Practical Minimal Perfect Hashing Functions for Large Databases

By Edward A. Fox, Lenwood Heath, Qi Fan Chen, and Amjad M. Daoud

TR 90-41

Practical Minimal Perfect Hash Functions for Large Databases *

Edward A. Fox Lenwood S. Heath Qi Fan Chen
Amjad M. Daoud
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

August 8, 1990

Abstract

We describe the first practical algorithms for finding minimal perfect hash functions that have been used to access very large databases (i.e., having over 1 million keys). This method extends earlier work wherein an $O(n^3)$ algorithm was devised, building upon prior work by Sager that described an $O(n^4)$ algorithm. Our first linear expected time algorithm makes use of three key insights: applying randomness wherever possible, ordering our search for hash functions based on the degree of the vertices in a graph that represents word dependencies, and viewing hash value assignment in terms of adding circular patterns of related words to a partially filled disk. Our second algorithm builds functions that are slightly more complex, but does not build a word dependency graph and so approaches the theoretical lower bound on function specification size. While ultimately applicable to a wide variety of data and file access needs, these algorithms have already proven useful in aiding our work in improving performance of CD-ROM systems and our construction of a Large External Network Database (LEND) for semantic networks and hypertext/hypermedia collections. Virginia Disc One includes a demonstration of a minimal perfect hash function running on a PC to access a 130,198 word list on that CD-ROM. Several other microcomputer, minicomputer, and parallel processor versions and applications of our algorithm have also been developed. Tests including those with a French word list of 420,878 entries and a library catalog key set with over 3.8 million keys have shown that our methods work with very large databases.

*This work was funded in part by grants from the National Science Foundation (Grant IRI-8703580), the Virginia Center for Innovative Technology (Grant INF-87-012), Nimbus Records, OCLC, and the State Council of Higher Education. AT&T and Apple Computer have provided equipment used in some of our experiments.

CR Categories and Subject Descriptors: E.2 **[Data Storage Representations]:** Hash-table representations; H.2.2 **[Database Management]:** Physical design—*access methods*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Hashing, perfect hash functions, minimal perfect hash functions, CD-ROM

1 Introduction

Ubiquitous in areas including artificial intelligence, data structures, database management, file processing, and information retrieval is the need to access items based on the value of a key. Classification systems use descriptors of various types, identifiers of myriad forms are assigned to items, and names for objects are commonplace. While various approaches to finding items through use of such keys have been explored, the promise of 'instant' access promised by *hashing* schemes is particularly enticing. However, most hashing methods involve a certain amount of wasted space (due to unused locations in a hash table) and wasted time (due to the need to resolve collisions).

Our objective is to improve upon current hashing techniques by eliminating these problems of wasted space and time. Our approach is to exploit the fact that there are many static collections of keys where it is worthwhile to undertake some preprocessing to build a minimal perfect hash function (MPHF) that will totally avoid the common problems of wasted space and time. Indeed, we have developed fast algorithms to search for MPHF's for large static key sets (i.e., over 1 million keys), and have used the resulting functions to improve access to large CD-ROM (compact disk, read-only memory) data collections, as well as to provide rapid access to a large lexicon built from machine readable dictionaries.

Static collections are rapidly becoming more common as non-erasable optical disc publishing activities increase [12]. CD-ROM production is increasing, and the use of WORM (write-once, read-many) units for archival storage or as part of a multi-level hierarchical memory system is growing. In addition to situations where the storage media enforces use of static files, there are natural cases where files rarely require revision. Dictionaries are published infrequently, and lexical databases generally expand rather slowly. Classification systems like the *Computing Reviews Category System* or the Library of Congress system for cataloging are slow to change. Our work with producing the first in the Virginia Disc series of CD-ROMs [13] and in constructing a large lexicon from machine readable dictionaries [18,17] thus naturally led us to commence an exploration of improved approaches to hashing.

1.1 Hashing

We begin with a collection of objects each of which has a (unique) associated *key*, say k, selected from U, a (usually finite) universe of keys. The cardinality of U is $N = |U|$. While some researchers assume that U is the set of integers

$$U = \{1 \ldots N\}$$

we adopt a less restrictive and more realistic assumption, and choose U to be the set of character strings having some finite maximum length. Clearly this is appropriate for keys that are words or names in any natural or artificial language, or that are elements of some descriptor or identifier set, possibly involving phrases as well.

