BIT 25 (1985), 148–164

A BACKTRACKING METHOD FOR CONSTRUCTING PERFECT HASH FUNCTIONS FROM A SET OF MAPPING FUNCTIONS

W. P. YANG and M. W. DU

Institute of Computer Engineering, National Chiao Tung University, 45 Po Ai Street, Hsinchu, Taiwan, Republic of China

Abstract.

This paper presents a backtracking method for constructing perfect hash functions from a given set of mapping functions. A hash indicator table is employed in the composition. By the nature of backtracking, the method can always find a perfect hash function when such a function does exist according to the composing scheme. Simulation results show that the probability of getting a perfect hash function by the backtracking method is much higher than by the single-pass and multipass methods previously proposed.

1. Introduction.

Hashing is a widely used technique for information storage and retrieval [11, 12, 13, 14, 15]. A one-to-one correspondence hash function from key space to address space is called a perfect hash function. Many approaches have been proposed for constructing perfect hash functions [1, 3, 4, 5, 6, 9, 10, 16, 17]. Recently, Du, Hsieh, Jea and Shieh [6] proposed a perfect hash scheme in which rehashing and segmentation have been employed in constructing hash functions. Rehash means to solve the key collision problem by using a series of hash functions, $h_1, h_2, \ldots, h_s$. Segmentation means to divide address space into segments before allocating them. A hash indicator table (HIT) is used to store the index of hash functions. A hash function h can be defined by HIT as follows [6]:

$$h(k_i) = h_j(k_i) = x_i \text{ if } \mathrm{HIT}[h_r(k_i)] \neq r \text{ for } r < j \text{ and } \mathrm{HIT}[h_j(k_i)] = j,$$
$$= \text{undefined, otherwise.}$$

When h is defined on all keys concerned, it is a perfect hash function.

The advantages of the perfect hash functions defined by HIT here are: they

Received January 1984. Revised April 1984.

are easy to implement and they use only small tables. Two ways for constructing HIT have been proposed in [6, 17]. We shall discuss them briefly in Section 2.2. In Section 3, we introduce another method to construct HIT which is based on a backtracking technique. We present the algorithm of this scheme and discuss its efficiency. In Section 4, we study the probability of getting a perfect hash function by this method in comparison with the other two methods.

2. Construction of a HIT.

2.1 *The Problem*

Consider the matrix of Figure 1. There are three keys, k_1, k_2, and k_3, and two

	k_1	k_2	k_3
h_1	2	2	0
h_2	3	1	1

Fig. 1. A 2×3 mapping table.

hash functions, h_1, and h_2; i.e., we have key space $KS = \{k_1, k_2, k_3\}$ and address space $AS = \{0, 1, 2, 3\}$. The matrix is called a mapping table. The problem here is to construct a perfect hash function h which is composed of h_1 and h_2. For example, one h is defined as follows:

$$h(k_i) = h_j(k_i) = \begin{cases} h_2(k_1) = 3; \\ h_2(k_2) = 1; \\ h_1(k_3) = 0. \end{cases}$$

The perfect hash function h can be expressed as Figure 2 or Figure 3.

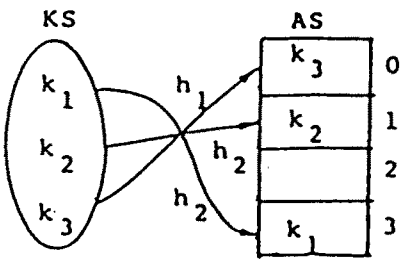

Fig. 2. A perfect hash function composed of h_1 and h_2.

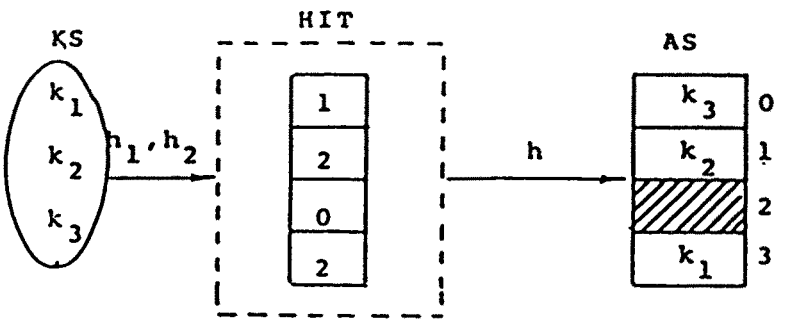

Fig. 3. A perfect hash function h is composed of h_1 and h_2 and HIT is used.

