A Faster Algorithm for Constructing Minimal Perfect Hash Functions *

Edward A. Fox*# and Qi Fan Chen*# and Lenwood S. Heath*

Department of Computer Science* and Computing Center#
Virginia Polytechnic Institute and State University
Blacksburg VA 24061-0106

Abstract

Our previous research on one-probe access to large collections of data indexed by alphanumeric keys has produced the first practical minimal perfect hash functions for this problem. Here, a new algorithm is described for quickly finding minimal perfect hash functions whose specification space is very close to the theoretical lower bound, i.e., around 2 bits per key. The various stages of processing are detailed, along with analytical and empirical results, including timing for a set of over 3.8 million keys that was processed on a NeXTstation in about 6 hours.

1 Introduction

Next generation information systems must support integrated access to large-scale data, information, and knowledge bases. That integration must facilitate efficient operation, as well as ease-of-understanding, for both users and developers. Information retrieval and filtering, hypertext, hypermedia, natural language processing, scientific data management, transaction processing, expert systems, library catalog access, and other applications can all be built upon such an integrated environment.

We have worked toward this goal of integrated access from two directions. First, the CODER (COmposite Document Expert/extended/effective Retrieval) system serves as a prototyping vehicle for our theories, models, approaches, and implementation efforts [6]. Its architecture allows blackboard as well as client-server style communication in one or more communities of experts or algorithmic modules. A knowledge representation language has been developed for CODER to give us control over inter-module communication, facilitating transmission in a distributed environment of various types of data, information, and knowledge structures (including atoms, frames, and relations) [17]. As different versions have been developed, CODER has matured to handle a variety of applications such as electronic mail messages [7], Navy intelligence messages [1], and access to literature on cardiology [10]. Lexical information, bibliographic records, thesauri, reference works, full-text, facsimile and other images, tabular data, hypertext, frames, semantic networks, and other forms have been processed. The collections of information already integrated into CODER have grown to hundreds of megabytes, and current efforts involve work on collections measured in gigabytes.

Our second direction has been to develop an object-oriented database system tailored to the information retrieval environment of interest, using minimal perfect hash functions (MPHFs) to ensure space and time efficient indexing. The LEND (Large External object-oriented Network Database) system, used in CODER, has evolved as well, through two major versions. While CODER originally used Prolog database facilities, or relied upon special manager routines coded in C to provide access to large collections of data or information, all shared access to information by CODER modules now involves use of Version 1 or Version 2 of LEND. A complete description of the current version of LEND appears in [4] and will also be published elsewhere. This paper focuses on a key part of LEND — our most recent algorithm for finding minimal perfect hash functions, which obtains results close to the theoretical lower bound.

2 Minimal Perfect Hashing

In our implementation of LEND, we use optimal hashing techniques to make operations as efficient as possible, providing:

- one-probe access to a record, given its key,
- no collisions to be resolved, and
- full utilization of hash table space.

*This work was funded in part by grants from the National Science Foundation (Grants IRI-8703580 and IRI-9116991) and PRC Inc.

15th Ann Int'l SIGIR '92/Denmark-6/92
© 1992 ACM 0-89791-524-0/92/0006/0266...$1.50

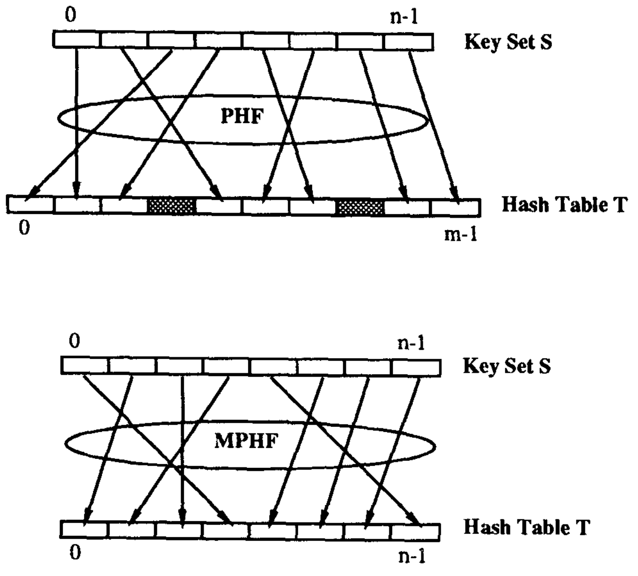

Figure 1: Perfect and Minimal Perfect Hash Functions

Optimal speed for hashing means that each key from the key set S will map to a unique location in the hash table T, thus avoiding time wasted in resolving collisions. That is achieved with a *perfect hash function (PHF)*, whose operation is illustrated at the top of Figure 1. When the hash table has minimal size, i.e., is fully loaded, with $|S| = |T|$, the hash function is called *minimal*. When both properties hold, one has a *minimal perfect hash function (MPHF)* as shown at the bottom of Figure 1. Note that, in reality, key set S itself is usually neither ordered nor sequential, but can clearly be indexed by the integers $0 \ldots n-1$ for convenience of illustration.