The actual set of keys used in a particular database at a fixed point in time is $S \subset U$ where typically $|S| \ll |U|$. The cardinality of S is $n = |S|$. Records are stored in (objects are accessible through) a *hash table* T having $m \geq n$ locations (or *slots*), indexed by elements of $Z_m = \{0, 1, \ldots, m-1\}$. We measure the utilization of space in T by the *load factor*, $\alpha = n/m$. Depending on the application, T may be in primary memory, magnetic disk, optical disc, or recorded on some other device; in all cases it is desirable for T to be as small as possible and for us to be able to quickly find the appropriate slot in T for any given key k.

The retrieval problem is to locate the record corresponding to a key $k \in U$ or to report that no such record exists. We do this by *hashing*, i.e., applying a *hash function* h that is computable in time proportional to the size of the key k, and examining the slot in T with address $h(k)$. If there are two keys $k_1, k_2 \in S$ such that $h(k_1) = h(k_2)$, then there is a *collision* of k_1 and k_2. Much effort is expended in traditional work on hashing in resolving collisions. Collisions force more than one *probe* (reading a slot) of T to access some keys. If h is a 1-1 function when restricted to S, h is called a *perfect hash function* (PHF) since there is no need to waste time resolving collisions. A PHF h allows retrieval of records (objects) keyed from S in one access, which is clearly optimal in terms of time. For any form of hashing, optimal space is attained when the hash table is fully loaded, i.e., when $\alpha = 1$; we use the name *minimal hash function* for any function with this property. The best situation, then, is to have a *minimal perfect hash function* (MPHF) where $\alpha = 1$ and there are no collisions, so h is a 1-1 onto mapping when restricted to S.

1.2 Outline

In the following sections, we describe work involving perfect hash functions. In Section 2 we discuss related work, including the approach of Sager which was the starting point for our investigations. We begin that discussion with an explanation of some of the theoretical issues relating to perfect hash functions, and return to that perspective in Section 3 where we explain the key concepts that underpin our approach. For those less interested in the theory underlying our work, Section 3 can be largely ignored. (To make it possible to skip Section 3, some terms defined in Section 3 are defined again in Section 4.) Section 4 describes our first fast (i.e., linear expected time) algorithm for finding MPHF's, and Section 5 illustrates the procedure using a very small but realistic example. Section 6 reports on some of our experimental results, giving a characterization of the internal representations required during MPHF construction, timings for MPHF construction involving various size sets and various constraints on the process, and a description of a CD-ROM we have created that uses a MPHF to access a word list with more than 130,000 entries. Section 7 describes a second algorithm that approaches the theoretical limits on space and time. Section 8 deals with efforts to use MPHFs in connection with our lexicon construction effort, and design of a Large External Network Database (LEND) that involves MPHF access at the lowest level. Section 9 concludes the discussion.

2 Related Work

Hashing has been a topic of study for many years, both in regard to practical methods and analytical investigations [23]. Recently there has been renewed interest in hashing due to the development of techniques suitable for dynamic collections [10]. A less extensive literature has grown up, mostly during the last decade, dealing with perfect hash functions; it is that subarea that we consider in this section.

2.1 Mapping to Integers

Given a key k from a static key set S of cardinality n, selected from a universe of keys U with cardinality N, our object is to find a function h that maps each k to a distinct entry in the hash table T containing m slots. Certainly function h must map each key k to an integer. In the simplest cases, keys are positive integers bounded above by N. This situation was assumed, for example, by Sprugnoli [32], Jaeschke [22], and Fredman, Komlós, and Szemerédi [19].

We focus here, however, on the more general case where keys are strings (since clearly integers can be represented as strings of digits or strings of bytes, for example). The usual approach is to associate integer values with all or some of the characters in the string, and then to combine those values into a single number. Chang [4] used four tables based on the first and second letters of the key. Cichelli [6] used the length of the key and tables based on the first and last letters of the key. Note, however, that the length of a key, its first letter, and its last letter are sometimes insufficient to avoid collisions; consider the case of the words 'woman' and 'women' in Cichelli's method.

Cercone et al. [3] enhance the discriminating power of transformations from strings to integers by generating a number of letter to number tables, one for each letter position. Clearly, if the original keys are distinct, numbers formed by concatenating fixed length integers obtained from these conversion tables will be unique. In practice, it often suffices to simply form the sum or product of the sequence of integers.

While in some schemes (e.g., [6]) the resulting integer is actually the hash address desired, in most algorithms, the h function must further map from the integer value produced into the hash table.