In Figure 3, we use HIT to store the index of the hash functions h_1 and h_2. This table stores the information of whatever function is used to compute the address of the keys in KS. For example, $h_1(k_3) = 0$ and HIT[0] = 1, imply that the key k_3 stored in the AS[0] at address 0 is computed by using h_1. This rule applies to the other entries of HIT. Since only the indices of hash functions are stored, the width of a HIT can be made small. When HIT is stored in the main memory, only one disc access is needed to retrieve a record. The retrieval algorithm is quite simple as proposed in [6, 17]. We list it below:

Procedure RETRIEVAL(k, s, HIT, AS);
// Assume that the hash functions $h_1, h_2,$ //
// ..., h_s, are used to create the HIT. //
// Now we want to retrieve key k. //
begin
  $j := 1$;
  while (HIT[$h_j(k)$] $\neq j$ and $j \leq s$) **do** $j := j+1$;
  if $j > s$ **then** failure
      else k is stored in AS[$h_j(k)$]
end.

In evaluating the function value, the number of loops executed in the statement in the algorithm is called the "number of internal probes" in HIT. Retrieval cost is defined by $n^{-1}\left|\sum_{i=0}^{r-1} \text{HIT}[i]\right|$, where n is the number of keys concerned and r is the size of the address space. The optimal solution is the one among all feasible solutions with the smallest retrieval cost. For a given mapping table, the problem is how to construct a HIT which defines a perfect hash function.

2.2 *Previous work on constructing HIT*

The HIT construction problem was first studied in [6] which proposed a procedure called multi-pass method and then in [17] with another procedure called single-pass method. In the following we explain the operations of these two procedures briefly by applying them to Figure 1 of Section 2.1.

Method 1: Multi-pass procedure

(1) Select all the singletons from the first row, such as 0 in the following table. An entry in row h and column k is a singleton if there is no other k' such that $h(k') = h(k)$.

(2) Select all the singletons from the second row except from those columns which have been selected in the first row, such as 3 and 1 in the following table. We then accomplish and obtain a perfect hash function with HIT = (1, 2, 0, 2).

	k_1	k_2	k_3
h_1	2	2	(0)
h_2	(3)	(1)	1

Method 2: Single-pass procedure

(Basically, we will process the keys in the order k_1, k_2, etc.)

(1) We process k_1 and circle 2.

	k_1	k_2	k_3
h_1	(2)	2	0
h_2	3	1	1

(2) We process k_2. Note that k_2 has the same hash value as k_1 by applying h_1. In this case, k_1 and k_2 collide and both need rehashing by h_2. Thus we get the following result:

	k_1	k_2	k_3
h_1	2	2	0
h_2	(3)	(1)	1

(3) Finally, we process k_3. Since zero is a singleton in the first row, we circle it and obtain a perfect hash function with HIT = (1, 2, 0, 2).

	k_1	k_2	k_3
h_1	2	2	(0)
h_2	(3)	(1)	1

From the discussion above, the main difference between the multi-pass and single-pass approaches is the way to select singletons. While multi-pass

approach would select singletons row by row, single-pass approach would process the keys column by column. The two schemes may have different results. So for a given mapping table it is possible to get a perfect hash function by a multi-pass method but it would be a failure by a single-pass method and vice versa.

EXAMPLE 1. [Failure for multi-pass; success for single-pass method]

Let the key set be KS = $\{k_1, k_2, k_3, k_4\}$. The address space has a size of five entries. The three hash functions are defined by the following table:

	k_1	k_2	k_3	k_4
h_1	2	3	2	3
h_2	3	1	1	0
h_3	4	2	3	1

(1) By applying the multi-pass procedure, we have:

$$\begin{matrix} - & - & - & - \\ 3 & - & - & 0 \\ - & 2 & - & - \end{matrix}$$

That is, the key k_3 cannot be placed in address space, therefore the hash function is not perfect.

(2) By applying the single-pass procedure, we have:

$$\begin{matrix} - & - & - & - \\ - & - & - & 0 \\ 4 & 2 & 3 & - \end{matrix}$$

That is, the hash function h is perfect and defined by HIT = (2, 0, 3, 3, 3).

