Finding Succinct Ordered Minimal Perfect Hash Functions

Steven S. Seiden* Daniel S. Hirschberg*

September 22, 1994

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 hash functions is presented. It creates hash functions with representation space requirements closer to the theoretical lower bound than previous methods. The method presented requires approximately 17% less space to represent generated hash functions and is easy to implement. However, a high time complexity makes it practical for small sets only (size < 1000).

Keywords: Data Structures, Hashing, Perfect 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 hash function is used. The hash 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 [11]. 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 hash function a perfect hash 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 hash function (MPHF) is a bijection, from the set of keys onto the set of table locations. If

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

all of the aforementioned conditions are met, and the keys are placed in some predetermined 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. OMPHFs allow predefined keys to be recognized quickly. In many applications, it is also necessary to be able to list the keys in a particular order (i.e. alphabetic order) and to find the predecessor and successor of a given key. OMPHFs allow listing, since the table merely needs to be printed in order. Predecessor and successor operations are easily implemented since the predecessor of a key is located immediately before it and the successor immediately after it.

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.

2 Preliminaries

The functions we propose have the form:

$$h(x) = \left[\sum_{j=0}^{m-1} g\Big(h_j(x)\Big) \right] \bmod p \tag{1}$$

where:

1. p is the least prime $p \geq n$.
2. $m \geq 1$ is some small integer.
3. k_i is a member of the predefined key set $K = \{k_1, k_2, \ \ldots \ , k_n\}$, and n is the number of keys. Note that the indices imply the ordering of the keys. For example, if the keys are character strings, then k_1 is the first key in lexicographic order.
4. h_i is a pseudo-random function from K to the integral range $[0 : s-1]$, where s is a user-defined parameter. Typically, the *ratio* (defined below) determines s to be a little more than n.
5. g is a mapping from $[0 : s-1]$ to $[0 : p-1]$.

Majewski, Wormald, Czech and Havas [9], and Czech, Havas and Majewski [3] utilize functions of this form, however, they impose additional requirements on images of the h_i's. Fox, Chen, Daoud, and Heath use functions of a very similar form [5] (Section 2.1.1, Method 1). (The differences are not important, the main one being that they fix $m = 2$.)

The composition of the keys (alphabetic, numeric etc...) is of concern to us only insofar that good pseudo-random functions can be found which map the keys to integers. Several

authors have given families of pseudo-random functions which work well for character string keys [4, 5, 6, 7, 10].

The goal of our algorithm, and that of the algorithms of Fox et al., Czech et al., and Majewski et al., is to determine the mapping g.

The cardinality of the domain of g governs the amount of space required to represent the OMPHF. Fox et al. have shown that the theoretical lower bound on the number of bits to represent an OMPHF approaches $n \log_2 n$ as n gets large [4]. The number of values in the domain of g is s, 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. After Fox et al., we refer to the value s/n as the *ratio* of the OMPHF.

The use of pseudo-random functions gives our algorithm and similar algorithms [3, 6, 9] a distinct advantage over other OMPHF methods [1, 2]. If our algorithm fails, different pseudo-random functions can be tried until hashing succeeds.

3 The Algorithm

We now present our algorithm for finding a function g, if it exists. We define:

$$b_{i,j} = \Big|\{h_\ell \mid h_\ell(k_i) = j\}\Big|$$

In words, $b_{i,j}$ is the number of values $h_0(k_i), h_1(k_i), \ldots, h_{m-1}(k_i)$ which are equal to j. The image of g contains s values, $[0 : s-1]$. For each of these, we must determine which value in $[0 : p-1]$ it maps to. We create a variable g_i which corresponds directly to the value of $g(i)$. Consider the following set of equations:

$$\begin{aligned}
b_{1,0}g_0 + b_{1,1}g_1 + \cdots + b_{1,s-1}g_{s-1} &= 0 \\
b_{2,0}g_0 + b_{2,1}g_1 + \cdots + b_{2,s-1}g_{s-1} &= 1 \\
b_{3,0}g_0 + b_{3,1}g_1 + \cdots + b_{3,s-1}g_{s-1} &= 2 \\
&\vdots \quad \vdots \quad \vdots \\
b_{n,0}g_0 + b_{n,1}g_1 + \cdots + b_{n,s-1}g_{s-1} &= n-1
\end{aligned}$$

over the finite field $\mathbb{Z}_p$. (Where all arithmetic operations are performed modulo p. Note that $\mathbb{Z}_n$ is not a field if n is not prime, and so we choose p, the least prime greater than or equal to n, to be our modulus. A field is needed for Gaussian elimination.) This system can be rewritten as:

