Finding Succinct Ordered Minimal Perfect Hashing Functions

Steven S. Seiden* Daniel S. Hirschberg*

February 28, 1992

Technical Report 92-23

Abstract

An ordered minimal perfect hash table is one in which no collisions occur among a predefined set of keys, no space is unused, and the data are placed in the table in order. A new method for creating ordered minimal perfect hashing functions is presented. The method presented is based on a method developed by Fox, Heath, Daoud, and Chen, but it creates hash functions with representation space requirements closer to the theoretical lower bound. The method presented requires approximately 10% less space to represent generated hash functions, and is easier to implement than Fox et al's. However, a higher time complexity makes it practical for small sets only (< 1000).

Keywords: Data Structures, Hashing

1 Introduction

A hash table is a data structure in which a number of keyed items are stored. To access an item with a given key, a hashing function is used. The hashing

*Department of Information and Computer Science, University of California, Irvine, CA 92717.

function maps from the set of keys, to the set of locations of the table. If more than one key maps to a given location, a *collision* occurs, and some collision resolution policy must be followed. On the average, locating an item in a hash table takes $O(1)$ time. Hash tables are used in a wide range of applications. They are quite popular due to their low average access time.

If the set of keys is predetermined, then we may attempt to create a hash table where no collisions occur, i.e., no two keys map to the same location. Such a hash table is called a perfect hash table, and its associated hashing function a perfect hashing function (PHF). Furthermore, we could create a table with the minimal number of locations, exactly one location per key. Such a table is called a minimal perfect hash table, and the associated minimal perfect hashing function (MPHF) is a bijection, from the set of keys onto the set of table locations. If all of the aforementioned conditions are met, and the keys are placed in order, then the function is called an ordered minimal perfect hash function (OMPHF).

OMPHFs are highly useful. Applications where a set of predefined keys need to be recognized abound. For instance, most programming languages have a fixed set of keywords. Compiler and interpreter performance could be improved, if we could decrease keyword recognition time. Another application which could benefit from OMPHFs is database retrieval. Slow access time media, such as CD ROM, demand a high performance method for locating records.

OMPHFs allow the hash table to be stored in the minimal number of locations, n locations are required for n keys. However, different methods require varying amounts of space to represent the hash function. The resources (principally execution time) required to determine an OMPHF will also vary with the method.

We describe a method, due to Fox et al, for determining OMPHFs which is quite efficient in representation space and execution time requirements. We then reformulate this method, so that we may improve on the space representation requirements. The result is a method which is easier to implement and requires less space for applicative use, but entails a greater initial time requirement and approximately the same access time requirement.

2 Fox et al's Method

Fox, Heath, Daoud and Chen [4, 5] have proposed a system for finding ordered minimal perfect hashing functions. Empirical data indicates that their method can find an OMPHF in $O(n)$ expected time. As we shall see, their algorithm is analogous to solving a linear algebraic system over a finite field. This view of the problem provides us with an elegant alternative method of solution.

Fox et al's algorithm (henceforth referred to as algorithm FHDC) generates hashing functions of the form:

$$h(k_i) = [h_0(k_i) + g(h_1(k_i)) + g(h_2(k_i))] \bmod n \tag{1}$$

where:

1. The value k_i is a member of the predefined key set $K = \{k_1, k_2 \dots k_n\}$, and n is the number of keys.
2. The function h_0 is a pseudo-random function from K to the integral range $[0 : n-1]$.
3. Function h_1 is a pseudo-random function from K to $[0 : r-1]$, where r is a user-defined parameter. Typically, the *ratio* (defined below) determines r to be a little more than $n/2$.
4. Function h_2 is a pseudo-random function from K to $[r : 2r-1]$.
5. Function g is a mapping from $[0 : 2r-1]$ to the $[0 : n-1]$.

The goal of algorithm FHDC is to determine the mapping g. Fox et al have shown that the theoretical lower bound on the number of bits to represent an OMPHF is $n \log_2 n$ [4]. In the case of algorithm FHDC, the cardinality of the domain of g governs the amount of space required to represent the OMPHF. The number of values in the domain of g is $s = 2r$, and each value in the range of g is in $[0 : n-1]$. The number of possible mappings is n^s. Therefore, representing g requires at least $\log_2(n^s) = s \log_2 n$ bits. So s must be at least n. Fox et al refer to the value s/n as the *ratio* of an OMPHF. Using functions of the form (1), and ratios of approximately 2.4, Fox et al are able to find OMPHFs with reasonable probability.