These hash functions can be grouped further into several categories. First, there are static and dynamic functions, when static or dynamic key sets are involved. Our emphasis has been on static functions, since in many retrieval and other applications the key sets change slowly, if at all (e.g., on CD-ROM).

Second, hash functions can point to individual objects, or to bins of objects. While LEND does support bin hashing [4], that discussion is beyond the scope of this paper, and in any case the methods used are derived from those considered here. Further, with large or variable size objects, or in-memory applications, direct location of single objects is desired.

Third, hash functions can preserve an a priori key ordering, or ignore that when ordered sequential access is not needed. In [8], we present some methods for building order-preserving minimal perfect hash functions. Since some types of OPMPHFs can be derived from MPHFs, we do not discuss that further here.

In this paper, we consider minimal perfect hash functions pointing at individual objects in static collections where there is no a priori key order that must be maintained. We explain a new algorithm to find MPHFs and give experimental evidence of its efficiency with large key sets.

3 MPHF Algorithm 1

To simplify discussion, we define essential terminology.

- U: key universe. $|U| = N$.
- S: actual key set. $S \subset U$, $|S| = n \ll N$.
- T: hash table. $|T| = m$, $m \geq n$.
- h: hash function. h: $U \rightarrow T$.
- h is a perfect hash function (PHF): no collisions, h is one-to-one on S.
- h is a minimal perfect hash function (MPHF): no collisions (i.e., PHF) and $m = n$.

For a given key set S taken from universe U, we desire a MPHF h that will map any key k in S to a unique slot in hash table T.

Until the 1980's there were no known algorithms to find MPHFs for large key sets. Since 1980, important contributions to the theory and practice of perfect hashing were made by various investigators including Cichelli [5], Jaeschke [13], Mehlhorn [14], Cercone, Krause, and Boates [2], Chang [3], Fredman and his colleagues [11, 12], and Sager [16]. The first practical algorithm for finding practical MPHFs for very large key sets, i.e., including millions of keys, was reported by Fox et al. The description [9] gives further details on earlier work as well.

The basic approach in [9] is to treat the problem as a search for desired functions in a large search space s. In actuality, preparatory Mapping and Ordering steps are needed so that fast Searching can take place. The overall Mapping-Ordering-Searching (MOS) scheme is illustrated in Figure 2. Mapping transforms the problem of hashing keys into a different problem, in a different space. Ordering paves the way for searching in that new space, so that locations can be identified in the hash table. Hashing then involves mapping from keys into the new space, and using the results of Searching to find the proper hash table location. From that perspective, the key results in [4, 9] are as follows.

- Search space s requires at least $1.4427n$ specification bits (at least $2^{1.4427n}$ distinct values must be in the space).
- Finding an MPHF is a search problem that determines some appropriate value in s for an instance S (which is the key set).
- S is related to s through partitioning both S and s into subsets S_i and s_i, for $i = 0, 1, 2, \ldots$

The basic algorithm discussed in [9], herein referred to as Algorithm 1:

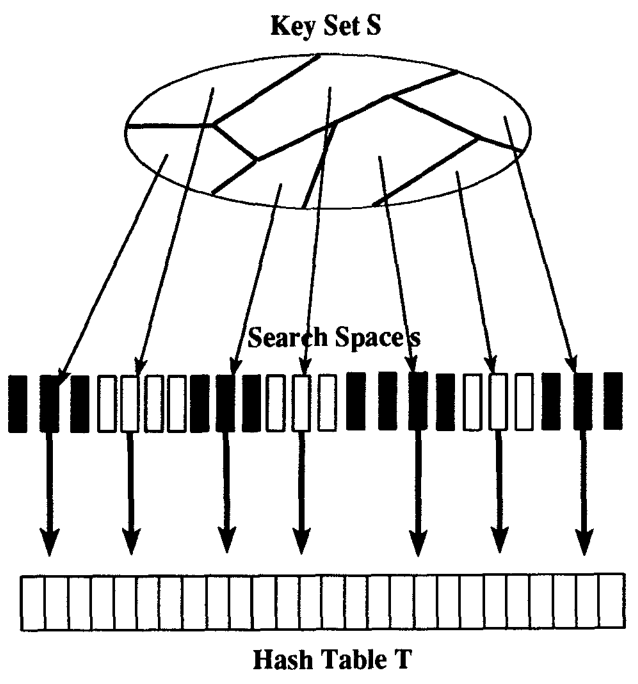

Figure 2: Illustration of the Key Concepts

- is a probabilistic algorithm;
- is based on ordering the vertices in a bipartite dependency graph;
- requires expected linear running time;
- handles large sets containing millions of keys; and
- yields MPHFs of size $c \log_2 n$ bits per key ($0.5 < c < 1$).