EXAMPLE 2 [Success for multi-pass; failure for single-pass method]

In the following mapping table:

	k_1	k_2	k_3	k_4
h_1	4	3	3	4
h_2	1	4	2	3

the results of applying the multi-pass procedure and the single-pass procedure

are as follows:

–	–	–	–	–	–	–	–
1	4	2	3	1	–	2	3

(success for multi-pass) (failure for single-pass)

EXAMPLE 3 [Failure for both multi-pass and single-pass methods]

Consider the following mapping table:

	k_1	k_2	k_3	k_3
h_1	3	4	2	4
h_1	4	3	2	1

The HIT constructed by the multi-pass and the single-pass methods are both (0, 2, 1, 1, 0), or

3	–	2	–
–	–	–	1

Therefore it does not define a perfect hash function.

In Example 3, however, we can find two perfect hash functions as defined by HIT = (0, 2, 1, 2, 2) and HIT = (0, 2, 2, 2, 2). I.e.,

–	–	2	–		–	–	–	–
4	3	–	1	and	4	3	2	1

In the following we will introduce a backtracking method to find (1) all the feasible solutions, (2) the optimal solution (i.e., with the lowest retrieval cost), and (3) an answer of "no solution" if no solution exists, according to a given mapping table.

3. Algorithm and efficiency.

3.1 *Basic concepts*

In applying a backtracking technique finding the "intelligent" bounding function is important. Different problems (for example, the 8-queens problem, sum of subsets, graph colouring, Hamiltonian cycle and knapsack problems [2, 7, 8]) have different bounding functions. Given a mapping table, our problem is to construct a HIT to define a perfect hash function. Thus the goals of our bounding function are: (1) Each key in the key space corresponds to a unique address in the address space, and (2) The retrieval algorithm should work correctly. Assume that the solution is $(x_1, x_2, \ldots, x_n)$, where x_i are chosen from

column i in the mapping table. The first of these two goals implies that no two x_i can be the same. That is:

(a) $$x_i \neq x_j \text{ for all } i \neq j,\ 1 \leqq i, j \leqq n.$$

Assume $i \neq j$ and $x_i = h_l(k_j)$, $x_j = h_{l'}(k_j)$, then the second goal is: (see Figure 4)

(b) $$x_i \neq d_j \text{ where } d_j = h_l(k_j), \text{ if } l < l'.$$

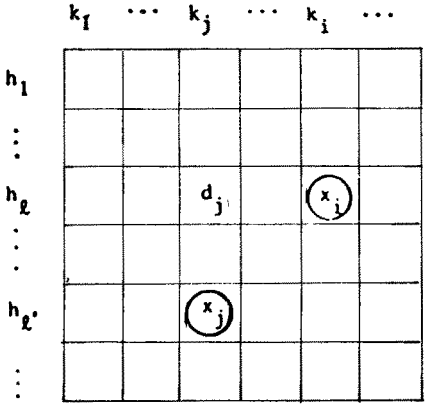

Fig. 4.

Suppose that $(x_1, x_2, \ldots, x_n)$ is a solution which satisfies the constraints (a) and (b), where $x_i = h_{l_i}(k_i)$ for key k_i and some hash function h_{l_i}. Then the HIT corresponding to this solution can be constructed by letting

$$\text{HIT}[h_{l_i}(k_i)] = l_i \text{ for each } i \leqq n.$$

To simplify the explanation, we will consider a problem of constructing a perfect hash function on a 2×3 matrix. Figure 5 shows the configurations as the backtracking proceeds.

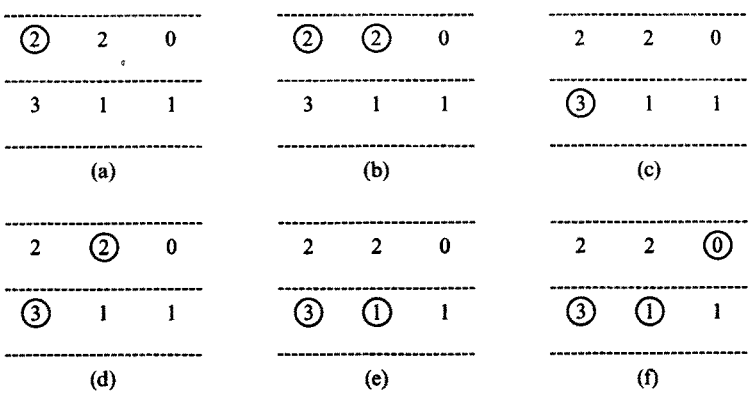

Fig. 5. Example of a backtracking solution to the perfect hashing.