The use of pseudo-random functions (originated by Sager [9]) gives algorithm FHDC a distinct advantage over other OMPHF methods [1, 2]. If algorithm FHDC fails, different pseudo-random functions can be tried until hashing succeeds. Several types of pseudo-random functions have been suggested [6, 4, 5, 7]. Fox et al claim that the probability of failure is low and, therefore, that the probability of not finding an OMPHF after the first few attempts is negligible. (They assume that the pseudo-random values generated are independent of the keys.)

Fox et al also present a more space efficient method, which uses functions of the form:

$$h(k_i) = \begin{cases} \alpha(k_i) & \text{if } b(h_1(k_i)) = b(h_2(k_i)) \\ \beta(k_i) & \text{otherwise} \end{cases} \tag{2}$$

where:

$$\begin{aligned} \alpha(k_i) &= g(h_0(k_i) + g(h_1(k_i)) + g(h_2(k_i))) \bmod 2r \\ \beta(k_i) &= h_0(k_i) + g(h_1(k_i)) + g(h_2(k_i)) \bmod n \end{aligned}$$

b is a Boolean function on the domain $[0 : 2r - 1]$, and g is a mapping as previously described. The functions b and g are determined by the algorithm. Using functions of form (2), the lowest ratio which Fox et al achieve is 1.13. Our method allows ratios much closer to one.

3 A Reformulation

Gori and Soda [8] were able to reformulate the minimal perfect hashing technique of Cichelli [3] in terms of linear algebra. They showed how Cichelli's algorithm is analogous to solving a set of simultaneous linear equations over the field $\mathbf{R}$ (the real numbers). Similarly, an OMPHF of the form (1) can be found by solving of a set of simultaneous equations, if n is prime. These simultaneous equations are not over $\mathbf{R}$, but over the finite field $\mathbf{Z}_n$ (the field of integers modulo n). The fact that n is prime is fundamental, because $\mathbf{Z}_n$ is a field if and only if n is prime. We define the following:

$$\begin{aligned} a_i &= h_0(k_i) \\ b_{i,j} &= \begin{cases} 1 & \text{if } (h_1(k_i) = j) \vee (h_2(k_i) = j) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Suppose n is prime and $2r \geq n$. Consider the following set of equations:

$$\begin{array}{rcl}
0 & = & g_0 b_{1,1} + g_1 b_{1,2} + \cdots + g_{2r-1} b_{1,2r-1} + a_1 \\
1 & = & g_0 b_{2,1} + g_1 b_{2,2} + \cdots + g_{2r-1} b_{2,2r-1} + a_2 \\
2 & = & g_0 b_{3,1} + g_1 b_{3,2} + \cdots + g_{2r-1} b_{3,2r-1} + a_3 \\
\vdots & \vdots & \vdots \\
n-1 & = & g_0 b_{n,1} + g_1 b_{n,2} + \cdots + g_{2r-1} b_{n,2r-1} + a_n
\end{array}$$

This system can be rewritten as:

$$\hat{H} = \hat{B}\hat{G}$$

where $\hat{H}$ is a column vector defined by $h_i = i-1-a_i$, $\hat{B}$ is a $n \times 2r$ matrix with entries $b_{i,j}$ as previously defined, and $\hat{G}$ is the vector to be solved for. The variables $g_0, g_1 \ldots g_{2r-1}$ correspond exactly to the values $g(0), g(1) \ldots g(2r-1)$ determined by algorithm FHDC. This system is solvable over $\mathbf{Z}_n$, if $\hat{B}$ has rank n. That is, if the rows of $\hat{B}$ generate $\mathbf{Z}_n^n$ (the n dimensional vector space over $\mathbf{Z}_n$). If this is the case, the keys may be hashed in any a priori order. Row reduction, for example, is one of the methods for finding a solution. We can solve this system in worst case time of $O(n^3)$.