Its behavior in terms of bits per key required to find an MPHF in a reasonable amount of time, for varying size key sets, is illustrated in Figure 3.

Note that Algorithm 1 requires less than one **word** of specification space for each key in S. However, this is significantly more space than the theoretical lower bound, which is roughly 1.5 **bits** per key.

An alternative algorithm discussed in [9], called Algorithm 2 herein, did manage to produce MPHFs for large key sets with specification space size below 5 bits per key. Unfortunately, this method is relatively complicated, and finding the address for keys using an Algorithm 2 MPHF involves expensive multiplications. We have developed a new algorithm, described in the next section, which eliminates the need for multiplication, yields MPHFs with specification space below 2.5 bits per key, and is relatively easy to understand and implement.

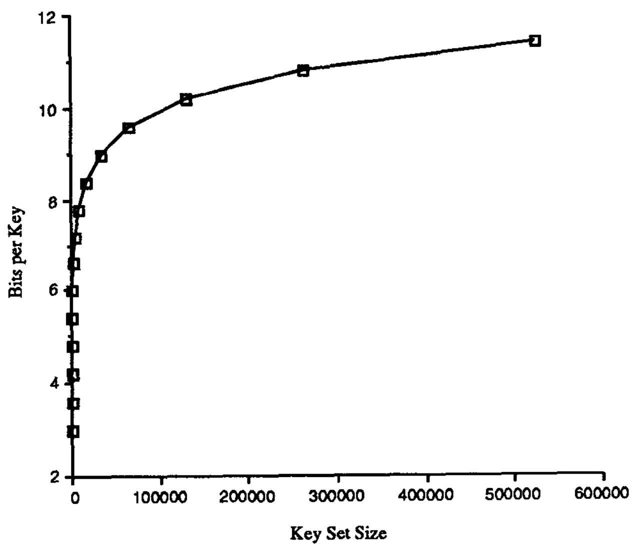

Figure 3: Bits per Key for Algorithm 1

4 MPHF Algorithm 3

The new algorithm for finding an MPHF, called herein Algorithm 3, is fully described in [4]. The basic results follow in this Section.

Algorithm 3 corrects many of the problems with Algorithm 1. First, Algorithm 1 makes use of moderately large tables to specify the mapping for the characters that make up keys, that in turn lead to the pseudo-random numbers used in the Mapping stage. By using and extending Pearson's method [15], mapping tables containing only 128 characters are produced. The results of the Mapping stage are sufficiently random so that more space-expensive approaches are unnecessary. Thus, only 128 bytes are used in the hash function specification to describe the Mapping process.

Second, in Algorithm 1, the Searching phase was less sophisticated, requiring many unsuccessful operations to locate an acceptable solution. By adding an auxiliary index data structure, we have been able to reduce the searching time significantly in Algorithm 3.

Third, Algorithm 3 deals with the need to reduce the size of the specification of the MPHF by radically changing the Mapping, Ordering, and Searching phases of Algorithm 1. In particular, no use is made of the bipartite dependency graph first suggested by Sager [16]. Rather, S is related to s in two steps:

- Keys are mapped to a bucket set B. (See Figure 4.) $b = |B| = \lceil cn/(\log_2 n + 1) \rceil, 2 \leq c \leq 4$.
- Keys in each bucket are separately mapped to T. (See Figure 5.)