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

where $\hat{H}$ is a column vector defined by $h_i = i - 1$, $\hat{B}$ is the $n \times s$ matrix with entries $b_{i,j}$ as previously defined, and $\hat{G}$ is the vector to be solved for. This system of equations will have a solution if and only if a g yielding an OMPHF exists. The system is solvable over $\mathbb{Z}_p$, if $\hat{B}$ has rank n. If this is the case, the keys may be hashed in any a priori order. Gaussian elimination, which can be performed in worst case time of $O(n^3)$, is one method for finding

a solution. Since the system is sparse, we might also use a probabilistic method, due to Wiedemann [12], which will solve such systems in expected $O(n^2)$ time. We note that Gori and Soda originated the idea of using algebraic methods to find OMPHFs [8].

We discuss briefly the importance of pseudo-random functions, since their use is central to our method. The methods of Fox et al., Czech et al., and Majewski et al. also depend on pseudo-random functions. We limit our discussion to pseudo-random functions of character strings. We assume that our pseudo-random functions have the following properties:

1. The values on differing keys, $h_i(x)$ and $h_i(y)$, are independent.
2. The values of differing functions on the same key, $h_i(x)$ and $h_j(x)$, are independent.
3. The functions are uniformly distributed. For any character string x, the probability that $h_i(x) = j$ should equal the probability that $h_i(x) = j'$, for all $j \neq j'$ in the range of h_i.

Property 1 is clearly important, if two keys are not hashed independently, the left hand sides of the corresponding equations will be identical, and no solution to the linear system can be found. Properties 2 and 3 are of a more heuristic nature. They ensure that the resulting matrix is truly random. Fox et al. [7] have proposed using functions of the following form:

$$h(x) = \left[\sum_{i=1}^{|x|} H(x_i, i) \right] \bmod s \tag{2}$$

where $H(c, i)$ is a mapping from each character and index to a random integer in $[0 : s-1]$. This family of functions has the properties we desire. Further, it has the property that functions in the family are easily created. To create a pseudo-random function, we merely generate a small table of random numbers. Note that because of this, and property 2, if one set of pseudo-random functions $\{h_0, \ldots, h_{m-1}\}$ fails to provide an OMPHF, we can easily generate a new set and try again. If a given random set of equations (of the form we use) is solvable with probability independent of its size, we have a Las Vegas algorithm for finding OMPHFs. Note that for smaller key sets, functions of a simpler nature than (2) may suffice.

The average time complexity of the method we propose is much greater than that of the methods of Fox et al., Majewski et al., and Czech et al., all of which have linear time complexity. However, for small sets of keys ($n < 1000$), implementation cost may be more important than running time. Mathematical software libraries exist which allow for the rapid implementation of our method. For instance, the symbolic algebra package *Mathematica* can be used to solve simultaneous equations over finite fields.

A small example is presented in Tables 1 and 2 and Figure 1. An OMPHF is created for the months of the year, using three pseudo-random functions ($m = 3$). Since $n = 12$ is not prime, we let $p = 13$. We use a ratio of 1.10, and thus s is also 13.

