LEND and Faster Algorithms for Constructing Minimal Perfect Hash Functions

Edward A. Fox, Qi Fan Chen,
and Lenwood S. Heath

TR 92-02

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

February 5, 1992

LEND and Faster Algorithms 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

Integrated access to data, information, and knowledge bases is an important goal which we have addressed in several ways. While the CODER (COmposite Document Expert/extended/effective Retrieval) system has helped in this regard, a new type of object-oriented database system supporting graphs is required to provide a firm and efficient foundation. The LEND (Large External object-oriented Network Database) system has been developed to provide efficient access to large collections of primitive or multimedia objects, semantic networks, thesauri, hypertexts, and information retrieval collections. The architecture of LEND calls for Storage, Object, and Application layers, all programmed using C++, with an extensible foundation of useful classes. An overview of LEND is given, emphasizing aspects that yield efficient operation. In particular, 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.

CR Categories and Subject Descriptors: E2 [**Data Storage Representations**]: Hash-table representations; H.2.2 [**Database Management**]: Physical Design—*Access methods;* H.2.4 [**Database Management**]: Systems; H.2.5 [**Database Management**]: Heterogeneous Databases; H.3.2 [**Information Storage and Retrieval**]: Information Storage—File organization

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: hashing, minimal perfect hash functions, object-oriented database system, perfect hash functions

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

 Submitted for publication to SIGIR '92. This work was funded in part by grants from the National Science Foundation (Grants IRI-8703580 and IRI-9116991) and PRC Inc.

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) [18]. CODER has matured as different versions have been developed, to handle a variety of applications such as electronic mail messages [7], Navy intelligence messages [1], and access to literature on cardiology [11]. 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. Here we focus on a recent release of Version 2, and its algorithms for finding MPHFs

2 LEND

LEND manages objects, including graph structures. We believe that this is the proper foundation to build data, information, and knowledge bases, and have set out to prove this hypothesis. Proponents of semantic network, hypertext, hypermedia, neural network, cluster, and network database approaches should need little convincing. Yet, it might be argued that traditional information retrieval, relational database, and other systems cannot easily be supported or included. In general, the counter-

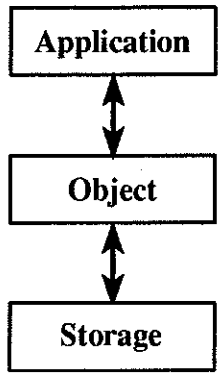

Figure 1: LEND's 3 Layers

arguments build upon concerns over two issues:

- the need for **efficiency**, and
- user requirements for appropriate **views.**

Regarding efficiency, we have carefully addressed this issue in our design and implementation work, relying in particular on the use of fast hashing algorithms, as discussed in later Sections. Regarding the latter point, LEND is extensible, since it is programmed in C++. There are three layers (see Figure 1), each extensible, though a rich foundation of usable classes already has been developed for each layer.

In particular, we expect LEND to be extended for each new type of application, by adding new classes that inherit properties from the existing classes. The Storage Layer now supports the following classes.

- UNIX file storage device (handling LRU access to files)
- page storage device (page prefetching, replacing, caching)
- hash functions (See Sections 3-5 for details.)
  - graph MPHF (Algorithm 1 in this paper — see also [10])
  - fast MPHF (Algorithm 3 in this paper)
  - OPMPHF (order preserving hash functions — see also [8])

  - bin hash functions

- index (locate, retrieve, next/iterate, insert, delete, etc.)
  - AVL tree index (main memory)
  - MPHF
  - OPMPHF

The Object Layer has classes that are primitive or base classes. Support is provided for persistency, memory or disk residency, nodes, and links. The classes include the following.