In order to have space measured in bits per key instead of words per key, it is necessary to search for values whose

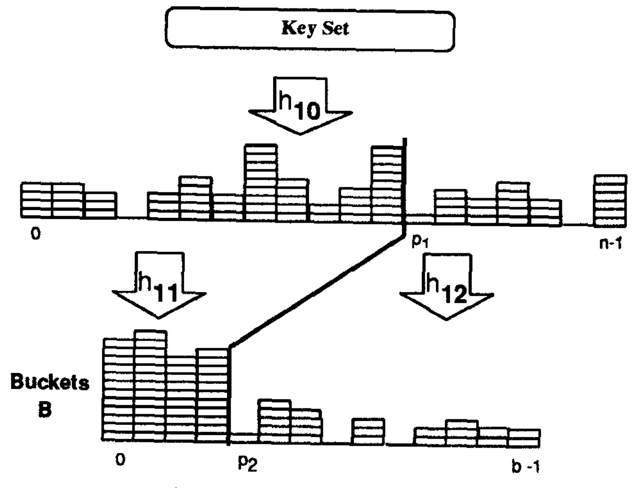

Figure 4: Mapping Stage of Algorithm 2

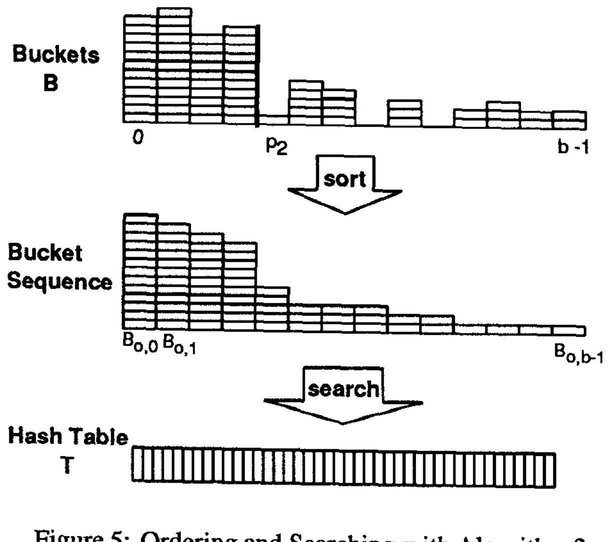

Figure 5: Ordering and Searching with Algorithm 2

number is proportional to $\lceil cn/(\log_2 n + 1)\rceil$ instead of n, as was done in Algorithm 1. This partially explains the need to introduce buckets into the process.

The Mapping stage, and the Ordering and Searching stages, are illustrated in Figures 4 and 5, respectively. Further details are given in the following subsections.

4.1 Mapping

The Mapping stage accomplishes several important goals. First, each of the n keys is mapped to an integer value, in the range $0 \ldots n-1$. This is done by pseudo-random hash function h_{10} which maps several keys onto some values and may leave other address values without any keys. See the top of Figure 5 for an illustration of the process.

$$h_{10} \quad : \quad S \rightarrow \{0, \ldots, n-1\}$$

Second, we wish to shrink the range of integer values from n to b so that later we need only search among b values. Finding an MPHF which has specification size close to the lower bound can be accomplished when c is close to 2, i.e., when b is roughly $2n/\log_2 n$. We can accomplish this by composing h_{10} with another function that will map into the range $0 \ldots b-1$.

However, in the process we can, if we are clever, accomplish a third goal. In particular, we wish to separate the keys into two major groupings. Our second function, then, is really accomplished by two functions that operate upon disjoint portions of $0 \ldots n-1$.

$$\begin{aligned} h_{11} &: \{0, \ldots, p_1 - 1\} \rightarrow \{0, \ldots, p_2 - 1\} \\ h_{12} &: \{p_1, \ldots, n-1\} \rightarrow \{p_2, \ldots, b-1\} \end{aligned}$$

These, together with h_{10}, accomplish the mapping from keys to buckets.

$$bucket(k) \quad = \quad \begin{cases} h_{11} \circ h_{10} & \text{if } h_{10}(k) < p_1 \\ h_{12} \circ h_{10} & \text{otherwise} \end{cases}$$

Thus, the mapping function $bucket(k)$ is composed of three functions: h_{10} randomly distributes keys into an auxiliary integer set $\{0, \ldots, n-1\}$, h_{11} and h_{12} in turn randomly deliver them into B, in particular into the unequal size subsets B^1 and B^2. Note that h_{11} and h_{12} depend on two parameters p_1 and p_2. Good values for these two parameters are experimentally determined to be around $0.6n$ and $0.3b$, respectively.

What this means is that roughly 60% of the keys (since $p_1 = 0.6n$ and h_{10} is likely to be relatively uniform at a coarse level) are mapped into roughly 30% of the buckets (since $p_2 = 0.3n$), i.e., $B^1 = \{0, \ldots, p_2 - 1\}$. In effect, we are forcing the buckets produced by h_{11} to each hold many keys. This is fine, since our earlier work with searching indicates that large groups of keys can be managed if dealt with early in the search process.

At the same time, the other 40% of the keys are "spread" by h_{12} into 70% of the buckets, i.e., $B^2 = \{p_2, \ldots, b - 1\}$, yielding fewer keys per bucket. This is handy since during searching it is desirable to have small groups of keys processed towards the end of the operation.

