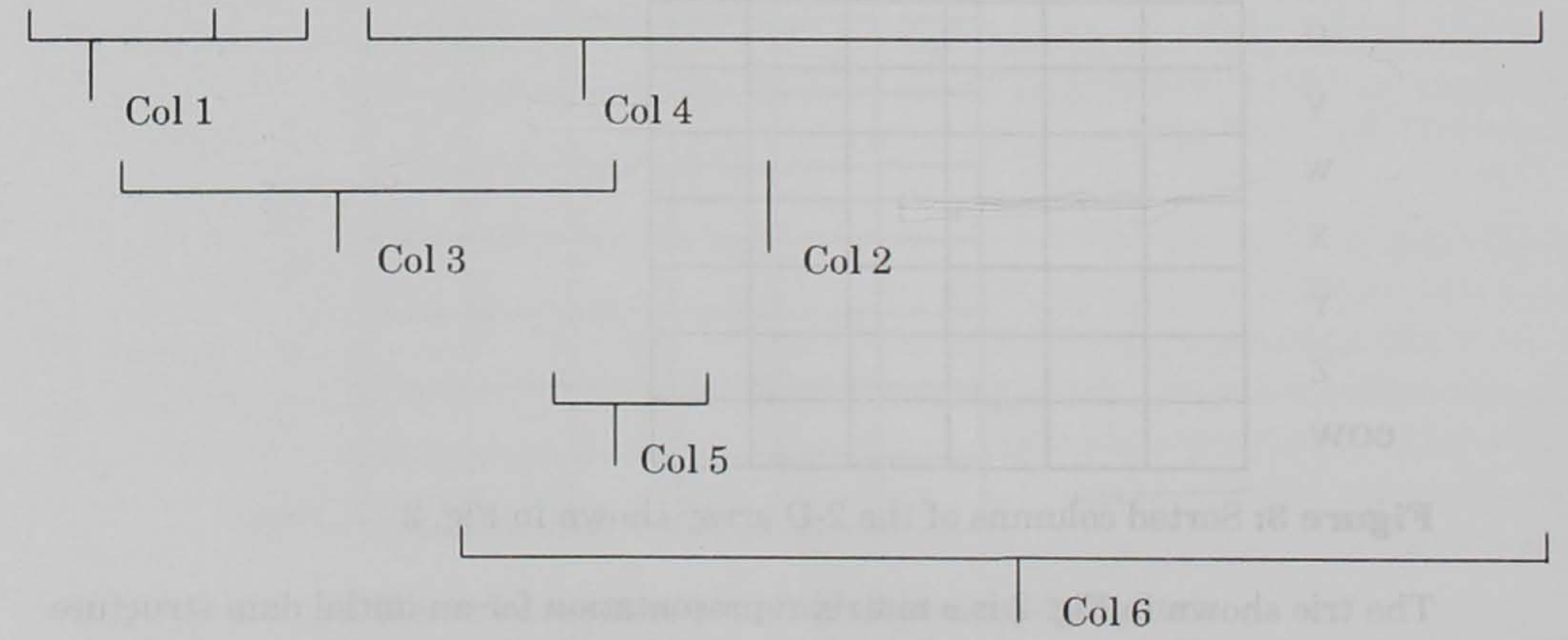


Figure 4: The 1-D array and Column Lookup Table for the 2-D array shown in Fig. 3

As the last step in the third phase, the CLT is incorporated into the 1-D array and only the starting position for the first column is stored on disk. Remembering the starting location for column 1 is needed because all words must index through column 1. The 1-D array is minimal if all its fields are populated. The 1-D array shown in Fig. 4 is not minimal because it contains empty fields. Its memory utilization is 54.54%. Moreover, for static databases it is very important to produce a minimal 1-D arrays. Although in most cases minimal 1-D array is produced, it cannot be guaranteed for every word list.

Experimental results published in [1] suggest that the column packing phase is dominant and extremely slow. For instance, packing the columns takes 4231 seconds for the standard UNIX dictionary with 24481 words using a SB 8000 machine. Taking into consideration that the third phase of the BT algorithm takes too much time, our goal is to find a way to speed up the column packing phase. Every solution that decreases the column packing time represents a significant contribution to the total improvement of the BT algorithm. The rest of the paper is focused on speeding up the BT algorithm.

4. IMPROVEMENT OF THE BT ALGORITHM

In the BT algorithm finding a set of initial column packing positions that minimizes the size of the 1-D array is a NP-complete problem [21]. A good heuristic algorithm for 1-D array column packing with the least amount of required time is still an open problem. In addition, determination of the actual size of the 1-D array is not possible before column packing. Theorem 1, given in the next section, is a good estimation of the size of the 1-D array.

Columns with one or two non-empty fields are easier to pack densely into the 1-D array than columns with many non-empty fields. Attempting to find a better way of column packing, we did not find a solution better than Ziegler's first-fit decreasing method. Namely, investigating the run-time activities of various column packing strategies showed no improvement. For example, we compared Ziegler's method to the two pass method, where in the first pass columns are simply sequentially copied into the 1-D array without testing for collision, and in the second pass columns are packed with testing for collision. In the first pass the number of non-empty fields in the 2-D array limits the size of the 1-D array. Ziegler's method had better performance in all our tests.

As can be seen from previous experimental results [1], the column packing phase takes most of the time. Taking this into consideration, we propose column packing improvements by introducing three new solutions. The first solution is based on *bit manipulation* instead of integer manipulation as a packing implementation technique, because bit manipulation provides parallel processing on the single processor architecture. The second solution is implemented with *intelligent setting of the initial column position* in the 1-D packing array. Simply, we eliminate impossible initial positions for columns in the 1-D array. And the third solution is a combination of bit manipulation and intelligent setting of the initial column position. In this Section we will describe all of these solutions.

4.1 Improvement of the BT algorithm by bit manipulation

The bit manipulation technique is introduced by analysing the essence of the BT algorithm. The essence of the BT algorithm is that it is character-based in the first phase and integer-based in the second and third phase. Bearing this in mind, we investigated a new packing paradigm, which reduces the processing time by simultaneously processing a larger number of elements of the 2-D trie. In fact, a very efficient improvement of the BT algorithm denoted as BT_BM can be achieved by the parallelization of column element processing. The major point of the solution based on bit manipulation is that bit representatives are created for all columns of the 2-D array. Parallelization of column element processing is achieved using bit manipulation on the bit representatives.