2.2 Existence Proofs

One might ask if a MPHF h for a given key set exists. Jaeschke [22] proves this and indeed presents a scheme guaranteed to find such a function (though his method requires exponential time in the number of identifiers). Assuming that the task is to map a set of positive integers $\{k_1, k_2, \ldots, k_n\}$ bounded above by N without collisions into the set of m indices of T, we also have demonstrated a fast algorithm to define a suitable (though large!) PHF [16].

Another way to view this situation is to realize that perfect hash functions are rare in the set of all functions. Knuth [23] observes that only one in 10 million functions is a perfect hash function for mapping the 31 most frequently used English words into 41 addresses. Our task then can be viewed as one of searching for rare functions, and of specifying them in a reasonable amount of space.

2.3 Space to Store PHF

Using an argument due to Mehlhorn [25] (see also [24]), we have shown an approximate lower bound of ≈ 1.4427 bits per key to represent an arbitrary MPHF [16]. It is important to realize that this cost is *required* for one-probe access, no matter what scheme is employed. In addition to the lower bound, Mehlhorn also gives a method of constructing MPHFs of size $O(n)$ bits. However, the construction requires exponential time and therefore is not practical. A more practical algorithm of Mehlhorn constructs MPHFs of size $O(n \log_2 n)$ bits.

It is more typical to state the size of PHFs in terms of the number of computer words used, each of $\log_2 n$ bits. Then Mehlhorn's lower bound is $\Omega(n/\log_2 n)$ computer words, and his practical upper bound is $O(n \log_2 n/\log_2 n) = O(n)$ computer words. Our initial algorithm (sections 3-6) achieves MPHF size of less than $O(n)$ computer words, and our newest algorithm (section 7) approaches the theoretical lower bound.

2.4 Classes of Functions

There are several general strategies for finding perfect hash functions. The simplest one is to select a class of functions that is likely to include a number of perfect hash functions, and then to search for a MPHF in that class by assigning different values to each of the parameters characterizing the class.

Carter and Wegman [2] introduced the idea of a class H of functions that are $universal_2$, i.e., where no pair of distinct keys collide very often. By random selection from H, one can select candidate functions and expect that a hash function having a small number of collisions can be found quickly. This technique has also been applied to dynamic hashing by Ramakrishna and Larson [29].

Sprugnoli [32] proposes two classes of functions, one with two and the other with four parameters, that each may yield a MPHF; searching the parameter values of either class is feasible only for very small key sets. Jaeschke [22] suggests a reciprocal hashing scheme with three parameters that is guaranteed to find a MPHF, but it is only practical when $n \leq 20$. Chang [5] proposes a method with only one parameter, though its value is likely to be very large, and requires a function that assigns a distinct prime to each key; however, he gives no algorithm for that function, so the method is only of theoretical interest. A practical algorithm finding perfect hash functions for fairly large key sets is described in [7]. They illustrate the tradeoffs between time and size of the hash function, but do not give tight bounds on total time to find PHFs or experimental details for very large key sets.

The above-mentioned 'search-only' methods may (if general enough, and if enough time is allotted) directly yield a perfect hash function when the right assignment of parameters is identified. However, analysis of the lower bounds on the size of a suitable MPHF suggests that if parameter values are not to be virtually unbounded, then there must be a moderate number of parameters to assign. Thus, in the algorithms of Cichelli [6] and of Cercone et al. [3] we see two important concepts: using tables of values as the parameters, and using a mapping, ordering, and searching (MOS) approach (see Figure 1). While their tables seem to be too small to handle very large key sets, the MOS approach is an important contribution to the field of perfect hashing.

$$\boxed{MAPPING \rightarrow ORDERING \rightarrow SEARCHING}$$

Figure 1: Method to Find Perfect Hash Functions

In the MOS approach, the construction of a MPHF is accomplished in three steps. First, the **Mapping** step transforms the key set from the original universe to a new universe. Second, the **Ordering** step places the keys in a sequential order that determines the order in which hash values are assigned to keys. The Ordering step may partition the order into subsequences of consecutive keys. Such a subsequence is called a *level*, and the keys of each level must be assigned their hash values at the same time. Third, the **Searching** step attempts to assign hash values to the keys of each level. If the Searching step encounters a level that it is unable to accommodate, it backtracks, sometimes to an earlier level, assigns new hash values to the keys of that level, and tries again to assign hash values to later levels. Sager's method is a good example of the MOS approach.