The problem is that exactly one value must be selected in each column. In the beginning a value is selected in the first column of the first row, at position (1, 1), as shown in Figure 5(a), indicated by a circle. Here we have $(x_1, x_2, x_3) = (2,-,-)$. The next value is selected from position (1, 2), $(x_1, x_2, x_3) = (2, 2, -)$, as shown in Figure 5(b). Since no two x_i can be the same, it is necessary to reconsider the first column and select the value at position (2, 1), $(x_1, x_2, x_3) = (3,-,-)$, as shown in Figure 5(c). Note that we need not check the states of (2, 2, 0) and (2, 2, 1). In Figure 5(d), the value 2 at position (1, 2) is the same as the value at position (1, 1). This selection will make future retrieval operations fail, so we discard this value, and select the value 1 at position (2, 2), as shown in Figure 5(e). Finally, the value 0 in column 3 at position (1, 3) is selected, and we get a feasible solution, as shown in Figure 5(f); i.e., we constructed a perfect hash function with HIT = (1, 2, 0, 2). The backtracking algorithm determines problem solutions by systematically searching the solution space for the given problem instance. This search is simplified by using a tree organization for the solution space. In the example the backtracking traverses the nodes of the tree as shown in Figure 6 along the dotted arrows.

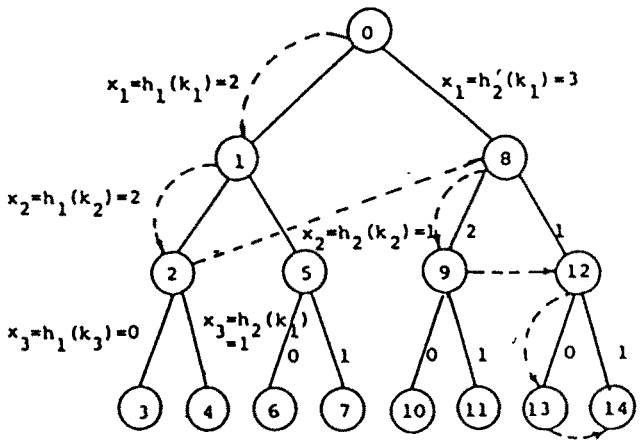

Fig. 6. The depth first order in which backtracking examines the tree of solution space is shown by the dotted arrows.

3.2 *Algorithm*

Here we define some terminological concepts regarding tree organization of solution spaces (see Figure 6). Each node in the tree defines a *problem state.* All paths from the root to other nodes define the state space of the problem. *Solution states* are those problem states S for which the path from the root to S defines a tuple in the solution space. In the tree of Figure 6, only the leaf nodes are solution states. Answer states or *feasible solutions* are those solution states S for which the path from the root to S defines a tuple which is an element of the set of solutions (i.e., it satisfies the constraints) of the problem. The tree

organization of the solution space will be referred to as the state space tree.

Given a mapping table MT, the procedure "FINDHIT" prints all the feasible solutions which define perfect hash functions by using backtracking techniques. The procedure is listed in Figure 7. The variables used are defined as follows:

N	number of the key set.
R	size of the address space.
S	number of rehash functions.
I	index of the value in the solution vector.
MT	mapping table.
X	solution vector.
INDEX	contains row numbers for the corresponding x-values in the mapping table (i.e. the index of hash functions).
NEXT	NEXT(i) points to the next row to be tried in column i.
BACKUP	is the number of columns to go back in the problem states.

```
PROCEDURE FINDHIT;
Var MT: array[1...S, 1...N] of integer;
    X, INDEX, NEXT: array[1...N] of integer;
    HIT: array[1...R] of integer;
    I, K, BACKUP: integer;

FUNCTION T(I): boolean; //generating the next problem states//
  begin  T := false;
    if NEXT[I] ≦ S then
      begin INDEX[I] := NEXT[I]; X[I] := MT[INDEX[I], I];
            NEXT[I] := NEXT[I]+1; T := true end
  end;

FUNCTION B(I): boolean; //bounding function//
  begin B := true;
    for K := 1 to I-1 do
      if X[K] = X[I]
      then begin B := false; //violate constraint(a)//
           if INDEX[K] = INDEX[I]   //with same hash function//
           then if INDEX[K] = 1     //h1 is used in the first row//
                then BACKUP := I-K //backtrack k-i columns//
                else BACKUP := 1   //backtrack to previous column/
           end
      else if (NEXT[K] > NEXT[I]) and (MT[INDEX[I], K] = X[I])
           then B := false  //violate constraint(b)//
  end;
```

```
begin //main program starts here//
  while not end-of-file do
    begin
      read (MT); //read mapping table//
      for I := 1 to N do NEXT[I] := 1, I := 1; //initializing//
      while I > 0 do
        begin
          if (T(I) and B(I)
          then if I = N then print(X and HIT) //obtain a feasible sloution//
                        else I := I+1 // consider the next column//
          else begin //actual backtracking is executed//
             while BACKUP > 0 do
              begin NEXT[I] := 1; I := I-1; BACKUP := BACKUP-1 end;
             while NEXT[I] > S do
              begin NEXT[I] := 1; I := I-1 end;
            end
    end
  end
end.
```

Fig. 7. The procedure FINDHIT prints all the feasible solutions.