- object (retrieve object id, retrieve status, construct, destruct, compare, export, import, copy, reference)
  - integer
  - name, label
  - string, phrase
  - edge, semantic network edge
  - frame
- collection (memory or disk, iteration, set queries)

The Application Layer supports a variety of means to access LEND data, and other interfaces are planned or under development (e.g., one for the SNePS semantic network system). The first type of currently supported access is through C++ programs. This has been extensively used in CODER, and in routines for managing the MeSH (Medical Subject Headings) thesaurus from the National Library of Medicine. A second type of access is also used in CODER, to support its Fact, Function, and Frame Language (F3L). A third access scheme is for ad hoc user queries. Using a pattern specification language (introduced in [9]), users or programs can describe and retrieve nodes and subgraphs from a LEND database. This powerful capability supports variables, regular expressions (in terms of paths), path composition, and intersection and union operations on subgraphs.

Most of our experimentation to date with Version 2 of LEND has been with MeSH data — including loading, clustering, analysis, and retrieval efforts. We began with roughly 100 megabytes of ASCII data to characterize this thesaurus, which has about 290K nodes and 340K links. After careful analysis we developed 18 individual new classes in LEND. Efficiency comparisons have been made between one physical representation of the resulting graph, where all elements of a class are stored together, and another, where storage pages are selected to minimize the number of links that cross page boundaries [4].

For another application, large object support is being developed for facsimile storage and routing. Ultimately, a variety of "views" of LEND data will be provided. Currently, however, our emphasis is on demonstrating how LEND can be used in various versions of the CODER system.

Of particular interest is the MARIAN system, using parts of CODER, and building atop LEND, to allow search, retrieval, browsing, and annotation of the Virginia Tech online library catalog. To manage the almost one million records, and to support scores of simultaneous users on 2-3 NeXT systems, efficient operation is essential. Thus, we face a serious challenge for this practical application, and an interesting test of the underlying hypotheses for LEND, in terms of providing very efficient access to large volumes of data, with perhaps 20 transactions per second on each NeXT workstation.

3 LEND and Hashing

As mentioned in the discussion of LEND storage and object layers, hash functions are supported. In particular, we use optimal hashing techniques to make LEND as efficient as possible, providing:

- instant access to a record, given a key,
- no collisions to be resolved, and
- hash table space fully utilized.

Optimal speed for hashing means that each key will map to a unique location in the hash table, thus avoiding wasted time for collisions. That is achieved with a perfect hash function (PHF) as shown in the top of Figure 2. When the hash table has minimal size, i.e., is fully loaded, it is called a minimal hash function. When both properties hold,

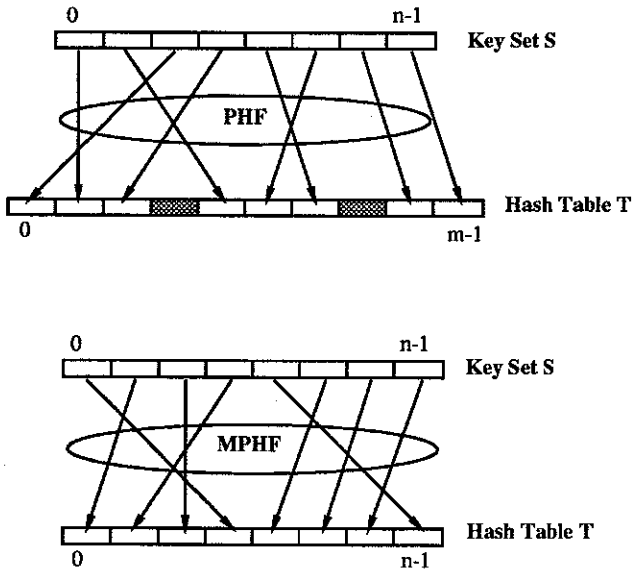

Figure 2: Perfect and Minimal Perfect Hash Functions

one has a minimal perfect hash function (MPHF) as shown at the bottom of Figure 2. Note that in reality key set S itself is usually neither ordered nor sequential, but can clearly be indexed by the integers $0 \ldots n-1$.

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). Other work in process on dynamic functions will be reported separately.

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 explain 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.

Thus, in the rest of this paper we consider minimal perfect hash functions, explaining a new algorithm to find MPHFs that is significantly better than our earlier methods.