2.5 Sager's Method and Improvement

Sager [30,31] proposes a formalization and extension of Cichelli's approach. Like Cichelli, he assumes that a key is a character string. In the Mapping step, three auxiliary hash functions are defined on the original universe of keys U

$$\begin{aligned} h_0 &: \ U \rightarrow \{0, \ldots, m-1\} \\ h_1 &: \ U \rightarrow \{0, \ldots, r-1\} \\ h_2 &: \ U \rightarrow \{r, \ldots, 2r-1\} \end{aligned}$$

where r is a parameter (typically $\leq m/2$) that ultimately determines how much space it takes to store the perfect hash function (i.e., $|h| = 2r$). These auxiliary functions compress each key k into a unique identifier

$$\big(h_0(k), h_1(k), h_2(k)\big)$$

which is a triple of integers in a new universe of size mr^2. The class of functions searched is

$$h(k) = \Big(h_0(k) + g\big(h_1(k)\big) + g\big(h_2(k)\big)\Big) \pmod{m}$$

where g is the function whose values are selected during the search.

Sager studies a graph that represents the constraints among keys. Indeed, the Mapping step goes from keys to triples to a special bipartite graph, the *dependency graph*, whose vertices are the $h_1()$ and $h_2()$ values and whose edges represent the words. The two parts of the dependency graph are the vertex set $\{0, \ldots, r-1\}$ and the vertex set $\{r, \ldots, 2r-1\}$. For each key k, there is an edge connecting $h_1(k)$ and $h_2(k)$; that edge carries the label k. See Figure 2.

In the Ordering step, Sager employs an ordering heuristic based on finding short cycles in the graph. This heuristic is called *mincycle*. At each iteration of the Ordering step, the mincycle

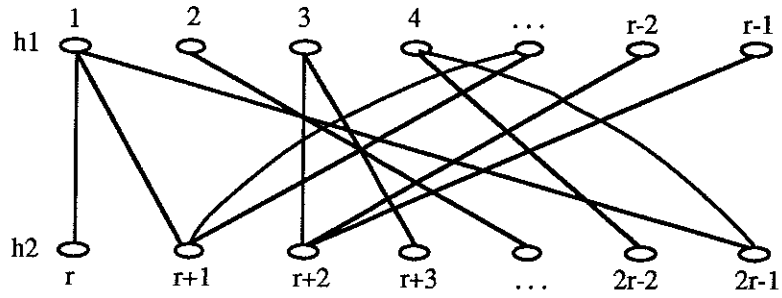


Figure 2: Dependency Graph

heuristic finds a set of unselected edges in the dependency graph that occur in as many small cycles as possible. The set of keys corresponding to the set of edges constitutes the next level in the ordering.

There is no proof given that a minimum perfect hash function can be found, but mincycle is very successful on sets of a few hundred keys. Mincycle takes $O(m^4)$ time and $O(m^3)$ space, while the subsequent Searching step usually takes only $O(m)$ time. The time and space required to implement mincycle are the primary barriers to using Sager's approach on larger sets.

Sager chooses values for r that are proportional to m. A typical value is $r = m/2$. In the case of minimal perfect hashing ($m = n$), it requires $2r = n$ computer words of $\log_2 n$ bits each to represent g. This is somewhat more than Mehlhorn's lower bound of $1.4427n/\log_2 n$ computer words. The *ratio* ($2r/n$) must be reduced as low as possible, certainly below 1. Our early work to explore and improve Sager's technique led to an implementation, with some slight improvements and with extensive instrumentation added on, described by Datta [9]. After further investigation we developed a modified algorithm [15] requiring $O(m^3)$ time. With this algorithm we were able to find MPHF's for sets of over a thousand words.

2.6 Summary of Related Work

Hashing has been used in many applications and, with recent development of dynamic hashing techniques, has witnessed a resurgence of interest [10]. However, most dynamic hashing involves low load values, on the order of 0.6 to 0.9, and requires resolution of collisions. In most dynamic hashing schemes, hash addresses identify buckets or bins wherein a number of records can be stored, and which are usually only partially full. The work of Gonnet and Larson does improve upon the typical approach, however, allowing high load factors, through the use of a small amount of extra storage used in buckets [20].

An examination of previous schemes for perfect hashing shows that many are generally only applicable to small sets, or require a prohibitive amount of space to store. Some approaches require input keys to already be integers in some restricted range, while others are really two level searches where extra storage in each bucket supports the second level of search. An example is the work of Brain and Tharp, which extends Cichelli's approach, but still cannot handle very large key sets [1].