Generating the next problem state, T

The boolean function $T(i)$ is used to test whether $(x_1, x_2, \ldots, x_i)$ is in the problem state. If it is true, it implies that x_i is assigned from one value of column i in the MT, and $(x_1, x_2, \ldots, x_{i-1})$ have already been chosen. Now the solution vector is extended to i values, and x_i (to be called the *value of current state* or *current value* in short) is considered as a component of a feasible solution.

Bounding function, B

The bounding function $B(i)$ is false for a path $(x_1, x_2, \ldots, x_i)$ only if the path cannot be extended to reach an answer node, i.e., the bounding function B_i returns a boolean value which is true if the ith x can be selected in the column i of the mapping rable MT, and can satisfy the constraints (a) and (b) in Section 3.1. Thus the candidates for position i of the solution vector $X(1 \ldots n)$ are those values which are generated by $T(i)$ and satisfy $B(i)$. If $i = n$, we obtain a feasible solution and then print it.

EXAMPLE 4 [Backtracking method to construct HIT]

Consider the following mapping table:

	k_1	k_2	k_3	k_4
h_1	1	1	2	1
h_2	3	0	2	4

The size of the solution space is 16. By using the procedure FINDHIT, two feasible solutions are obtained as

$$\begin{matrix} - & - & 2 & - \\ 3 & 0 & - & 4 \end{matrix} \quad \text{and} \quad \begin{matrix} - & - & - & - \\ 3 & 0 & 2 & 4 \end{matrix}$$

with HIT = (2, 0, 1, 2, 2) and HIT = (2, 0, 2, 2, 2) respectively. The first solution is the optimal with retrieval cost equal to 1.75.

EXAMPLE 5 [Backtracking one or more steps]

Consider the following mapping table:

	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}
h_1	4	2	5	6	4	3	8	12	4	12
h_2	6	5	12	3	4	13	3	5	6	5
h_3	5	15	3	6	7	5	6	4	3	11
h_4	4	2	5	11	4	3	17	16	4	14

By using the procedure FINDHIT, the first four solutions can be easily obtained as shown in the Appendix. Before we find the first solution, we should reach a problem state as shown in Figure 8(a). In the figure the partial values (circled) of the solution vector are $X = (4, 2, 5, 6, -, -, -, -, -, -)$, and the current value is 4 (shown by a square box). Since $X[1] = X[5] = 4$, it is impossible to have feasible solutions such that $X[1] = 4$ because the constraints (a) or (b) would be violated. Therefore, it must backtrack the problem state from $I = 5$ to $I = 1$ as shown in Figure 8(b). Figure 8(c) shows the fourth solution. In order to find the next solution, the following problem states are generated and are tested. We show them by Figures 8(d), 8(e), 8(f) and 8(g). In Figure 8(g), again, we have $X[1] = X[6] = 5$. If we backtrack 5 steps (backtrack to the first column), as shown in Figure 8(h), we may lose some feasible solutions. Only a one-step backtracking (backtrack to column 5) is correct as shown in Figure 8(i). Therefore, the solution obtained in Figure 8(j) is our fifth feasible solution. Some other solutions are listed in the Appendix.

④	②	⑤	⑥	[4]	3	8	12	4	12
6	5	12	3	4	13	3	5	6	5
5	15	3	6	7	5	6	4	3	11
4	2	5	11	4	3	17	16	4	14

(a)

4	2	5	6	4	3	8	12	4	12
[6]	5	12	3	4	13	3	5	6	5
5	15	3	6	7	5	6	4	3	11
4	2	5	11	4	3	17	16	4	14

(b)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	(4)	(13)	3	5	6	5
(5)	15	3	6	7	5	6	4	(3)	11
4	2	5	11	4	3	(17)	(16)	4	(14)

(c)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	(4)	(13)	3	5	6	5
(5)	15	3	6	7	5	6	4	3	11
4	2	5	11	4	3	(17)	(16)	[4]	14

(d)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	(4)	(13)	3	5	6	5
(5)	15	3	6	7	5	6	4	3	11
4	2	5	11	4	3	(17)	[16]	4	14

(e)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	(4)	(13)	3	5	6	5
(5)	15	3	6	7	6	6	4	3	11
4	2	5	11	4	3	(17)	16	4	14

(f)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	(4)	13	3	5	6	5
(5)	15	3	6	7	[5]	6	4	3	11
4	2	5	11	4	3	17	16	4	14

(g)

Fig. 8. (Continued).