We divide the column packing phase in the BT algorithm into 2 steps. In the first step bit representatives are created for all columns of the 2-D trie. In the second step the column binary bit representative is packed instead of real column packing. After the position for the column packing is found, the real column is packed. A pseudocode of the proposed improvement of the packed trie algorithm is shown in Fig. 5. The main technique of the algorithm combines basic bit manipulation operations.

```
Build the 2-D trie array.

Create bit representatives for columns of the 2-D array.

Sort the columns of the 2-D array. The columns with the most entries should be at
the beginning of the sorted list, and other columns should fill the list in decreasing
order.

Loop through all columns containing 1 or more values.

        Attempt to place the current column bit representative at the first location
        of the auxiliary vector AV.

        While the new column bit representative's bits collide with the bits already
        in the AV move the column bit representative one position down in the AV
        and check for collisions again.

        Once a non-colliding position for the new column bit representative is
        found, place the column bit representative in the AV, the column in the 1-D
        array and update the CLT (Column Lookup Table).

Save the final CLT formed.
```

Figure 5: Pseudocode of the proposed improvement of the packed trie algorithm [1] by bit manipulation

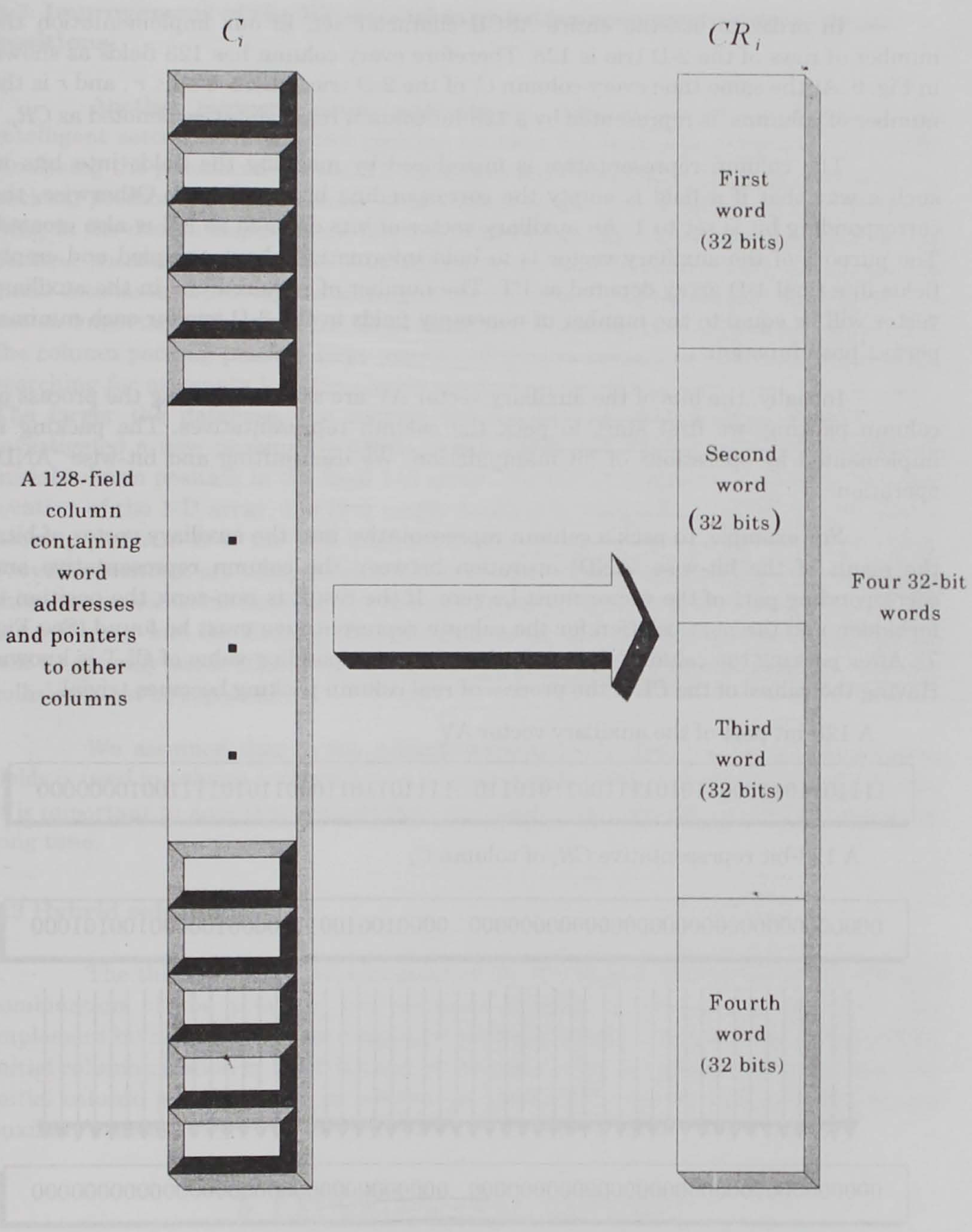


Figure 6: Introducing a 128-bit representative CR_i for each column C_i, where $1 \le i \le r$ and r is the number of columns in a 2-D trie. Each occupied field of the column C_i is mapped to binary 1 and each empty field is mapped to binary 0.

In order to use the entire ASCII character set, in our implementation the number of rows of the 2-D trie is 128. Therefore every column has 128 fields as shown in Fig. 6. At the same time every column C_i of the 2-D trie, where $0 \leq i \leq r$, and r is the number of columns, is represented by a 128-bit column representative denoted as CR_i.

The column representative is introduced by mapping the fields into bits in such a way that if a field is empty the corresponding bit is set to 0. Otherwise, the corresponding bit is set to 1. An auxiliary vector of bits denoted as AV is also created. The purpose of the auxiliary vector is to hold information about occupied and empty fields in a final 1-D array denoted as PT. The number of required bits in the auxiliary vector will be equal to the number of nonempty fields in the 2-D trie for each minimal perfect hash function.