The average time complexity of the method we propose is greater than that of algorithm FHDC. However, our method may be easier to implement. For small sets of keys ($n < 1000$), implementation cost may be more important than running time. It is likely that mathematical software libraries exist which allow for the rapid implementation of our method.

4 A New Method

The reformulation we described in the previous section can be improved in several ways. The effect of h_0 is to create random entries for $\hat{H}$. This function plays no role in whether $\hat{B}$ has rank n, therefore, h_0 would be better utilized if it contributed to the rank of $\hat{B}$. This is facilitated by replacing $h_0(k_i)$ with $g(h_0(k_i))$, thereby increasing the average number of non-zero entries in each row of $\hat{B}$. The average can also be increased by using a greater number, m, of pseudo-random functions. Secondly, empirical data indicates that it makes little difference that h_1 and h_2 map to disjoint ranges. We let all

i	k_i	$h_0(k_i)$	$h_1(k_i)$	$h_2(k_i)$	$h_3(k_i)$	$h(k_i) = \Sigma g(h_j(k_i)) \bmod p$
1	'january'	2	4	4	10	$0 = 3 + 2 + 2 + 6 \bmod 13$
2	'february'	3	4	5	9	$1 = 4 + 2 + 4 + 4 \bmod 13$
3	'march'	0	2	2	7	$2 = 1 + 3 + 3 + 8 \bmod 13$
4	'april'	2	3	6	9	$3 = 3 + 4 + 5 + 4 \bmod 13$
5	'may'	1	1	2	6	$4 = 11 + 11 + 3 + 5 \bmod 13$
6	'june'	3	3	4	7	$5 = 4 + 4 + 2 + 8 \bmod 13$
7	'july'	2	5	7	9	$6 = 3 + 4 + 8 + 4 \bmod 13$
8	'august'	2	9	11	11	$7 = 3 + 4 + 0 + 0 \bmod 13$
9	'september'	1	2	7	8	$8 = 11 + 3 + 8 + 12 \bmod 13$
10	'october'	0	0	0	10	$9 = 1 + 1 + 1 + 6 \bmod 13$
11	'november'	0	1	6	10	$10 = 1 + 11 + 5 + 6 \bmod 13$
12	'december'	7	9	10	10	$11 = 8 + 4 + 6 + 6 \bmod 13$

Figure 1: Values of $h_0 \ldots h_3$ and h, for the set of months (See Figure 4 for values of g).

pseudo-random functions map to $[0 : s-1]$. Thirdly, if n is not prime, we pick a prime $p > n$ and use p as the modulo. In summary, the functions we propose have the form:

$$h(k_i) = \left(\sum_{j=0}^{m-1} g(h_j(k_i)) \right) \bmod p \tag{3}$$

where each h_j is a pseudo-random function mapping from K to $[0 : s-1]$, g is a mapping from $[0 : s-1]$ to $[0 : p-1]$, and p is the least prime $p \geq n$. We redefine $\hat{B}$. The value of an entry, $b_{i,j}$, is the number of the values $h_0(k_i)$, $h_1(k_i) \ldots h_{m-1}(k_i)$ which are equal to j. We define $\hat{H}$ by $h_i = i - 1$ (So that the values $[n : p-1]$ are excluded from the image of h). The resulting linear system is solved as before. A small example is presented in Figures 1–4. An OMPHF is created for the months of the year, using four pseudo-random functions ($m = 4$). Since $n = 12$ is not prime, we let $p = 13$. The values of $h_0 \ldots h_3$ are displayed in Figure 1. The matrix $\hat{B}$ augmented by $\hat{H}$ is displayed in Figure 2, the row reduced matrix in Figure 3. The values of g appear in Figure 4. In this case, we are able to find an OMPHF with a ratio of 1.

We implemented our method in C, and used Monte Carlo methods to

$$\begin{pmatrix}
0 & 0 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 3 \\
0 & 2 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 6 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 2 & 7 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 8 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 9 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 10 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 2 & 0 & 11
\end{pmatrix}$$

Figure 2: The matrix $(\hat{B}|\hat{H})$ for the set of month names (over $\mathbf{Z}_{13}$).