4	2	5	6	4	3	8	12	4	12
6	5	12	3	4	13	3	5	6	5
5	15	3	6	7	5	6	4	3	11
5	15	3	6	7	5	6	4	3	11
[4]	2	5	11	4	3	17	16	4	14

(h)

4	(2)	5	(6)	4	3	8	12	4	12
6	5	(12)	3	4	13	3	5	6	5
(5)	15	3	6	[7]	5	6	4	3	11
4	2	5	11	4	3	17	16	4	14

(i)

4	(2)	5	(6)	4	(3)	(8)	12	4	12
6	5	(12)	3	4	13	3	5	6	5
(5)	15	3	6	(7)	5	6	4	3	(11)
4	2	5	11	4	3	17	(16)	(4)	14

(j)

Fig. 8. Behavior of the backtracking of Example 4.

3.3 *Efficiency*

How effective is the algorithm FINDHIT over the brute force approach? For a 2×4 matrix as shown in Example 2, there are 31 nodes in the tree organization of the solution space. That is, there are 31 problem states. Some states are ignored by the constraints of the bounding function B. Hence we need to examine only 12 problem states. The efficiency of the algorithm is thus defined by:

$$\text{efficiency} = T_1/T_2,$$

where T_1 denotes the number of nodes generated by the backtracking algorithm, and T_2 denotes the number of nodes in the state space tree. In Example 2, the efficiency is only about 0.387. This implies that only 38.7 percent of the nodes need be examined in order to find the feasible solutions. Since the efficiency is data dependent, we design an experiment to estimate some average values of efficiency. Column (a) in Figure 9 shows the expected values of efficiency to find a feasible solution set. Column (b) shows the expected values of efficiency to find the optimal solution. For a given mapping table, if the perfect hash function does not exist, we say there is no solution. Using the backtracking algorithm, we can determine whether there is no solution by only generating the

partial nodes (the number is denoted by T_1) instead of testing all the nodes in the state space tree (T_2). Column (c) shows the expected values of efficiency to find no solution. All the values are computed from the average of 100 independent test data sets, with loading factor (i.e., (size of KS)/(size of AS)) $\alpha = 0.8$.

($\alpha = 0.8$; $s = 3$)

key no.	(a) a feasible	(b) optimal	(c) no solution
5	0.0744	0.2312	0.1567
10	0.0042	0.0246	0.0093
15	$1.8 * 10^{-4}$	$7.4 * 10^{-4}$	$3.0 * 10^{-4}$
20	$7.4 * 10^{-6}$	$2.6 * 10^{-5}$	$6.7 * 10^{-6}$

Fig. 9. Values of efficiency.

4. Comparison results.

Suppose the mapping tables have random hash values; i.e., the hash values in each mapping table are uniformly distributed over the range. Let P_m, P_s, and P_b denote the probability of getting perfect hash functions by using multi-pass,