The most competitive alternative to our approach comes from the work of Fredman, Komlós, and Szemérdi [19] and others who have built on their work (e.g., [31]). These methods assume a universe that is the set of integers $\{1, \ldots, N\}$ in contrast to our assumption of keys that are

arbitrary character strings. Membership queries are accommodated in constant time, and the hash function requires $n+o(n)$ space. Construction has random expected time $O(n)$. Our bounds are tighter, and really practical algorithms for very large key sets have been demonstrated for the first time only in our work.

3 Key Concepts of New Algorithms

After careful analysis of Sager's algorithm [31] and our enhanced version [15], Heath made three crucial observations that serve as the foundation of our new algorithms:

1. First, it appears that we must exploit randomness whenever possible. We trade a small probability of failure for an outstanding average case performance.

2. Second, the vertex degree distribution is highly skewed; this can be exploited to carry out the Ordering step in a much more efficient manner.

3. Third, assigning g values to a set of related words can be viewed as trying to fit a pattern into a partially filled disk, where it is important to enter large patterns while the disk is only partially full.

These three insights allow our new algorithms to obtain an Ordering in linear expected time. Since Mapping and Searching have expected time $O(n)$, the time complexity of our method is $O(n)$ in the expected sense.

We consider each of these key concepts in more detail in the next three subsections. Because we are primarily concerned with **minimal** perfect hash functions, we assume $m = n$ and use only n. All results easily generalize to the case $m \geq n$.

3.1 Randomness

Our first use of randomness is in the Mapping step where a good set of triples is obtained to serve as identifiers for the original word strings. Here it is essential that the triples are distinct. Our technique is essentially to obtain a random number for each key (mod n), making use of all of the information in the key to give maximum discrimination. The pseudo-random number generator we selected, to allow machine independence and to ensure good behavior, is due to Park and Miller [27]. The random integers generated by their random number generator are between 0 and $2^{31} - 2$. Now consider the use of these random numbers to obtain the desired triples. First, three tables ($table_0$, $table_1$, $table_2$) of random numbers are constructed, one for each of the functions h_0, h_1, and h_2. Each table contains one random number for each possible character at each position i in the key. Let a key be the character string $k = k_1 k_2 \ldots k_y$. Then the triple is computed using the following formulas:

$$h_0(k) = \left(\sum_{i=1}^{y} table_{0i}(k_i) \right) \bmod n$$

$$h_1(k) = \left(\sum_{i=1}^{y} table_{1i}(k_i)\right) \bmod r$$
$$h_2(k) = \left(\sum_{i=1}^{y} table_{2i}(k_i)\right) \bmod r + r.$$

Assuming the triples $(h_0(k), h_1(k), h_2(k)), k \in S$, are random, it is possible to derive the probability that the triples are distinct. Let $t = nr^2$ be the size of the universe of triples. The probability of distinctness for n triples chosen uniformly at random from t triples is

$$p(n,t) = \frac{t(t-1)\cdots(t-n+1)}{t^n} = \frac{(t)_n}{t^n}.$$

By an asymptotic estimate from Palmer [26],

$$p(n,t) = \frac{(t)_n}{n} \sim \exp\left\{-\frac{n^2}{2t} - \frac{n^3}{6t^2}\right\}$$

and for typical values of $r = cn/\log_2 n$, where c is a constant,

$$p(n,t) \approx \exp\left\{-\frac{n^2(\log_2 n)^2}{2n(cn)^2}\right\} = \exp\left\{-\frac{(\log_2 n)^2}{2c^2 n}\right\} \approx 1 - \frac{(\log_2 n)^2}{2c^2 n}$$

so that $p(n,t)$ goes to 1 quite rapidly with n. We almost always find a suitable set of triples the first time, and even have good success when h_0, h_1, and h_2 are all determined using permutations (different views) of one random table.

Pearson [28] advances an alternative approach that might allow smaller tables than ours. Further research can investigate this possibility.