4 MPHF Algorithm 1

To simplify discussion, we define essential terminology.

- U: key universe. $|U| = N$
- S: actual key set. $S \subset U$, $|S| = n$
- 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 minimal perfect hash function (MPHF): no collisions 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 1980s 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 [14], Mehlhorn [15], Cercone, Krause, and Boates [2], Chang [3], Fredman and his colleagues [12, 13], and Sager [17]. The first practical algorithm for finding practical MPHFs for very large key sets, i.e., including thousands or millions of keys, was reported in [10], which gives further details on earlier work as well.

The basic approach in [10] 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 scheme is illustrated in Figure 3. 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, 10] are as follows.

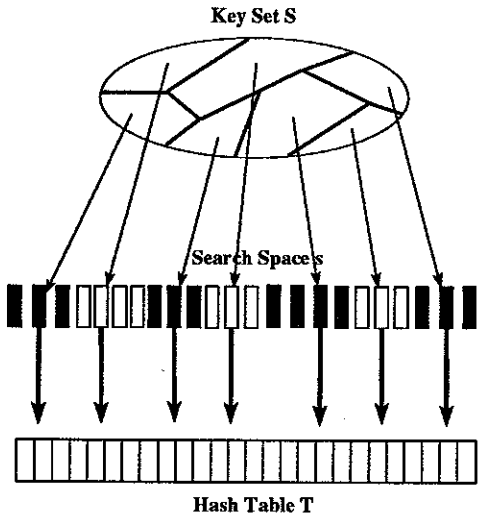

Figure 3: Illustration of the Key Concepts

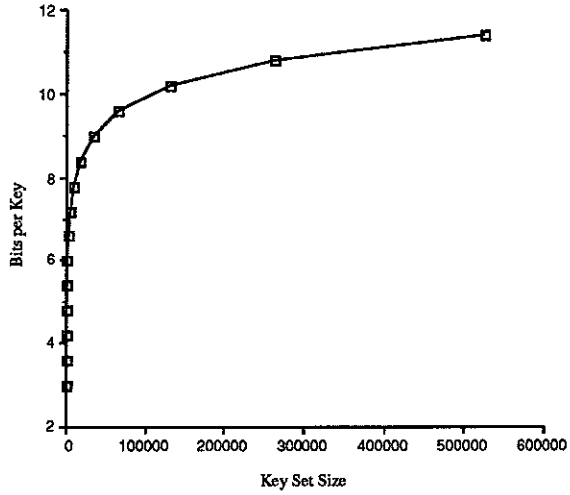

Figure 4: Bits per Key for Algorithm 1

- 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 the proper 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 [10], herein referred to as Algorithm 1:

- 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 4.

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 [10], called Algorithm 2 herein, requires approximately 4 bits per key in order to find an MPHF in a reasonable amount of time, but hashing then involves expensive multiplications and complex marking procedures. A much faster algorithm, based on further analysis of the problem and characteristics of key sets, yields MPHFs with 2-4 bits per key, as discussed in the next section.

5 MPHF Algorithm 3

The new algorithm for finding MPHF, called herein Algorithm 3, is fully described in [4] along with further discussion of LEND and other topics touched on in this paper. The basic results follow in this Section.

Algorithm 3 corrects many of the problems with Algorithm 1, which is implemented for the "graph MPHF" class of LEND. 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 [16], mapping tables containing only 128 characters are produced. The results of the Mapping stage are sufficiently random so that more space-expensive approaches are not needed. Thus, only 128 bytes are needed in the hash function specification to describe the Mapping process.