In summary, the Mapping stage, illustrated in Figure 5, accomplishes our goals of mapping to integers, compressing the range of integers, and separating big from small groupings of keys.

4.2 Ordering

During the Ordering stage, illustrated in the top portion of Figure 5, we use the organization developed during Mapping to prepare for Searching. The key features of this stage are as follows.

- Buckets are ordered by decreasing sizes to obtain the bucket sequence:

  $$\{B_{o,0}, B_{o,1}, \ldots, B_{o,b-1}\}.$$

  (where the subscript o designates ordered buckets as opposed to initial buckets)

- Bucket sorting can be used as the maximal number of keys in B is known.

Analysis indicates that because of our use of pseudo-random functions at each stage of the Mapping stage, we can estimate the number of buckets of each size. Even for very large key sets the largest buckets will have relatively small sizes. Clearly then a single pass through the buckets will yield the desired bucket sequence. Searching processes all keys in a bucket together, and proceeds from the largest to the smallest buckets.

4.3 Searching

The Searching stage involves choosing a $\log_2 n + 1$ bit parameter value $g()$ for each of the buckets, so that each key in each bucket can be mapped by the finally constructed hash function, h, to a previously unused slot in the hash table T.

Essentially, the group of keys in a bucket must all be "fit" into T at the same time, since they are mutually constrained by virtue of the earlier processing that put them into the same bucket. Choosing the parameter value for the bucket must assure that its "pattern" of entries can be "fit" into open slots in T. As we try different $g()$ values, we "rotate" the pattern until we find a good fit.

The Searching process maps keys in each bucket $B_{o,i}$ to T via the function h:

$$\begin{aligned} h_{20}: &\quad S \times \{0,1\} \rightarrow \{0, \ldots, n-1\} \\ h(k) &= \{h_{20}(k, d) + g(B_{o,i})\} \ mod \ n. \end{aligned}$$

This final hashing function $h()$ has simple form and is easily computable for any key in S. It is formed as the sum of two values.

- h_{20} is a pseudo-random function mapping keys in each $B_{o,i}$ to distinct values in $\{0, n-1\}$. Recall that $\log_2 n+1$ bits are allocated to each bucket. A designated bit d in these bits is used by h_{20} as part of the seed. As 0 and 1 can be the value for d, h_{20} can generate two different sets of integers for keys in $B_{o,i}$. This adds a degree of freedom to the searching, avoiding failures by changing the d values as needed. An integer r global to all buckets has also been used as part of the seed to h_{20}. Should some bucket fail to be mapped to distinct integers, a new value for r is tried. With the help of r, the same bucket sequence can be maintained. In the following, we use the term pattern set P_i for the set of values of h_{20} corresponding to keys in $B_{o,i}$.
- Each $g(B_{o,i})$ takes $\log_2 n$ bits.
- $g(B_{o,i})$ rotates the pattern set for a fit.
- $g(B_{o,i})$ can be selected by aligning an item in the pattern with an empty slot in T. (This is an important heuristic to improve efficiency.)

4.3.1 Auxiliary Data Structure

During the Searching phase, a considerable speedup results from using an auxiliary index data structure to locate empty slots. Recall that a fit means that h maps each member in the pattern set P_i to an empty slot. Therefore, an arbitrary member in P_i can be aligned with an empty slot, and testing can then determine whether the remaining members fit into other empty slots. A proper alignment then yields a proper g value. We define:

$$\begin{aligned} x &\equiv \text{the index of the empty slot, and} \\ u &\equiv \text{the member of} P_i \text{ to be aligned with } x. \end{aligned}$$

The rotation offset or $g(B_{o,i})$ is $(x - u)$ mod n. The method gives considerable speedup when key sets are of moderate to large size.

Figure 6 illustrates the auxiliary index data structure, along with the hash table. In the program and diagram, there are three arrays called `randomTable`, `mapTable` and `hashTable`. The `randomTable[0,n-1]` array is used to remember currently empty slots in the hash table. As it is preferable for each P_i to fill the hash table in a