3.2 Vertex Degree Distribution

Our second key concept is that the distribution of degrees of vertices in the dependency graph is decidedly skewed, and indeed has a number of interesting properties. In particular, we show that in a random dependency graph most vertices have low degree. We exploit this fact to obtain small levels in the ordering. Concentrate on a particular vertex v and the edges incident on it. The probability that a particular edge from edge set E is incident on v is $p = 1/r$. Let X be the random variable that equals the degree of v. Then (see [11]), X is binomially distributed with parameters n and p. Since the case of large n is of interest, the Poisson approximation to the binomial distribution applies:

$$\Pr(X = d) \approx \frac{e^{-\lambda}\lambda^d}{d!}$$

where $\lambda = np = n/r$. From the Poisson approximation, the expected number of vertices of degree d is given by

$$2r\Pr(X = d) \approx \frac{2re^{-n/r}\left(\frac{n}{r}\right)^d}{d!}$$

When $r = n/2$ ($\lambda = 2$), the expected number of vertices of degree 0, 1, 2, 3, and 4 are approximately 0.27r, 0.54r, 0.54r, 0.36r, and 0.18r, respectively. The skewed distribution of vertex degrees provides the inspiration for a new Ordering heuristic. Instead of ordering the *edges* (keys) of the dependency graph as mincycle does, the new heuristic orders the *vertices*. Let $v_1, v_2, \ldots, v_{2r}$ be any ordering of the vertices of the dependency graph. For each v_i, there is a set of edges $K(v_i)$ that go from v_i to vertices earlier in the ordering

$$K(v_i) = \{(v_i, v_j) \in E | j < i\}.$$

This set of edges is also a set of keys, and every key occurs in exactly one $K(v_i)$. $K(v_i)$ may be empty, but cannot be larger than the degree of v_i (it is often smaller). The ordering of the set of keys into levels is just the ordering of the nonempty $K(v_i)$.

As discussed in the next section, it is desirable to have levels that are as small as possible and to have all large levels early in the ordering. This suggests that a vertex of larger degree should be processed earlier than a vertex of smaller degree. This also suggests that a vertex whose K set is (currently) larger should be chosen next in the ordering over a vertex whose K set is (currently) smaller. Finally, it is imperative that the Ordering heuristic be able to choose the next vertex in the ordering quickly and simply. The new Ordering heuristic described in Section 4 evolved from these insights gained by examining the skewed distribution of vertex degrees and from the imperative of choosing the next vertex quickly.

3.3 Fitting into a Disk

At each iteration of the Searching step, one level of the ordering is to be placed in the hash table. Each level is the key set $K(v_i)$ corresponding to a vertex v_i. For purposes of illustration, assume that $v_i \in \{r, \ldots, 2r-1\}$. Each key $k \in K(v_i)$ has the same h_2 value $h_2(k) = v_i$ and, therefore, will have the same $g \circ h_2$ value $g(h_2(k)) = g(v_i)$. By assumption, the $g \circ h_1$ value of k is already selected. Since all h_0 values are already defined, $h(k)$ is determined by the selection of $g(v_i)$. Consider the sum of the two values already known for k. Let

$$b(k) = h_0(k) + g\big(h_1(k)\big).$$

Then

$$h(k) = \big(b(k) + g(v_i)\big) \pmod{n}.$$

The $b(k)$ values for all keys $k \in K(v_i)$ yield offsets from $g(v_i)$ (mod n) to the hash values of the keys.

The set of $b(k)$ values constitutes a *pattern* (mod n). The pattern may be viewed as being in a circle of n slots and subject to *rotation* by the amount $g(v_i)$. The hash table is viewed as a disk with n slots, some of which may already be filled. To successfully assign hash values to the keys in $K(v_i)$, the Searching step must determine an offset value $g(v_i)$ that puts all the $b(k) + g(v_i)$ values in empty slots of the hash table *simultaneously*. This process we refer to as fitting a pattern into a disk.

Finding hash values for a set of j related words corresponds to finding suitable g values so that the pattern of size j can be placed into the disk, with each of the j words fitting into an

empty slot. Clearly, when $j = 1$ this is possible as long as there is an empty slot. Further, regardless of the size of j, it is always possible to fit a pattern into an empty table. We would expect, therefore, to be able to find a hash function if vertices of large degree are handled when the disk is mostly empty, and if when the table is starting to get full, remaining vertices are of low degree, preferably degree 1. In [16] we derive the probability of fitting a pattern of size j into a m slots disk with f slots occupied already:

$$\Pr(\text{fit}) \quad = \quad 1 - e^{-\mu}$$

where

$$\mu = \left(\left(1 - \frac{f}{m} \right)^{j-1} (m - f) \right).$$

Note that if f is a function of m such that $f < (1 - \epsilon)m$, for some constant $\epsilon > 0$, then $\mu \to \infty$ and $\Pr(\text{fit}) \to 1$. For our purposes, this means that the disk must be slightly less than full ($f < (1 - \epsilon)m$) when the last pattern of size $j > 1$ is placed.