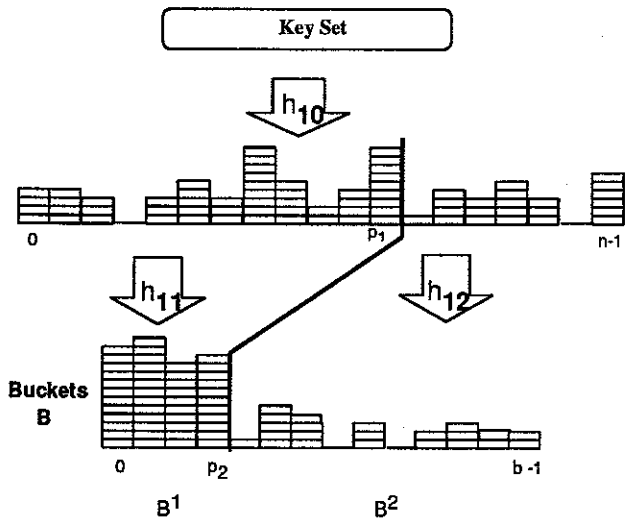

Figure 5: Mapping Stage of Algorithm 2

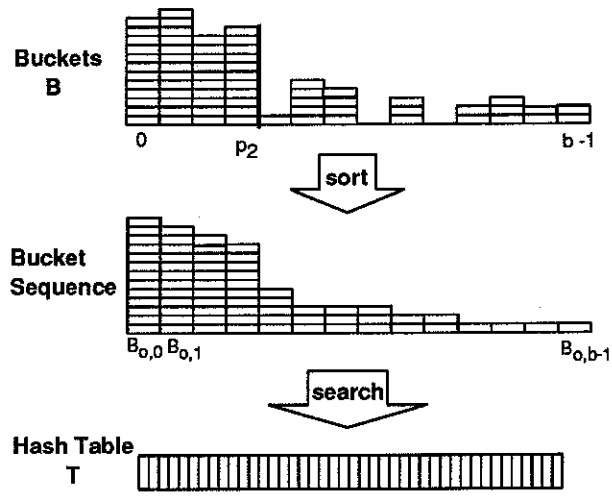

Figure 6: Ordering and Searching with Algorithm 2

Second, in Algorithm 1, the Searching phase was slow, requiring many unnecessary tries to locate an acceptable solution. By adding an 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 [17]. Rather, S is related to s in two steps:

- Keys are mapped to a bucket set B. (See Figure 5.)
  $b = |B| = \lceil cn/(\log_2 n + 1) \rceil$
  $2 \leq c \leq 4$
- Keys in each bucket are separately mapped to T. (See Figure 6.)

In order to have space measured in bits per key instead of words per key, it was necessary to search for values whose 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 5 and 6, respectively. Further details are given in the following subsections.

5.1 Mapping

The Mapping stage accomplishes several important goals. First, the n keys must be mapped to integer values, in the range $0 \ldots n-1$. This is done by pseudo-random hash function h_{10} which will almost certainly map several keys into the same address, and leave other addresses without any keys. See the top of Figure 5 for an illustration of the process.

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

Second, we wish to shrink the range of integer values from n to b so that later we need only search for 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 $2n/\log 2n$. We can accomplish this by composing $h_1 0$ 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 $0 \ldots n-1$.

$$h_{11} \ : \ \{0, \ldots, p_1 - 1\} \rightarrow \{0, \ldots, p_2 - 1\}$$

$$h_{12} \quad : \quad \{p_1, \ldots, n-1\} \rightarrow \{p_2, \ldots, b-1\}$$

These, together with $h_1 0$, accomplish the mapping from keys to buckets.

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

Thus, the mapping function $bucket(k)$ is composed of three functions: h_{10} randomly distributes keys into an auxiliary integer set $\{0, n-1\}$, 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 $h_1 1$ 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 say 40% of the keys are "spread" by $h_1 2$ into 70% of the buckets, i.e., $B^2 = \{p_2, \ldots, s-1\}$, 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 phase, illustrated in Figure 5, accomplishes our goals of mapping to integers, compressing the range of integers, and separating big from small groupings of keys.

5.2 Ordering

During the Ordering stage, illustrated in the top portion of Figure 6, we use the organization developed during Mapping to prepare for Searching. They 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 only have perhaps 20 entries. Clearly then a single pass through the buckets will yield the desired bucket sequence. Searching will deal with all of the buckets, processing all keys in a bucket together, and proceeding from the largest to the smallest buckets.