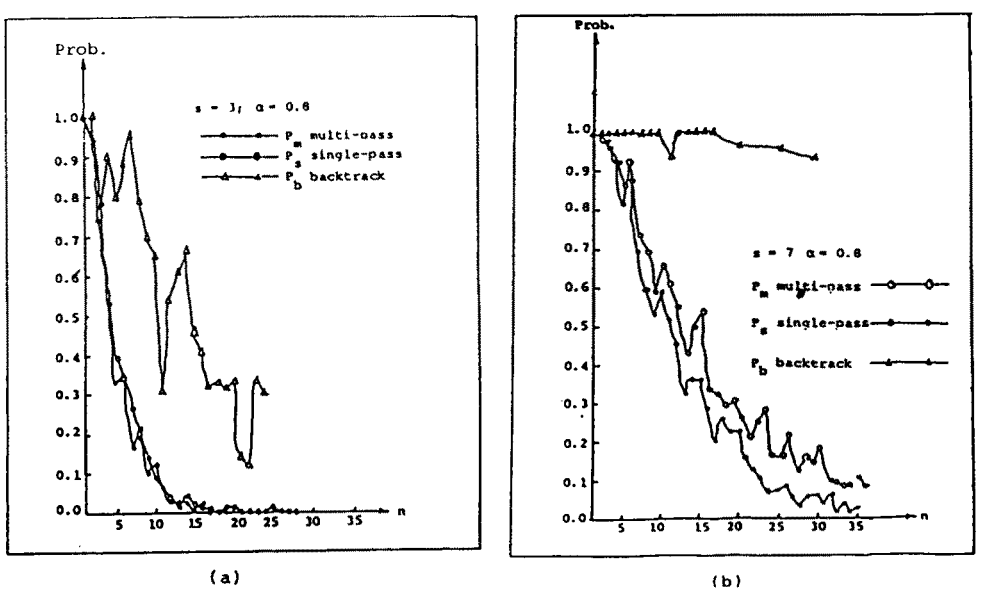

Fig. 10. Probability of constructing perfect hashing: (a) $s = 3$, (b) $s = 7$.

single-pass, and backtracking methods respectively. The procedures for the three methods were programmed in Pascal and their probabilities evaluated on a CDC Cyber 170/720 computer. The results of this evaluation are presented in Figure 10. As the figure indicates, the backtracking method is better than multi-pass and single-pass methods. The retrieval costs of the three methods are also presented in Figure 11. All of them are within a limit, not exceeding 1.95, in the case $s = 3$.

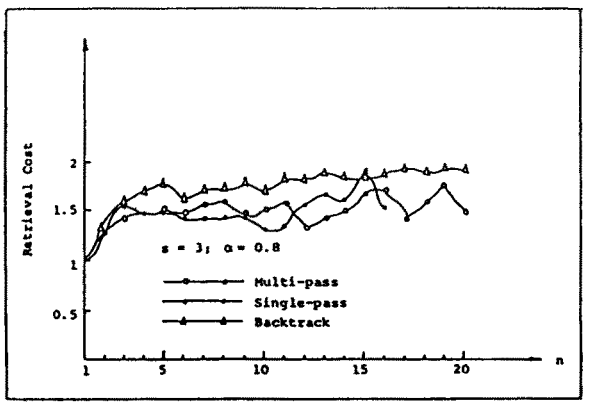

Fig. 11. Retrieval cost.

5. Conclusion.

This paper presents a backtracking method to construct perfect hash functions from a set of mapping functions. Compared with the other two methods proposed before, this new method has a much better chance to get perfect hash functions. For example, as $n = 25$, $\alpha = 0.8$, $s = 7$, the probability of getting a perfect hash function is around 97%. The only problem is that when n increases, the probability of getting a perfect hash function decreases. But this difficulty can be overcome by segmentation, i.e., by dividing the address space into segments. This idea was first used in [6] and was also applied in [17].

If perfect hash functions do exist in a given mapping table, the backtracking method can certainly find all of them and give the optimal solution. It is surprising that, in a mapping table of dimensions 4×10, we get 204 feasible solutions, with efficiency of 0.006. The matrix and some of the feasible solutions are listed in the Appendix.

Appendix.

MAPPING TABLE IS:

4	2	5	6	4	3	8	12	4	12
6	5	12	3	4	13	3	5	6	5
5	15	3	6	7	5	6	4	3	11
4	2	5	11	4	3	17	16	4	14

SIZE OF SOLUTION SPACE: 1048576

FEASIBLE SOLUTION:

(1) – 2 – 6 – – 8 – – –
– – 12 – 4 13 – – – –
5 – – – – – – – 3 11
– – – – – – – 16 – –

HIT = (0, 1, 3, 2, 3, 1, ...) COST = 2.20

(2) – 2 – 6 – – 8 – – –
– – 12 – 4 13 – – – –
5 – – – – – – – 3 –
– – – – – – – 16 – 14

HIT = (0, 1, 3, 2, 3, 1, ...) COST = 2.30

(3) – 2 – 6 – – – – – –
– – 12 – 4 13 – – – –
5 – – – – – – – 3 11
– – – – – – 17 16 – –

HIT = (0, 1, 3, 2, 3, 1, ...) COST = 2.50

(4) – 2 – 6 – – – – – –
– – 12 – 4 13 – – – –
5 – – – – – – – 3 –
– – – – – – 17 16 – 14

HIT = (0, 1, 3, 2, 3, 1, ...) COST = 2.60

(5) – 2 – 6 – 3 8 – – –
– – 12 – – – – – – –
5 – – – 7 – – – – 11
– – – – – – – 16 4 –