4 Algorithm 1 Outline

Our first algorithm for finding MPHFs for large key sets is an extension of earlier work by Sager [31] and builds on our new insights. To aid in subsequent discussion, we therefore summarize the terminology introduced by Sager and later extended by us as our method developed. Please refer to Table 1 for clarification in the discussion below. Because the algorithm that we describe finds **minimal** perfect hash functions, $m = n$, and we only mention n.

Recall that the class of functions from which the perfect hash function is selected is

$$h(k) = \Big(h_0(k) + g\big(h_1(k)\big) + g\big(h_2(k)\big) \Big) \bmod n$$

where

$$g : \{0, \ldots, 2r - 1\} \to \{0, \ldots, n - 1\}$$

is a function whose values are to be determined during the Searching step. r is a parameter that is typically $n/2$ or less. The larger r is, the greater the probability of finding a MPHF, but the greater the size of the resulting MPHF.

Our algorithm for selecting h has three steps: Mapping, Ordering, and Searching.

4.1 The Mapping Step

The Mapping step takes a set of n keys and produces the three auxiliary hash functions h_0, h_1, and h_2 (see Section 2.5). These three functions map each key k into a triple

$$\big(h_0(k), h_1(k), h_2(k)\big).$$

$U =$	universe of keys
$N =$	cardinality of U
$k =$	key for data record
$S =$	subset of U, set of keys in use
$n =$	cardinality of S
$T =$	hash ordering, with slots numbered $0, \ldots, (m-1)$
$m =$	number of slots in T
$h =$	function to map key k into hash ordering T
$\|h\| =$	space to store hash function
$r =$	parameter specifying the number of vertices in one part of the dependency graph
$h_0, h_1, h_2 =$	three separate random functions easily computable over keys
$g =$	function mapping $0, \ldots, (2r-1)$ into $0, \ldots, (m-1)$
$t =$	number of levels in the ordering

Table 1: Summary of Terminology

Because the ultimate MPHF must distinguish any two of the original keys, it is essential that these n triples be distinct. As discussed in Section 3.1, if h_0, h_1, and h_2 are random functions, it is very likely that the triples will be distinct. The h_0, h_1, and h_2 values are used to build a bipartite graph called the *dependency graph*. In turn, the graph can be employed to verify that triples are distinct.

Half of the vertices of the dependency graph correspond to the h_1 values and are labeled $0, \ldots, r-1$. The other half of the vertices correspond to the h_2 values and are labeled $r, \ldots, 2r-1$. There is one edge in the dependency graph for each key in the original set of keys. A key k corresponds to an edge labeled k between the vertex labeled $h_1(k)$ and the vertex labeled $h_2(k)$. Notice that there may be other edges between $h_1(k)$ and $h_2(k)$, but those edges are labeled with keys other than k. If the value $h_0(k)$ is associated with the edge k, then all the information that the Ordering and Searching steps need to construct a MPHF is present in the dependency graph.

Two data structures describe the dependency graph, one for the edges (keys) and one for the vertices (h_1 and h_2 values). Both are implemented as arrays. The `vertex` array is

```
vertex:  array [0..2r-1] of record
    firstedge:  integer;
    degree:  integer;
    g:  integer;
    end
```

`firstedge` is the header for a singly-linked list of the edges incident on the `vertex`. `degree` is the number of vertices incident on the vertex. `g` is the g value for the `vertex`, which is assigned in the Searching step. The `edge` array is

```
edge:  array [1..n] of record
```

```
(1)     build random tables for h0, h1, and h2
(2)     for each v ∈ [0...2r − 1] do
            vertex[v].firstedge = 0
            vertex[v].degree = 0
(3)     for each i ∈ [1...n] do
            edge[i].h0 = h0(ki)
            edge[i].h1 = h1(ki)
            edge[i].h2 = h2(ki)
            edge[i].nextedge1 = 0
            add edge[i] to linked list with header vertex[h1(ki)].firstedge
            increment vertex[h1(ki)].degree
            edge[i].nextedge2 = 0
            add edge[i] to linked list with header vertex[h2(ki)].firstedge
            increment vertex[h2(ki)].degree
(4)     for each v ∈ [0...r − 1] do
            check that all edges in linked list vertex[v].firstedge have
                distinct (h0,h1,h2) triples.
(5)     if triples not distinct then
            repeat from step (1).
```

Figure 3: The Mapping Step

```
h0, h1, h2: integer;
nextedge1: integer;
nextedge2: integer;
end
```

h_0, h_1, and h_2 contain the h_0, h_1, and h_2 values for the edge (key). Also, $\texttt{nextedge}_i$, for side i ($= 1, 2$) of the graph (corresponding to h_1, h_2, respectively), points to the next edge in the linked list whose head is given by firstedge in the vertex array.