5.3 Searching

The Searching stage involves choosing a $\log 2n + 1$ 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" at the same time, since they are mutually constrained by virtue of the earlier processing that put them in 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.

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

$$\begin{aligned} h_{20}: \quad & Sx0, 1 \rightarrow \{0, \ldots, n-1\} \\ h(k): \quad & \{h_{20}(k, d) + g(B_{o,i})\} \ mod \ n. \end{aligned}$$

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 is 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.)

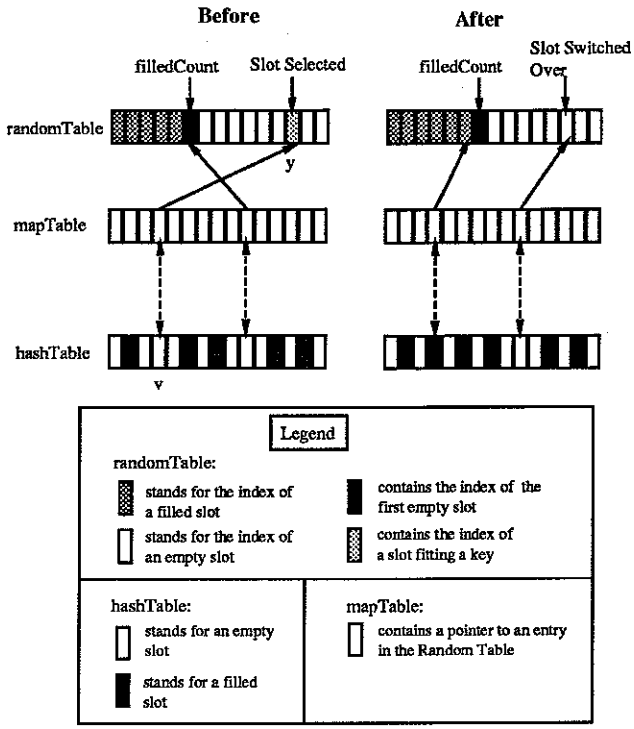

Figure 7: Index Data Structure and Filling of a Hash Table Slot

5.3.1 Auxiliary Data Structure

During the searching phase, a considerable speedup results from using an index data structure. Recall that a fit implies that each member in the pattern set P_i matches an empty slot. Therefore, an arbitrary member in P_i can be aligned with an empty slot, and testing can then determine whether the rest of the members fit into other empty slots. A proper alignment then yields a proper g value. We define:

$x \equiv$ the index of the empty slot
$u \equiv$ the member of P_i to be aligned with x.

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

Figure 7 illustrates the 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 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 filled 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 7. When $|P_i| > 1$, a sequence of switching is

required.

5.3.2 Analysis of Tries Required to Fit a Pattern

Analytical study of the search process lets us predict the number of tries 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 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

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

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

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

The expected value of Z_j is

$$\begin{aligned} \mathrm{E}(Z_j) &= \sum_{z=1}^{n} z\Pr(Z_j = z) \\ &\approx \sum_{z=1}^{\infty} z\Pr(Z_j = z) \end{aligned}$$

Denoting $\left(\frac{B_j}{B_j+W_j}\right)$ by q and $\left(\frac{W_j}{B_j+W_j}\right)$ by p, we have

$$\begin{aligned} \mathrm{E}(Z_j) &= \sum_{z=1}^{\infty} z\Pr(Z_j = z) \\ &= \sum_{z=1}^{\infty} z q^{z-1} p \\ &= p\sum_{z=1}^{\infty} \frac{d}{dq} q^z \\ &= p\frac{d}{dq}\frac{q}{1-q} \\ &= \frac{p}{(1-q)^2} \\ &= \frac{1}{p} \\ &= \frac{B_j + W_j}{W_j} \\ &= \frac{\binom{n}{j}}{\binom{n-f}{j}}. \end{aligned}$$