Initially, the bits of the auxiliary vector AV are set to 0. During the process of column packing, we first start to pack the column representatives. The packing is implemented by operations of bit manipulation. We use shifting and bit-wise "AND" operation.

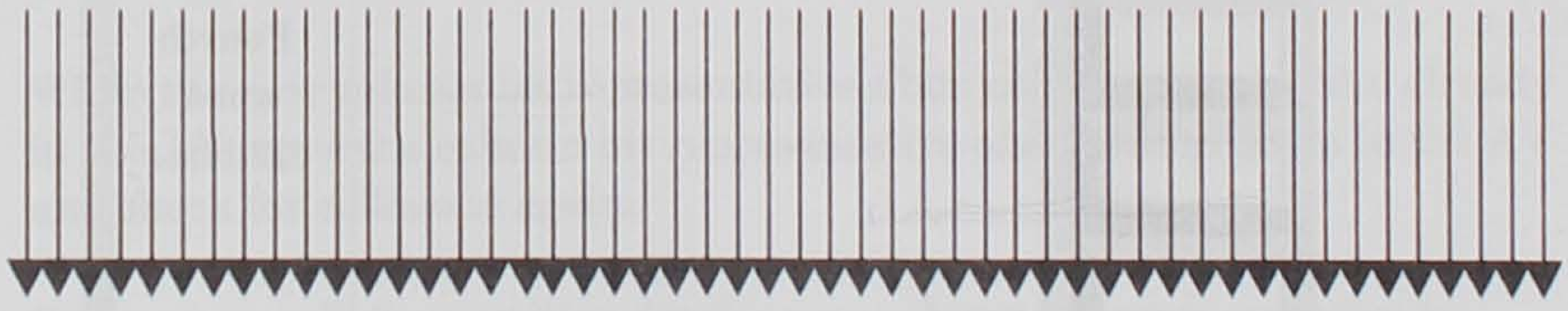
For example, to pack a column representative into the auxiliary vector of bits, the result of the bit-wise "AND" operation between the column representative and corresponding part of the vector must be zero. If the result is non-zero, the position is forbidden and the next position for the column representative must be found (See Fig. 7). After packing the column representative, the corresponding value of CLT is known. Having the values of the CLT, the process of real column packing becomes trivial.

A 128-bit part of the auxiliary vector AV

11110110110001101011110011010110...11110110110001101011110010000000

A 128-bit representative CR_i of column C_i

00000000000000000000000000000000...00001001001110000100000100101000



00000000000000000000000000000000...00000000000000000000000000000000

The 128-bit result R of the bit-wise "AND" operation between the above two operands

Figure 7: The result of the bit-wise "AND" operation between operands (128 bits). The result R equal to zero means the right position. Otherwise, the position is forbidden.

4.2. Improvement of the BT algorithm by better setting of initial packing positions

Another proposed simple and efficient solution denoted as BT_IS is the intelligent setting of the initial packing position for each column in the 1-D array. Analysing the pseudocode of the BT algorithm shown in Fig. 1 it can be seen that the packing position for each column is initialised to the first location of the 1-D array. Also, in the double displacement method described in [20, 21] and suggested in [1], the packing position for each row is initialised to the first location of the 1-D array. For small databases improving the setting of the initial packing position for each column is not so important. However, for large databases it can be very important. Namely, in the column packing phase a large amount of time is wasted for each column because searching for an empty location always starts from the first location of the 1-D array. The larger the database, the greater the amount of time wasted. Therefore, we investigated a new packing paradigm, which eliminates the wasted searching for an initial column position in the final 1-D array. Instead of always starting from the first location of the 1-D array, the first empty location is assigned to be the initial column packing position. It is the first position that might be used for the column packing. Previous positions are occupied and therefore cannot be used for packing the column. If there are no collisions between the filled elements of the new column being placed into the 1-D array and the already filled elements of the 1-D array then the initial position will be actually used. Otherwise, the initial position must be changed and the test for collision must be repeated.

We assumed that in the column packing phase *list*(i), a list of the nonzero fields is used in column i where $1 \leq i \leq r$ and r is the number of columns in a 2-D trie. It is important to note that without lists, the column packing phase takes an extremely long time.

4.3 Hybrid solution

The third proposed improvement of the BT algorithm denoted as BT_HS is a combination of the previous two solutions created to improve performance. We implement bit manipulation on column bit representatives and intelligent setting of the initial column position in the 1-D array at the same time. It is important to say that the initial column position is set in relation to column bit representative packing in the auxiliary vector AV.

5. EXPERIMENTAL RESULTS

Experimental results are given for the following versions of the BT algorithm:

BT - the original BT algorithm,

BT_BM - modified BT algorithm with column bit representatives and bit manipulation, BT_IS - modified BT algorithm with intelligent setting of the initial column position in the 1-D array, and without column bit representatives and bit manipulation, and

BT_HS - modified BT algorithm with intelligent setting of the initial column position in the 1-D array, and with column bit representatives and bit manipulation.

The BT algorithm and its modified versions BT_BM, BT_IS and BT_HS algorithms are all implemented in C programming language. The *radix sort* method is used for the 2-D array's column sorting [21]. Performance testing was performed on a standard UNIX dictionary with 25143 words usually found in the file */usr/dict/words*. For all tests we used an unloaded Pentium/166MHz computer with 64 MB memory under SCO UNIX V 3.2v.4.2 operating system. We analysed the impact of the number of words on algorithm performance. Minimal 1-D arrays are obtained in all tests. Experimental results represent average performance measures for 5 tests.

Each column of the 2-D array has at least one non-empty field. A histogram presenting the number of columns as a function of the number of nonempty fields is given in Fig. 8 for the whole standard UNIX dictionary. The total number of columns of the 2-D trie is 17532. The number of columns containing 1 nonempty field is 5472; 7562 columns contain two nonempty fields, 2112 columns contain three nonempty fields, and so on. The maximum number of nonempty fields is 61 in a column. The total number of nonempty fields is 42674.