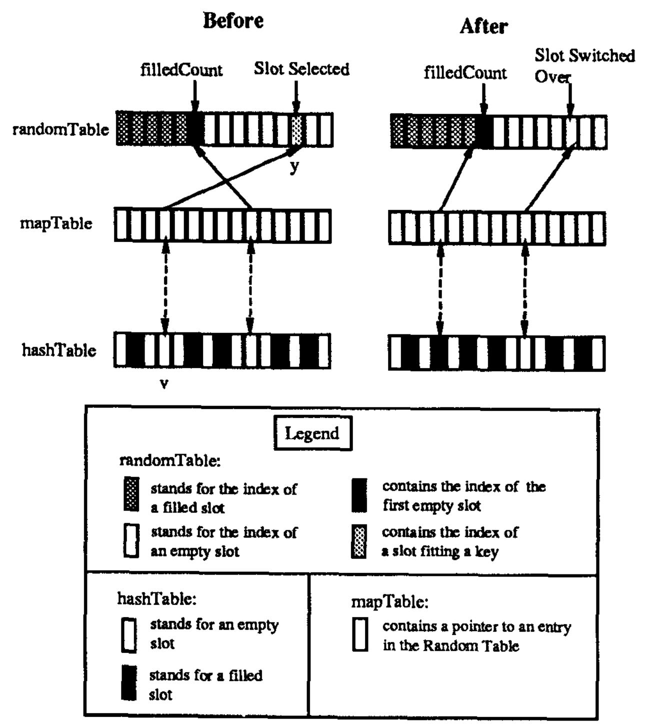
Before
After
filledCount
Slot Selected
filledCount
Slot Switched Over
randomTable
y
mapTable
hashTable
v
Legend
randomTable:
stands for the index of a filled slot
contains the index of the first empty slot
stands for the index of an empty slot
contains the index of a slot fitting a key
hashTable:
stands for an empty slot
stands for a filled slot
mapTable:
contains a pointer to an entry in the Random Table

Figure 6: Auxiliary Index Data Structure and Filling of a Hash Table Slot

random fashion, this array initially contains a random permutation of the hash addresses in $[0, n-1]$. The pointer `filledCount` is initially 0. It is an invariant that any slots to the right side of `filledCount` (inclusive) are empty and any ones to the left are filled. This property guarantees only empty slots are searched to fit P_i. For any unfilled slot x in `hashTable[]`, the `mapTable[0,n-1]` array contains pointers pointing at `randomTable[]` such that `randomTable[mapTable[`x`]]` $\equiv x$. Thus, given an empty slot x in the hash table, we can locate its position in the `randomTable[]` array through `mapTable[]`. Suppose a slot v of `hashTable`, which is referred to in location y in `randomTable[]`, needs to be filled and the invariant needs to be maintained after the filling action. Then we can just switch the pointers corresponding to `mapTable[randomTable[filledCount]]` and `mapTable[`y`]` and advance `filledCount` right one position. See the positions of the two differently shaded boxes in the topmost part of Figure 6. When $|P_i| > 1$, a sequence of switching is required.

4.3.2 Analysis of Tests Required to Fit a Pattern

Analytical study of the Search process lets us predict the number of tests that are needed during searching. The cost of fitting a pattern of size j into a n slots hash table T with f slots already filled can be approximated in the following way. The total number of slot subsets of size j from T is $\binom{n}{j}$, out

j	n=1K, f=40		n=1K, f=120	
	Avg.	Exp.	Avg.	Exp.
4	2.1	1.2	2.5	1.7
13	2.6	1.7	5.6	5.1
22	3.4	2.4	18	16
31	3.8	3.5	55	51

j	n=4K, f=163		n=4K, f=489	
	Avg.	Exp.	Avg.	Exp.
7	2.2	1.3	2.8	2.4
13	2.4	1.7	4.8	5.2
19	2.5	2.2	11	11
25	2.6	2.8	26	24

j	n=8K, f=327		n=8K, f=654	
	Avg.	Exp.	Avg.	Exp.
9	2.4	1.4	3.0	2.1
17	2.7	2.0	5.1	4.1
25	2.8	2.8	8.7	8.0
33	4.6	3.9	21	16

Table 1: Number of Tests — Average vs. Expected Value

of which only $\binom{n-f}{j}$ can fit the pattern. Imagine $\binom{n}{j}$ subsets as $\binom{n}{j}$ balls in a bag, and among them $W_j \equiv \binom{n-f}{j}$ are white balls and $B_j \equiv \binom{n}{j} - \binom{n-f}{j}$ are black balls. The cost of fitting the pattern is equivalent to repeatedly drawing balls from the bag until the first white ball is seen, without putting back previously drawn black balls. Let V_j be a random variable equating to the number of draws to obtain the first white ball in such an experiment. We have

$$\Pr(V_j = z) = \left(\prod_{r=0}^{z-1} \frac{B_j - r}{B_j + W_j - r}\right) \frac{W_j}{B_j + W_j}.$$

When j is small and n is large, the fitting process can be approximated as a Bernoulli experiment where the balls after being drawn are returned to the bag. Let Z_j be a random variable equating to the number of draws to obtain the first white ball in the Bernoulli experiment. We have

$$\Pr(Z_j = z) = \left(\frac{B_j}{B_j + W_j}\right)^{z-1} \frac{W_j}{B_j + W_j}.$$

Let

$$p = \frac{W_j}{B_j + W_j}.$$

Then, the expected value of Z_j is $1/p$.