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

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

This situation is further illustrated in Figure 8, which records the number of tries required at each phase of the Searching stage. The horizontal axis shows the progression over time as buckets are processed. The staircase type curve shows the size of

[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, Qi Fan Chen, and Robert K. France. Integrating search and retrieval with hypertext. In Emily Berk and Joseph Devlin, editors, *Hypertext/Hypermedia Handbook*, pages 329–355. McGraw-Hill, Inc., New York, 1991.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

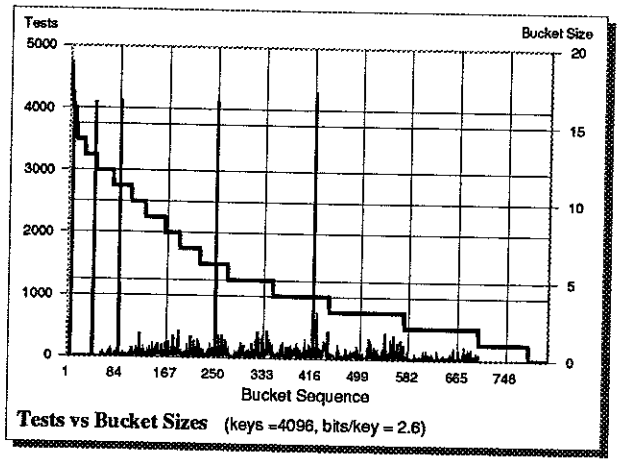

Figure 8: Tests vs. Bucket Sizes – 4K keys, 2.6 bits/key

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

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" value 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 tries in a random fashion, by avoiding tries of previously filled slots, and by reducing the number of memory accesses for each test. Our analysis yields a closed form estimate for the number of tries needed at any given stage of the processing, allowing us to omit testing when failure is likely. Empirical studies show our estimates to be relatively accurate, indicating that Searching is generally fast.

5.4 Timing Results

Algorithm 3 has been used with a wide variety of key sets, from very 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 9 illustrates how the Searching, 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. Algorithm 2 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 we believe has 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

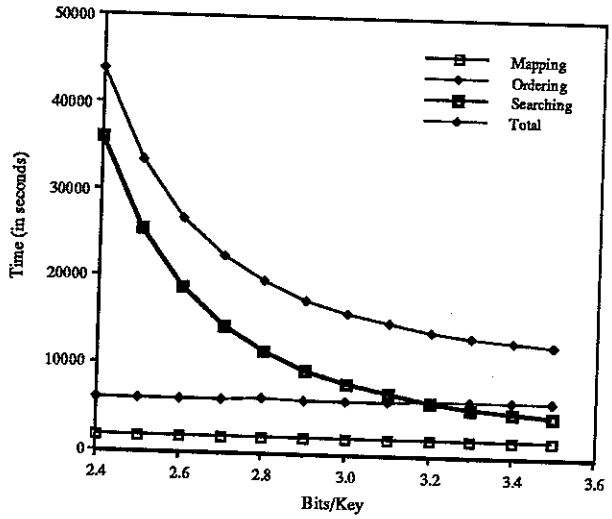

Figure 9: Plot of Table 2 Timing Results

that Algorithm 3 is quite feasible for use on modern workstations. These results are better than our previous best findings with Algorithm 2 [10], which yielded an MPHF with 4.58 bits per key, running on a Sequent (68030 processor) for about 9 hours.

6 Conclusion

This paper has discussed LEND and 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.

LEND supports not only rapid access to data through our various hashing functions, but also flexible pattern searching, and manipulation of representations of data, information, and knowledge. An early version of LEND was used in one production system by the Virginia Tech Computing Center, and the current release of LEND is being licensed for use at other locations. In concert with the CODER system, LEND will be used for a variety of applications illustrating how integrated information access can take place.

7 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. Sangita Betrabet assisted with some of the programming of LEND.

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 efficient object-oriented database for information retrieval.* PhD thesis, Virginia Tech Dept. of Computer Science, February 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.