In general, the exact number of fields in the compressed 1-D array is not easy to determine before the column packing phase. The following theorem gives a good estimate of the size of the 1-D array. Let t be the number of nonempty fields in the 2-D array, let the function $t(l)$, for $l \geq 0$, be the total number of nonempty fields in columns with more than l nonempty values, and let the matrix representation of a trie be sparse. By a sparse matrix, we mean a matrix in which the number of nonempty fields is much less than the size of the matrix itself. Given the above definitions Tarjan and Yao [21] proved the following theorem:

Theorem 1: *Suppose the 2-D array has the following "harmonic decay" property:*

For any l, the number of nonempty fields in columns with more than l nonempty value is at most $t/(l+1)$, that is

$$t(l) \leq \frac{t}{l+1} \tag{5}$$

Then every column displacement $c(j)$ computed for the 2-D array by the first-fit decreasing method satisfies:

$$0 \leq c(j) \leq t \tag{6}$$

Proof: For any column j, consider the choice of $c(j)$. Suppose $c(j)$ contains $l \geq 1$ nonempty values. By the harmonic decay property the number of nonempty values in previous columns is at most t/l. Each such nonempty value can block at most l choices for $c(j)$. Altogether at most t choices are blocked, and $0 \leq c(j) \leq t$.

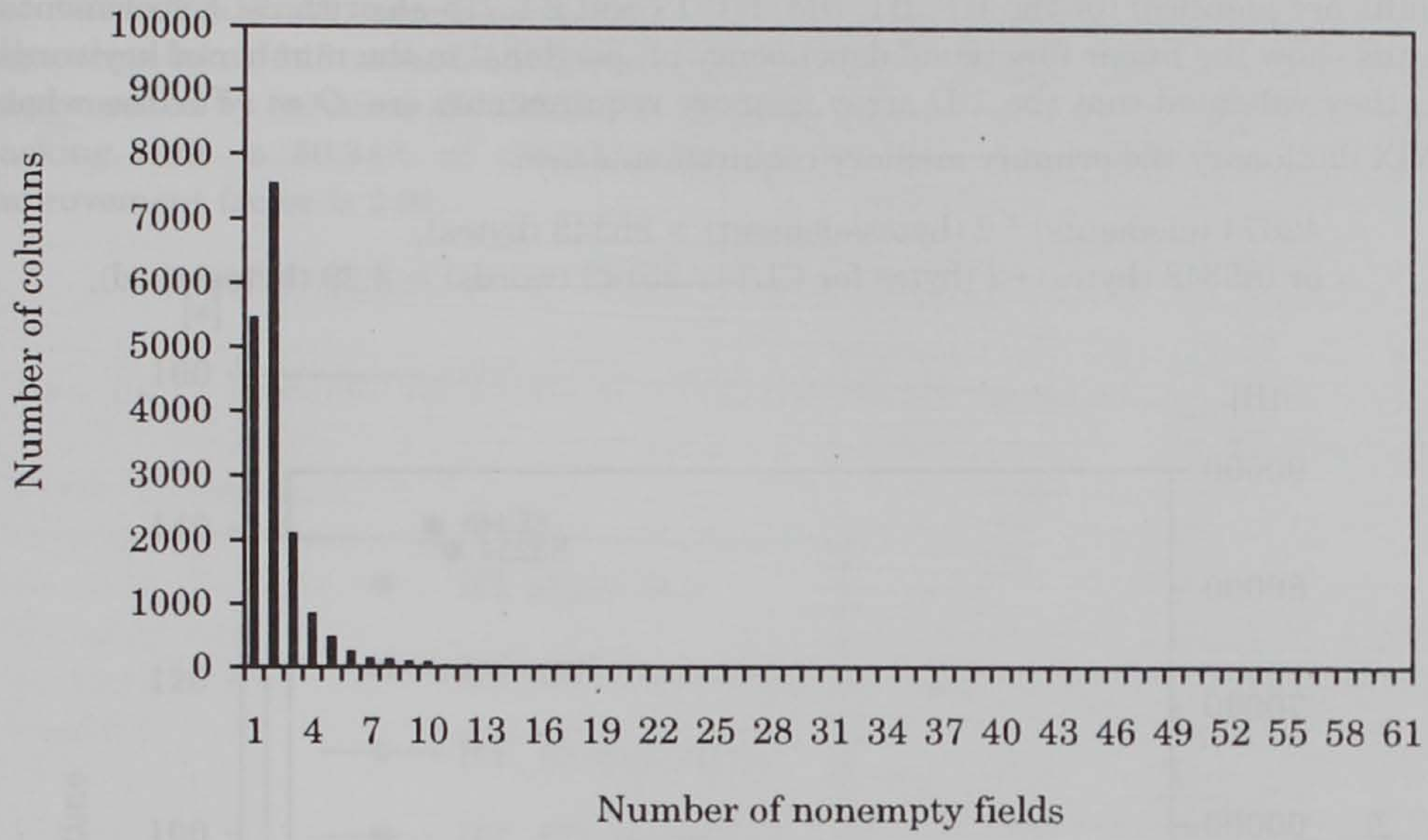


Figure 8: Histogram for columns of the 2-D array-based trie for standard UNIX dictionary

The harmonic decay property requires that at least half of the non-empty values come from columns with one non-empty value. The time complexity of the procedure to determine the harmonic decay property is linear and proportional to the number of columns in the 2-D array. In the case of the standard UNIX dictionary the 2-D array does not have the harmonic decay property. In other words, $t(1) = 37202$, and $t(1) \leq 42674/2 = 21337$ is not true. However, after column packing a minimal 1-D array is obtained. In order to have a guaranteed single disk access, generated 1-D arrays should be located in the primary memory at retrieval time.

As the BT, BT_BM, BT_IS and BT_HS algorithms provide data retrieval with guaranteed single disk access, average time needed to retrieve data is

$$T_1 = s + rl + btt , \tag{7}$$

where s is average seek time, rl is average rotational latency, and btt is a block transfer time. If data are stored on several consecutive blocks then average time needed to retrieve data is

$$T_2 = s + rl + b * btt\ , \tag{8}$$

where b is the number of disk blocks, where desired data are stored.