HIT = (0, 1, 1, 4, 3, 1, ...) COST = 2.30

(6) – 2 – 6 – 3 8 – – –
– – 12 – – – – – – –
5 – – – 7 – – – – –
– – – – – – – 16 4 14

HIT = (0, 1, 1, 4, 3, 1, ...) COST = 2.40

(7) – 2 – 6 – 3 – – –
– – 12 – – – – – – –
5 – – – 7 – – – – 11
– – – – – – 17 16 4 –

HIT = (0, 1, 1, 4, 3, 1, ...) COST = 2.60

(8) – 2 – 6 – 3 – –
– – 12 – – – – – – –
5 – – – 7 – – – – –
– – – – – – 17 16 4 14

HIT = (0, 1, 1, 4, 3, 1, ...) COST = 2.70

(9) – 2 – 6 – – 8 – – –
– – 12 – – 13 – – – –
5 – – – 7 – – 4 3 11

HIT = (0, 1, 3, 3, 3, 1, ...) COST = 2.20

(10)	–	2	–	6	–	–	8	–	–	–
	–	–	12	–	–	13	–	–	–	–
	5	–	–	–	7	–	–	4	3	–
	–	–	–	–	–	–	–	–	–	14

HIT = (0, 1, 3, 3, 3, 1, ...) COST = 2.30

OPTIMAL SOLUTION: (1)
EFFICIENCY = 0.006 = 8605/1398101

(204)	–	–	–	–	–	–	–	–	–	–
	–	–	12	–	–	13	–	–	–	–
	–	–	–	–	7	–	–	–	3	–
	4	2	–	11	–	–	17	16	–	14

HIT = (0, 4, 3, 4, 0, 0, ...) COST = 3.00

Acknowledgement.

The authors wish to thank the anonymous referee for his valuable and constructive comments.

REFERENCES

1. M. R. Anderson and M. G. Anderson, *Comments on perfect hashing functions: A single probe retrieving method for static sets.* Comm. ACM 22, 2 (Feb. 1979), 104.
2. J. R. Bitner and E. M. Reingold, *Backtrack programming techniques.* Comm. ACM 18, 11 (Nov. 1975), 651–656.
3. R. J. Chichelli, *Minimal perfect hash functions made simple.* Comm. ACM 23, 1 (Jan. 1980), 17–19.
4. C. R. Cook and R. R. Oldehoeft, *A letter oriented minimal perfect hashing function.* ACM Trans. on SIGNPLAN NOTICES 17, 9 (Sept. 1982), 18–27.
5. C. C. Chang, *The study of an ordered minimal perfect hashing scheme.* Comm. ACM 27, 4 (April 1984), 384–387.
6. M. W. Du, T. M. Hsieh, K. F. Jea and D. W. Shieh, *The study of a new perfect hash scheme.* IEEE Trans. on Software Engineering, SE–9, 3 (May 1983), 305–313.
7. S. W. Goloma and L. D. Baumert, *Backtrack programming.* JACM 12, 4 (1965), 516–524.
8. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* Computer Science Press, INC., 1978.
9. G. Jaeschke and G. Osterburg, *On Chichelli's minimal perfect hash functions method.* Comm. ACM 23, 12 (Dec. 1980), 728–729.
10. G. Jaeschke, *Reciprocal hashing: A method for generating minimal perfect hashing functions.* Comm. ACM 24, 12 (Dec. 1981), 829–833.
11. D. E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.
12. W. D. Maurer, *An improved hash code for scatter storage.* Comm. ACM 11, 1 (Jan. 1968), 35–37.
13. W. D. Maurer and T. G. Lewis, *Hash table method.* Computing Surveys 7, 1 (Mar. 1975), 5–19.
14. R. Morris, *Scatter storage techniques.* Comm. ACM 11, 1 (Jan. 1968), 38–44.
15. D. G. Severance, *Identifier search mechanisms: A survey and generalized model.* Computing Surveys 6, (Sep. 1974), 175–194.
16. R. Sprugnoli, *Perfect hashing, functions: A single probe retrieving method for static sets.* Comm. ACM 20, 11 (Nov. 1977), 841–850.
17. W. P. Yang, M. W. Du and J. C. Tsay, *Single-pass perfect hashing for data storage and retrieval.* Proc. 1983 Conf. on Information Sciences and Systems, Baltimore, Maryland, Mar. 1983, 470–476.