For our example, the value of the pseudo-random function h_i on a character string is the sum of the values of h_i for each character. A random number is assigned to each character 'a' – 'z' by each h_i. Note that such a simple pseudo-random function works for this example,

	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'
h_0	0	6	10	6	12	2	9	5	1	8	0	7	7
h_1	12	9	7	2	0	3	7	3	1	3	12	2	12
h_2	4	7	8	10	0	11	8	8	5	4	3	9	3
	'n'	'o'	'p'	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'	'y'	'z'
h_0	4	12	6	7	11	12	11	6	6	9	2	6	10
h_1	12	12	12	3	1	6	7	9	7	0	6	1	8
h_2	12	4	4	12	1	7	2	10	10	8	5	7	11

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

Table 1: Values of h_0, h_1, h_2 and g

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

Table 2: Values of h_0, h_1, h_2 and h, for the set of months

but might not work for larger examples. For instance, if the key set contained anagrams, then a more complicated pseudo-random scheme would be required. The values of h_0, h_1, h_2 are displayed in Table 2. Also note that if this particular set of pseudo-random functions were not to work, we could easily produce new pseudo-random functions, by simply assigning different random values in Table 2.

The matrix $\hat{B}$ augmented by $\hat{H}$ is displayed in Figure 1. By using Gaussian elimination on this matrix, we solve for g. Note that if g was not found, we could simply choose new numbers for Table 2, and try again. Since $p \neq n$, the ratio is $13\lceil \log_2 13 \rceil / (12 \lceil log_2 12 \rceil) = 1.083$.

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

Figure 1: The matrix $(\hat{B}|\hat{H})$ for the set of month names (over $\mathbb{Z}_{13}$)

4 Analysis

Theoretical analysis of solvability of random equations (of our particular form) would seem to be difficult, and so we test our method empirically.

We implemented our method in C, and used Monte Carlo methods to test it. We tested our method on sets of keys chosen randomly from the Unix online dictionary, `/usr/dict/words`. Keys with length less than 3 or greater than 16 were not considered as candidates. For these OMPHFs, we use pseudo-random functions as described in [7].

The results of our Monte Carlo study appear in Table 3. The parameters n, m and the ratio value were varied. For $n \leq 500$, 100 cases were tried, while for $n > 500$, 20 cases were tried. We find that larger values of m result in a higher incidence of solvability. By increasing m, it is possible to find OMPHFs with ratios approaching 1. However, it appears that the ratio must be selected carefully. If the ratio is too low, then the rate of solvability decreases with increasing n. On the other hand, if the ratio is too large, then the rate of solvability will go to 1 as n increases, but space will be wasted. For fixed n, the probability of finding a hashing increases with s/n. In order to demonstrate this, for each choice of $m = 3, 4, 5$, we try three different ratios: one which results in a solvability rate approaching 0 for large n, one which results in a solvability rate of $\approx .80$, and one which results in a solvability rate approaching 1 for large n, The case of $m = 2$ has been studied by Fox et al., and is not of primary concern to us, since hash functions with $m = 2$ are space wasteful (they require more than 10% above the theoretical lower bound). For $m = 2$, with a ratio of 2.40, about 78% of the cases are solved. This confirms the results of Fox et al. [4]. For $m = 3$, with a ratio of 1.10, an OMPHF is found about 73% of the time. Further, for $m = 4$, a ratio of 1.03 suffices to find an OMPHF in 84% of the cases and, for $m = 5$, a ratio of 1.01 is sufficient in 86% of the cases.

Our method does not provide a success rate of 100%. However, using the family of

pseudo-random functions described by Fox et al. [7], we are able to quickly generate new pseudo-random functions if hashing fails. From the probabilities determined by our empirical data we calculate, for $m = 2$ with ratio 2.40, the expected number of tries to find a hash function for a given set of keys is ≈ 1.3. For $m = 3$ with ratio 1.10 the expected number of tries is ≈ 1.4. For $m = 4$ and $m = 5$ with ratios 1.03 and 1.01 the expected number of tries to find a hashing is ≈ 1.2.

We compare the space requirements of our method with that of others. For $m = 2$, our results confirm those of Fox et al., a ratio of 2.40 is sufficient [4] (analysis of method 1, Section 3.2.2.). Fox et al. also give a method for finding OMPHFs which uses functions quite different from (1) [4] (Method 3, Section 2.1.3). This method achieves ratios around 1.20. (In one case they find a ratio of 1.13.) Majewski et al. found that their method achieved a minimal ratio of 1.23 with $m = 3$. The method of Majewski et al. requires ratios of 1.29 and 1.41 for $m = 4$ and $m = 5$, respectively. Our method requires less space than any other for $m > 2$.