Fig. 9 shows the 1-D array memory requirements measured in bytes as a function of the number of keywords from the standard UNIX dictionary. The shown results are identical for the BT, BT_BM, BT_IS and BT_HS algorithms. Experimental results show the linear functional dependency proportional to the number of keywords, i.e., they validated that the 1-D array memory requirements are $O(n)$. For the whole UNIX dictionary the primary memory requirements are:

42674 (elements) * 2 (bytes/element) = 85348 (bytes),
or (85348 (bytes)+2 (bytes for CLT))/ 25143 (words) = 3.39 (bytes/word).

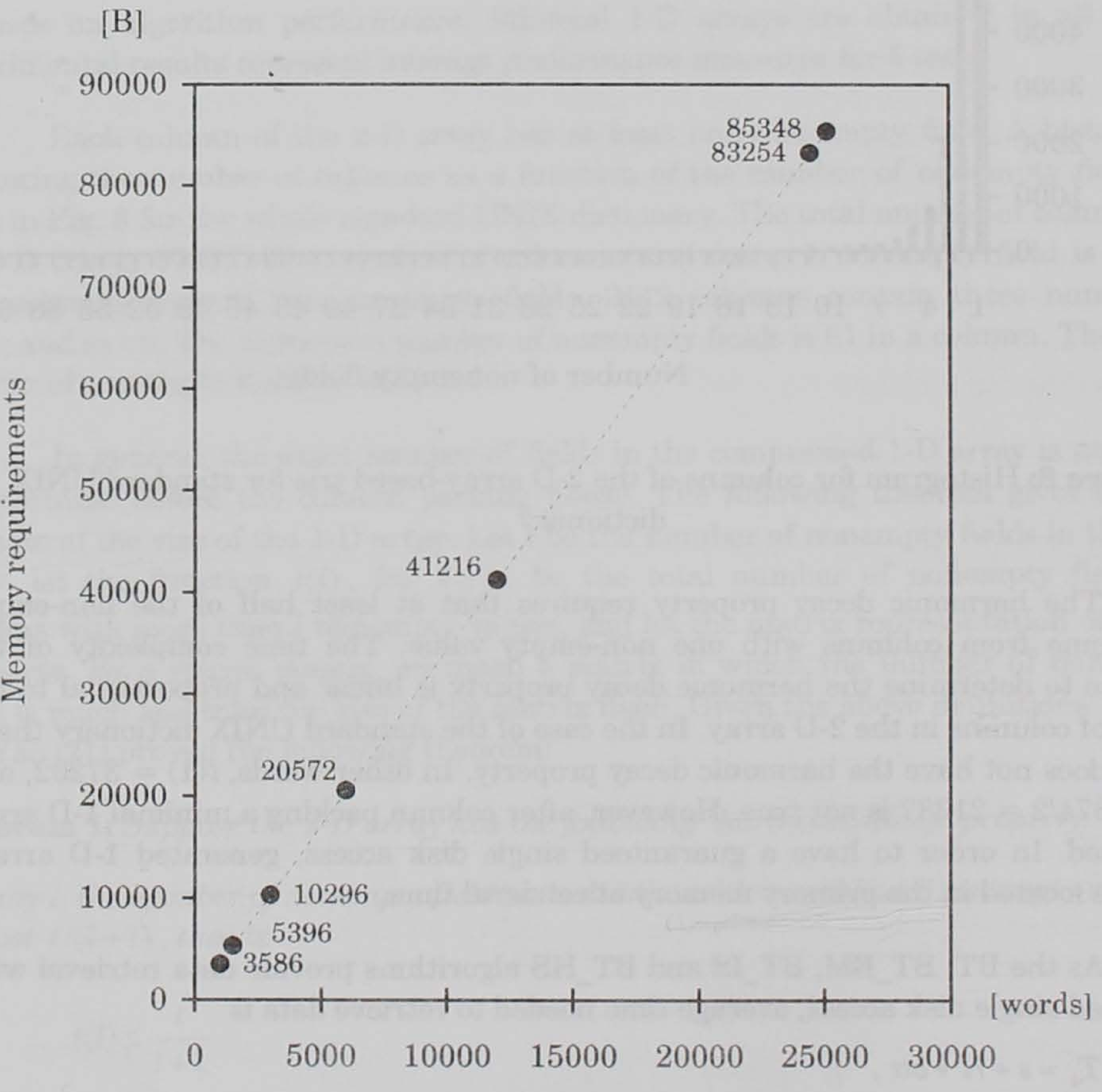


Figure 9: The 1-D array memory requirements as a function of the number of keywords from the standard UNIX dictionary.
Experimental results are identical for the BT, BT_BM, BT_IS and BT_HS algorithms.

The primary metric used in this paper is run time. Figure 10 shows column packing time in seconds as a function of the number of words for the BT, BT_BM, BT_IS and BT_HS algorithms. The column packing time is measured for the first 1000, 1500, 3000, 6000, 12000, 24481, and 25143 words of the standard UNIX dictionary. The modified algorithms are better than the original algorithm in all tests. In the case of the whole UNIX dictionary the column packing phase implemented by the original algorithm takes 145 seconds. Under the same conditions the column packing phase implemented by modified algorithm BT_HS takes 72 seconds. Improvement of column packing time is 50.34% of the time needed by the original algorithm, i.e., the improvement factor is 2.01.