This simple formula can be used in the algorithm to predict the number of tests for a fit. If the predicated value is too large ($> n$), then there is no point to attempt fitting the pattern. The expected values closely match those found empirically, as given in Table 1.

This situation is further illustrated in Figure 7, which records the number of tests required during the Searching

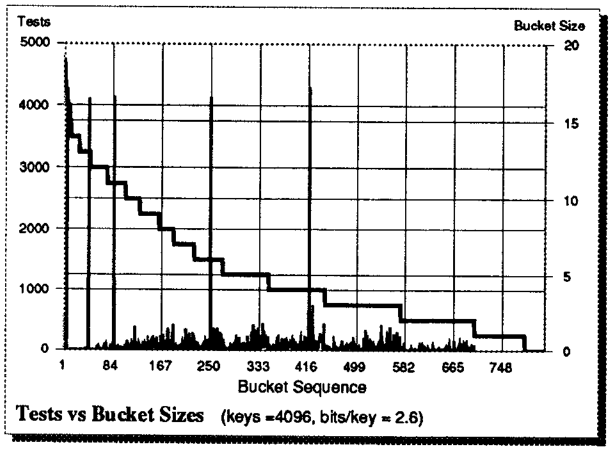

Figure 7: Tests vs. Bucket Sizes – 4K keys, 2.6 bits/key

stage. The horizontal axis shows the progression over time as buckets are processed. The staircase curve shows the size of the buckets as they are handled. The labels on the right show the range of bucket sizes, from less than 20 down to 1. The number of tests required to handle each bucket, as indicated by labels on the left, is shown by the many "spikes" near the horizontal axis (but sometimes rising to over 4000, which indicates that the original designated bit d for the bucket did not work and had to be changed). In general, the number of tests required is relatively small.

In summary, the Searching phase computes $g()$ values that properly rotate patterns until all elements of a bucket fit into the hash table. Our auxiliary data structure speeds up searching by assuring we make tests in a random fashion, by avoiding tests involving previously filled slots, and by reducing the number of memory accesses for each test. Our analysis yields an estimate for the number of tests needed at any given stage of the processing, allowing us to omit testing when failure is likely. Empirical studies show our estimates to be quite accurate, indicating that Searching is generally fast.

4.4 Timing Results

Algorithm 3 has been applied to a wide variety of key sets, from small ones to very large ones. A 256-key set and the resulting hash function specification, for example, is shown in its entirety in [4]. The 3.8 million key set used in previous studies has been processed using Algorithm 3 with parameters set to obtain MPHFs with 2.4 up to 3.5 bits per key. Results are given in Table 2. Note that our algorithm processes very large key sets in "chunks" which are saved on disk when all cannot fit into primary memory.

Figure 8 illustrates how the Searching time, and hence the total time, for finding MPHFs varies with bits per key requirements. We have found similar behavior with a key set of French words numbering approximately one half million.

Bits/Key	Map	Order	Search	Total
2.4	1890	5928	35889	43706
2.5	1886	5936	25521	33343
2.6	1887	5978	18938	26802
2.7	1887	6048	14486	22421
2.8	1897	6170	11602	19669
2.9	1894	6088	9524	17506
3.0	1905	6108	8083	16095
3.1	1894	6119	6998	15011
3.2	1885	6141	6110	14136
3.3	1884	6224	5436	13544
3.4	1886	6197	4958	13041
3.5	1886	6191	4586	12663

Note: CPU times are for NeXTstation (68040, 64MB), cc++ v.1.36.4, GNU g++ library v.1.39, 3,875,766 keys in 5 chunks (of 800K)