The tradeoff is that our method requires much more time to find a hashing. The other cited methods all achieve an expected linear time complexity. They can be used for very large key sets, Our method can only be used for key sets of small size. For several case sizes, $m = 3$, and a ratio of 1.10, the average times to find hash functions (using an $O(n^3)$ Gaussian elimination algorithm) are displayed in Table 4. Timings were made on a SPARC station IPC. In comparison, the method of Fox et al. needs only 3 seconds to find an OMPHF for 16,384 keys, on a DECsystem 5000 model 200 [4].

5 Conclusion

The method we present has a higher time complexity than the algorithms of Fox et al., Majewski et al., and Czech et al.. However, it has several advantages:

1. It is conceptually simple.

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.

An open problem is calculating the theoretical probability of finding an OMPHF, as a function of (primarily) m and the ratio. Czech et al. and Majewski et al. have performed such analysis for the case of $m = 2$ and ratios greater than two [3, 9]. However, for $m > 2$ analysis seems much more difficult. This work and that of Czech et al. and Majewski et al. rely on empirical analysis for $m > 2$.

n	p	m									
		2	3			4			5		
		s/n									
		2.40	1.05	1.10	1.20	1.01	1.03	1.05	1.005	1.01	1.03
50	53	0.81	0.36	0.61	0.94	0.66	0.85	0.99	0.75	0.94	0.99
100	101	0.74	0.15	0.67	0.95	0.34	0.78	1.00	0.79	0.79	0.99
150	151	0.73	0.15	0.74	0.98	0.38	0.86	1.00	0.72	0.96	1.00
200	211	0.74	0.06	0.75	0.97	0.16	0.73	1.00	0.51	0.80	1.00
250	251	0.77	0.06	0.70	0.99	0.17	0.86	1.00	0.36	0.86	1.00
300	307	0.72	0.03	0.71	0.98	0.10	0.84	1.00	0.62	0.75	1.00
350	353	0.78	0.00	0.71	1.00	0.10	0.85	1.00	0.46	0.84	1.00
400	401	0.79	0.02	0.69	0.98	0.06	0.82	1.00	0.34	0.74	1.00
450	457	0.79	0.01	0.71	1.00	0.06	0.85	1.00	0.34	0.83	1.00
500	503	0.77	0.00	0.71	1.00	0.04	0.81	1.00	0.41	0.88	0.98
600	601	0.70	0.00	0.65	1.00	0.00	0.95	1.00	0.40	0.80	1.00
700	701	0.85	0.00	0.85	1.00	0.00	0.65	1.00	0.40	0.75	1.00
800	809	0.90	0.00	0.80	0.95	0.00	1.00	1.00	0.40	0.90	1.00
900	907	0.80	0.00	0.85	1.00	0.00	0.85	1.00	0.50	1.00	1.00
average		.78	.06	.73	.98	.15	.84	1.00	.50	.86	1.00

Table 3: Rates of solvability for various n, m and s/n.

n	50	100	150	200	250	300	350
time	0.05	0.25	0.72	1.69	2.96	5.12	8.52
n	400	450	500	600	700	800	900
time	11.8	17.1	24.1	38.0	72.7	108	164

Table 4: Average times for various n (in seconds).

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] Czech, Z. J., Havas, G., and Majewski, B. S. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters 43*, 5 (Oct 1992), 257–264.

[4] Fox, E. A., Chen, Q., Daoud, A. M., and Heath, L. S. Order preserving minimal perfect hash functions and information retrieval. In *Proceedings 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] Majewski, B. S., Wormald, N. C., Czech, Z. J., and Havas, G. A family of generators of minimal perfect hashing functions. Tech. Rep. 92-16, DIMACS, 1992.

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

[11] Standish, T. A. *Data Structure Techniques.* Addison-Wesley, 1980, pp. 145–149.

[12] Wiedemann, D. H. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory IT-32*, 1 (Jan 1986), 54–62.