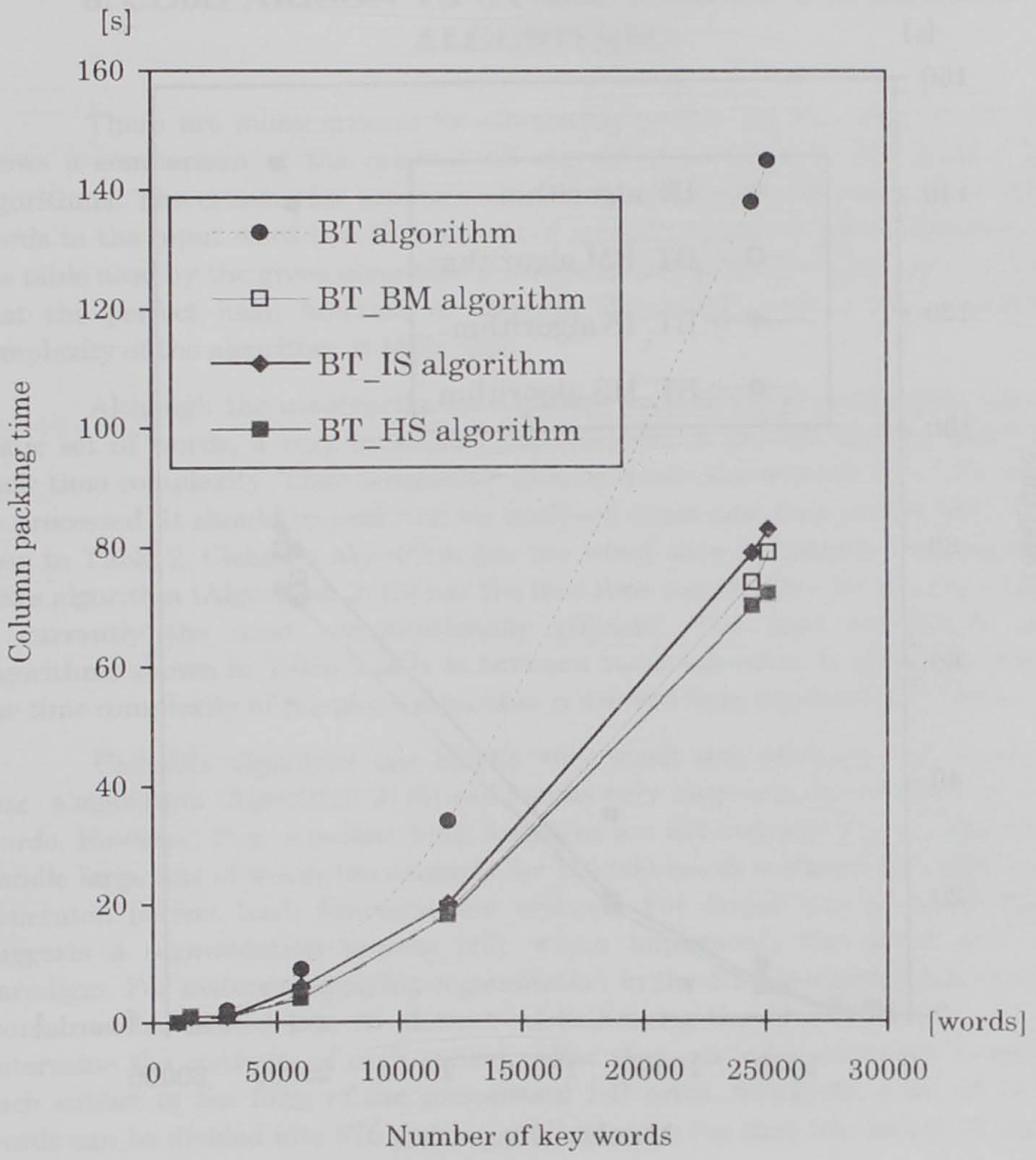


Figure 10: Column packing time as a function of the number of words from the standard UNIX dictionary

Figure 11 shows the total time needed to create the 1-D array as a function of the number of words from the standard UNIX dictionary. The total time is equal to the sum of the time needed to create the 2-D array, the time needed to sort columns of the 2-D array, and the column packing time. Running implementation of the original algorithm takes 147 seconds for the whole UNIX dictionary. Running implementation of modified algorithm BT_HS takes 74 seconds under the same conditions. Consequently, using modified algorithm BT_HS the improvement of the total time is 49.65% of the time needed to construct an ordered perfect hash function by the original BT algorithm. It is important to note that total time for the 2-D array creation and column sorting takes only 2 seconds.