Table 2: Timing Results for Algorithm 2

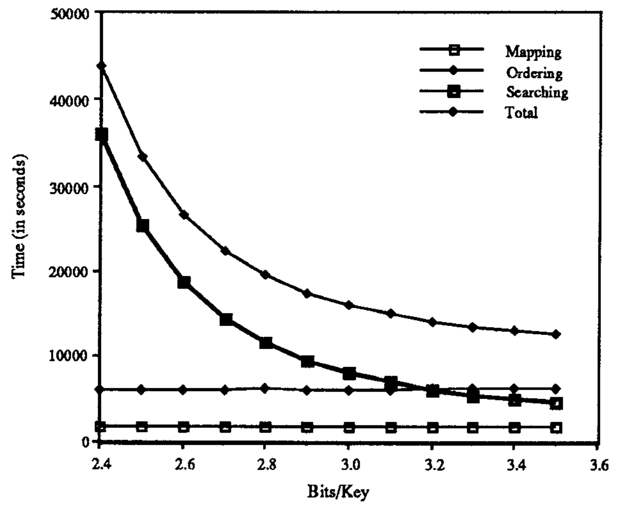

Figure 8: Plot of Table 2 Timing Results

Algorithm 3 seems to be able to find MPHFs for very large key sets using less than 3 bits per key of specification space, the most space efficient results that have been reported to date. Note that when more than 3 bits per key are used, there is a linear relationship between key set size and total time to find an MPHF; as the lower bound for bits per key is approached, the time required grows rapidly as more and more time is used to fit a bucket into the hash table during the Searching process.

Note that Algorithm 3 was able to find an MPHF for the 3.8 million key set in about 6 hours on a NeXT workstation, with 2.7 bits per key. This translates into about a megabyte of space needed to store the MPHF specification for one of the largest key sets we have been able to identify, suggesting that Algorithm 3 is quite feasible for use on modern workstations.

5 Conclusion

This paper has discussed a new fast algorithm for finding minimal perfect hash functions. Even the largest key set we have found can be processed in a number of hours on modern workstations using our new algorithm. With about 2.5 bits per key of space for the MPHF specification, single access to a key is guaranteed, using a fully loaded hash table.

The algorithm described has been applied to the problem of trie compaction, yielding efficient operation with greatly reduced space utilization. A related algorithm, for bin hashing, where $m = kb$ keys are perfectly distributed into b buckets each holding a group of k keys, has also been developed. These two methods are included in the LEND system and explained in [4], along with discussions regarding the utility of hashing methods for information retrieval applications.

6 Acknowledgements

Professor Abraham Bookstein provided the large French word list used in our study from the ARTFL Project at the University of Chicago. Dr. Martin Dillon, of OCLC Inc. in Dublin, Ohio provided the data used in our 3.8 million key file from their catalog records.

References

[1] Richard Barnhart. The Advanced Naval Message Analyzer. Videotape production, VPI&SU, November 1990. Richard Barnhart as host, producer, script-writer; Edward Fox as executive producer, project director. Discusses the CODER system.

[2] N. Cercone, M. Krause, and J. Boates. Minimal and almost minimal perfect hash function search with application to natural language lexicon design. *Computers and Mathematics with Applications*, 9:215–231, 1983.

[3] C. C. Chang. Letter oriented reciprocal hashing scheme. *Information Sciences*, 38:243–255, 1986.

[4] Qi Fan Chen. *An object-oriented database system for efficient information retrieval applications*. PhD thesis, Virginia Tech Dept. of Computer Science, March 1992.

[5] R. J. Cichelli. Minimal perfect hash functions made simple. *Communications of the Association for Computing Machinery*, 23:17–19, 1980.

[6] E. A. Fox and Robert K. France. Architecture of an expert system for composite document analysis, representation and retrieval. *International Journal of Approximate Reasoning*, 1(1):151–175, 1987.

[7] Edward A. Fox. Development of the CODER system: A testbed for artificial intelligence methods in information retrieval. *Information Processing & Management*, 23(4):341–366, 1987.

[8] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9(3):281–308, July 1991.

[9] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the Association for Computing Machinery*, 35(1):105–121, January 1992.

[10] Edward A. Fox, M. Prabhakar Koushik, Qi Fan Chen, and Robert K. France. Integrated access to a large medical literature database. Technical Report TR-91-15, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, May 1991.

[11] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.

[12] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the Association for Computing Machinery*, 31:538–544, 1984.

[13] G. Jaeschke. Reciprocal hashing – a method for generating minimal perfect hash functions. *Communications of the Association for Computing Machinery*, 24:829–833, 1981.

[14] K. Mehlhorn. On the program size of perfect and universal hash functions. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 170–175, 1982.

[15] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the Association for Computing Machinery*, 33(6):677–680, June 1990.

[16] T. J. Sager. A polynomial time generator for minimal perfect hash functions. *Communications of the Association for Computing Machinery*, 28:523–532, 1985.

[17] M. Weaver, R. France, Q. Chen, and E. Fox. Using a frame-based language for information retrieval. *International Journal of Intelligent Systems*, 4(3):223–257, 1989.