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 11 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}$$

Figure 3: Row reduced matrix for the months.

i	0	1	2	3	4	5	6	7	8	9	10	11
$g(i)$	1	11	3	4	2	4	5	8	12	4	6	0

Figure 4: Values of g.

a)

n	m, s/n			
	2, 2.40	3, 1.10	4, 1.03	5, 1.01
53	.83	.67	.84	.93
101	.73	.80	.84	.82
151	.72	.74	.82	.86
199	.78	.94	.78	.81
251	.77	.81	.87	.85
307	.77	.68	.79	.77
349	.79	.75	.81	.74
401	.80	.78	.78	.85
449	.78	.85	.80	.76
503	.75	.84	.79	.77
	.77	.79	.81	.82

b)

$m = 3$, $s/n = 1.10$		
n	p	Solved
50	53	.60
100	101	.84
150	151	.73
200	227	.94
250	251	.85
300	307	.82
350	353	.70
400	401	.87
450	457	.86
500	503	.88
		.81

Figure 5: a) Rates of solvability for various m and n, with n prime, and b) for various non-prime n with p the least prime greater than n.

test it. Instead of generating random keys, and then pseudo-random values from them, we simulated keys by directly generating random values of $h_0(k_i)$, $h_1(k_i) \ldots h_{m-1}(k_i)$. Generating random values from random values seems to be unnecessary. We find that, indeed, larger values of m result in a higher incidence of solvability. By increasing m, it is possible to find OMPHFs with ratios approaching 1. For $m = 2$, our results confirm those of Fox et al—with a ratio of 2.40, about 77% of the cases are solved. For $m = 3$, a much lower ratio may be used. We note that, with a ratio of 1.10, an OMPHF is found about 79% of the time. Further, for $m = 4$, a ratio of 1.03 suffices to find an OMPHF in 81% of the cases and, for $m = 5$, a ratio of 1.01 is sufficient in 82% of the cases. Our data indicates that the ratio required to achieve a given success rate is invariant. We find that performance does not degrade when $p > n$. The results of our Monte Carlo study appear in Figure 5. For each n and m, 100 cases were tried.

We were also able to create OMPHFs for several real key sets, including a set of 557 words drawn from the Unix online dictionary. For these OMPHFs, we use pseudo-random functions as described in [7].

5 Conclusions

The method we present has a higher time complexity than algorithm FHDC. However, it has several advantages:

1. It is conceptually elegant.
2. For small sets, implementation costs may be more significant than time complexity. Our method may be implemented using pre-existing mathematical library routines.
3. By using higher values of m, OMPHFs may be found with ratios approaching 1, the theoretical lower bound. Our method uses approximately 10% less space to store generated functions than algorithm FHDC, with $m = 4$.

References

[1] Chang, C. C. On the design of letter oriented minimal perfect hashing functions. *Journal of the Chinese Institute of Engineers 8*, 3 (1985), 285–297.

[2] Chang, C. C., and Lee, R. T. C. A letter oriented minimal perfect hashing scheme. *The Computer Journal 29*, 3 (1986), 277–281.

[3] Cichelli, R. J. Minimal perfect hash functions made simple. *Communications of the ACM 23*, 1 (Jan 1980), 17–19.

[4] Fox, E. A., Chen, Q., Daoud, A. M., and Heath, L. S. Order preserving minimal perfect hash functions and information retrieval. In *Proceeding of the 13th Annual ACM Conference on Research and Development of Information Retrieval* (1989), pp. 279–311.

[5] Fox, E. A., Chen, Q., Daoud, A. M., and Heath, L. S. Order preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems 9*, 3 (Jul 1991), 281–308.

[6] Fox, E. A., Chen, Q., Heath, L. S., and Datta, S. A more cost effective algorithm for finding minimal perfect hashing functions. In

Computing Trends in the 90's: 17th ACM Computer Science Conference (1989), pp. 114–122.

[7] FOX, E. A., HEATH, L. S., DAOUD, A. M., AND CHEN, Q. Practical minimal perfect hash functions for large databases. *Communications of the ACM 35*, 1 (Jan 1992), 105–121.

[8] GORI, M., AND SODA, G. An algebraic approach to Cichelli's perfect hashing. *BIT 29*, 1 (1989), 2–13.

[9] SAGER, T. J. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM 28*, 5 (May 1985), 523–532.