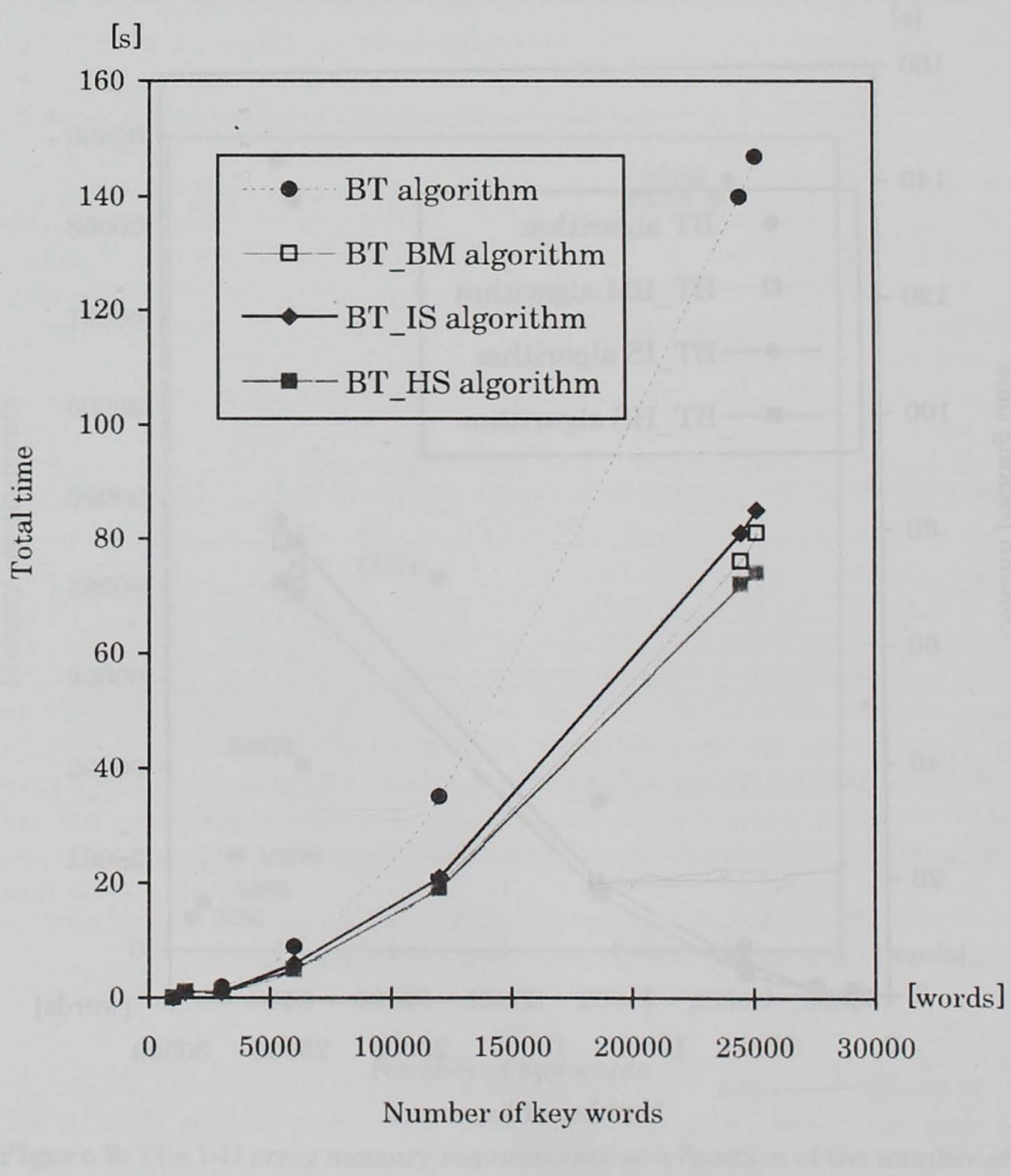


Figure 11: Total time needed to create the 1-D array as a function of the number of words from the standard UNIX dictionary

According to the results shown in Fig. 10 and Fig. 11, the time functions for the BT algorithm have a quadratic form, i.e., $y = kx^2$, where k is a positive constant, and $x > 0$. It can be seen that the time functions for BT_BM, BT_IS, and BT_HS algorithm have a quadratic form as well. The nature of the time functions for the original and modified algorithms is not changed. The difference is only in the values of positive constant k, which graphically shows the improvement achieved in the construction of ordered minimal perfect hash functions. In other words, decreasing the number of needed operations resulted in decreasing value of the positive constant in the quadratic function.

6. COMPARISON TO OTHER PERFECT HASHING ALGORITHMS

There are many criteria for comparing perfect hashing algorithms. Table 2 shows a comparison of the original BT algorithm to other known perfect hashing algorithms. The criteria for comparison are: the function build order, the number of words in the input word list, the amount of memory space (in bytes) required to hold the table used by the given algorithm per item stored in the item list, and confirmation that the perfect hash function is ordered. The build order is the computational complexity of the algorithm at build time.

Although the construction of a perfect hash function occurs once for a given static set of words, a very important characteristic of perfect hashing algorithms is their time complexity. Time complexity directly limits the number of words which can be processed. It should be said that we analysed worst case time complexity. As can be seen in Table 2, Cichelli's algorithm has the worst time complexity - exponential, and Fox's algorithm (Algorithm 2) [8] has the best time complexity - linear. Fox's algorithm is currently the most computationally efficient. The time complexity of other algorithms shown in Table 2 falls in between these extremes. It should be noted that the time complexity of Karplus's algorithm is derived from experimental results.

Cichelli's algorithm can handle very small sets of words (40 words), while Fox s algorithm (Algorithm 2) [8] can handle very large sets on the order of a million words. However, Fox s perfect hash functions are not ordered. The BT algorithm can handle large sets of words (an example for 100,000 words is shown in Table 3) and the generated perfect hash functions are ordered. For larger sets of words Sprugnoli suggests a segmentation process [19], which implements the *divide and conquer* paradigm. For instance, applying segmentation in the BT algorithm a list of 1,000,000 words can be divided into 26 distinct sublists using the first letter of each word to determine the contents of each sublist. After that, an index structure is created for each sublist in the form of the compressed 1-D array. Similarly, a list of 10,000,000 words can be divided into 676 distinct sublists using the first two letters of each word, and so on.

Table 3 shows a comparison of the BT algorithm to the BT_HS algorithm, the best improvement of the BT algorithm. Our experiments are performed in such a way that both algorithms are implemented on the same computer under the same conditions. Improvement of the BT algorithm is achieved in the column packing phase.

Due to the linear build order, Fox s algorithm [8] has the best results for the total time needed to build a perfect hash function for sets of words where ordering is not important. If ordering is important the BT_HS algorithm is better than the others. As Table 3 shows, the total time needed to build a perfect hash function for a given word set by the BT_HS algorithm is also better than the original BT algorithm and on the example of a simulated dictionary with 100,000 words the improvement factor is 1.40. As can be seen in section 5 in the case of a real dictionary the improvement factor is close to 2. Different improvements are due to different distributions of keyword values.

Table 2: Comparison of the BT algorithm to other perfect hashing algorithms

Algorithm Name	Reference	Build Order	List Size	Space (bytes/entry)	Ordered Function
Cichelli	[6]	$O(c^n)$	40	0.65	N
Karplus	[10]	$O(n^{1.5})^*$	667	N.A.	N
Chang	[4]	N.A.	N.A.	N.A.	Y
Sager	[16]	$O(n^6)$	256	4.0	N
Fox	[7]	$O(n^3)$	1,000	4.0	N
Fox Alg2	[8]	$O(n)$	524,288	0.45	N
Brain	[2]	$O(n^2)^*$	1,696	2.4	N
MSMP	[3]	$O(n^2)$	5,000	2.0	N
S & H	[18]	$O(n^3)$	900	1.35	Y
BT	[1]	$O(r^2)$	24,481	3.4	Y

N.A. = Not available.
r is the number of columns of the 2-D array
*Derived from experimental results rather than theoretical analysis

Table 3: Comparison of the BT algorithm to the BT_HS algorithm

Algorithm Name	Reference	Build Order	List Size	Total Time (in s)	Machine	Space (bytes/entry)	Ordered Function
BT	——	$O(r^2)$	25,143	147	Pentium/166	3.39	Y
BT_HS	——	$O(r^2)$	25,143	74	Pentium/166	3.39	Y
BT	——	$O(r^2)$	100,000	916	Pentium/166	5.34	Y
BT_HS	——	$O(r^2)$	100,000	652	Pentium/166	5.34	Y

r is the number of columns of the 2-D array

The amount of memory space required to hold the table used by the given algorithm per item stored in the item list increases from 0.45 bytes for Fox's algorithm (Algorithm 2) [8] to 5.34 bytes for the BT and the BT_HS algorithms.

In Table 2 and Table 3 the very important fact cannot be seen that for the BT and BT_HS algorithms worst case search time is proportional only to the length of the search string. In other words, search times do not directly relate to the number of keywords.

An important open problem in the class of ordered perfect hash algorithms is that the BT and BT_HS algorithms work only with static databases.

7. CONCLUSION

Perfect and minimal perfect hashing algorithms have been studied in many research projects in the last two decades. So far, the BT algorithm is the best perfect hashing algorithm with the possibility to create ordered minimal perfect hash functions. In relation to other competitive algorithms, the BT algorithm is superior in terms of function building efficiency and function complexity at retrieval time. Unlike some previous perfect hashing algorithms, the BT algorithm eliminates all pattern collisions. However, the main disadvantage of the BT algorithm is the time consuming column packing phase. This paper presents three different improvements to the BT algorithm. Namely, the BT_BM algorithm introduces column bit representatives and the bit manipulation paradigm. In the BT_IS algorithm the column-packing phase in the original BT algorithm is modified in such a way that instead of always returning to the first position of the 1-D array, intelligent setting of the initial packing position for each column in the 1-D array is introduced. Hence, the time for finding column packing positions into the 1-D array is decreased. Finally, the BT_HS algorithm combines bit manipulation and intelligent setting of the initial packing position for column bit representatives.

Our experimental results indicate that significant performance improvement may be achieved by the aforementioned three modified versions of the BT algorithm. In order to comparatively evaluate the algorithms detailed experimental analysis was performed for the BT, BT_BM, BT_IS and BT_HS algorithms on the example of the standard UNIX dictionary. Test results have shown that the modified algorithms are better than the original. In the case of the whole standard UNIX dictionary, improvement in total time achieved by the BT_HS algorithm is about 50% of the time needed by the original algorithm to create the 1-D array, that is, the improvement factor is close to 2. The improvement represents a significant contribution to providing possibilities for wider applicability of the described algorithm.

REFERENCES

[1] Brain, M.D., and Tharp, A. L., "Using tries to eliminate pattern collisions in perfect hashing", *IEEE Transactions on Knowledge and Data Engineering*, 6 (2) (1994) 239-247.

[2] Brain, M., and Tharp, A., "Near-perfect hashing of large word sets", *Software Practice and Experience*, 19 (10) (1989) 967-978.

[3] Brain, M., and Tharp, A., "Perfect hashing using sparse matrix packing", *Information Systems*, 15 (3) (1990) 281-290.

[4] Chang, C. C., "The study of an ordered minimal perfect hashing scheme", *Communications of the ACM*, 27 (4) (1984) 384-387.

[5] Chang, C. C., Chen, C. Y., and Jan, J. K., "On the design of a machine-independent perfect hashing scheme", *The Computer Journal*, 34 (5) (1991).

[6] Cichelli, R. J., "Minimal perfect hashing made simple", *Communications of the ACM*, 23 (1) (1980) 17-19.

[7] Fox, E. A., Chen, Q. F., Heath, L. S., and Datta, S., "A more cost effective algorithm for finding minimal perfect hash functions", *ACM Conf. Proc.*, 1989, 114-122.

[8] Fox, E. A., Heath, L. S., Chen, Q. F., and Daoud, A. M., "Practical minimal perfect hash functions for large databases", *Communications of the ACM*, 35 (1) (1992) 105-121.

[9] Jaeschke, G., "Reciprocal hashing: A method for generating minimal perfect hash functions", *Communications of the ACM*, 24 (12) (1981) 829-833.

[10] Karplus, K., and Haggard, G., "Finding minimal perfect hash functions", *Comput. Sci. Dept.*, Cornell Univ., TR84-637, 1984.

[11] Knuth, D. E., *The Art of Computer Programming, Vol. 3, Sorting and Searching, Second Edition*, Addison Wesley, Reading Massachusetts, 1998.

[12] Majewski, B. S., Wormald, N. C., Havas, G., and Czech, Z. J., "A family of perfect hashing methods", *The Computer Journal*, 39 (6) (1996) 547-554.

[13] Park, J. S., Chen, M.-S., and Philip, S. Yu, "Using a hash-based method with transaction trimming for mining association rules", *IEEE Transactions on Knowledge and Data Engineering*, 9 (5) (1997) 813-825.

[14] Ramakrishna, M. V., and Larson, P. A., "File organization using composite perfect hashing", *ACM Transactions on Database Systems*, 14 (2) (1989) 231-263.

[15] Ramakrishna, M. V., Fu, E., and Bahcekapili, E., "A performance study of hashing functions for hardware applications", *Proc. of International Conf. on Computing and Information*, 1994, 1621-1636.

[16] Sager, T. J., "A polynomial time generator for minimal perfect hash functions", *Communications of the ACM*, 28 (5) (1985) 523-532.

[17] Salzberg, B., *File Structures: An Analytic Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[18] Seiden, S. S., and Hirschberg, D. S., "Finding succinct ordered minimal perfect hash functions", Tech. Report, University of California, Irvine, ICS-TR-92-23, Sep. 1994.

[19] Sprugnoli, R. J., "Perfect hashing functions: A single probe retrieval method for static sets", *Communications of the ACM*, 20 (11) (1977) 841-850.

[20] Tarjan, R. E., "Storing a sparse table", Tech. Report, Computer Science Department, School of Humanities and Sciences, Stanford University, STAN-CS-78-683, Dec. 1978, 1-26.

[21] Tarjan, R. E., and Yao, A. C., "Storing a sparse table", *Communications of the ACM*, 22 (11) (1979) 606-611.