Figure 3 details the Mapping step. Let $k_1, k_2, \ldots, k_n$ be the set of keys. The h_0, h_1, and h_2 functions are selected (1) as the result of building tables of random numbers as described in section 3.1. The construction of the dependency graph in (2) and (3) is straightforward. In (3) when edges are added to the appropriate linked list, values for $\texttt{nextedge}_i$ in the edge array are updated as needed. (4) examines sets of edges having the same h_1 value to check for distinct (h_0, h_1, h_2) triples; since vertex degrees are small (section 3.2), (4) takes expected time that is linear in n. In the rare (recall section 3.1) circumstance that distinct triples are not produced (5), new random tables are generated, defining new h_0, h_1, and h_2 functions. The probability that random tables must be generated more than twice is exceedingly small. Therefore, the expected time for the Mapping step is $O(n)$.

4.2 The Ordering Step

The Ordering step explores the dependency graph so as to partition the set of keys into a sequence of levels. The step actually produces an ordering of the vertices of the dependency graph (at least those that do not have degree zero). From the vertex ordering, the sequence of levels is easily derived. If the vertex ordering is $v_1, \ldots, v_t$, then the level of keys $K(v_i)$ corresponding to a vertex v_i, $1 \leq i \leq t$, is the set of edges incident both to v_i and to a vertex earlier in the ordering. More formally, if $0 \leq v_i \leq r-1$, then let $x = 1$, $y = 2$, while if $r \leq v_i \leq 2r-1$, then let $x = 2$, $y = 1$.

$$K(v_i) = \{k_j | h_x(k_j) = v_i, h_y(k_j) = v_s,\ s < i\}.$$

The rationale for the vertex ordering is discussed in section 3.2.

An analogy with Prim's algorithm [33] for constructing a minimum spanning tree will help illuminate the heuristic for ordering vertices. At each iteration of Prim's algorithm, an edge is added to the minimum spanning tree that is lowest in cost such that one endpoint of the edge is in the tree and the other endpoint is not. Of course, when an edge is added to the tree, so is a vertex. One implementation of Prim's algorithm maintains the unexamined edges that have at least one endpoint in the tree in a heap so that the cheapest edge can always be selected in logarithmic time.

Our ordering heuristic initiates the ordering with a vertex v_1 of maximal degree. At each iteration of the Ordering step, a previously unselected vertex v_i is added to the ordering. v_i is selected from among those unselected vertices that are adjacent to at least one of $v_1, \ldots, v_{i-1}$; from among these vertices, v_i is selected to have maximal degree. If there are no such unselected vertices and there remain unselected vertices of nonzero degree (i.e., another connected component needs to be processed), then select any vertex of maximal degree to be v_i. The algorithm maintains the unselected vertices that are adjacent to selected vertices in a heap `VHEAP` ordered by degree. Figure 4 gives the Ordering step.

The heap operations are **`initialize`** (start an empty heap), **`insert`** (add a vertex to the heap), and **`deletemax`** (select a vertex of maximum degree and remove it from the heap). Each heap operation can be accomplished in $O(\log n)$ time (since $r = O(n)$). Because the vertex degrees of a random dependency graph are (mostly) small, there is an optimization possible to speed the heap operation. In this optimization, `VHEAP` is implemented as a series of stacks and one bounded size heap. Most vertices have degree 1, 2, 3, or 4. One stack is provided for each degree, so that the size of the heap is kept below a constant. Usually, the heap `VOHEAP` contains all vertices of degree ≥ 5. When a vertex w is to be added to `VHEAP`, the degree of w is checked. If the degree is ≤ 4, then w is added to the appropriate stack; otherwise, w is inserted in `VOHEAP`. All list operations take constant time. The time for the Ordering step is thus actually linear.

There is one issue not addressed in Figure 4: the dependency graph may not be connected. Typically, the dependency graph consists of one large connected component and a number of smaller components. By choosing v_1 as a vertex of maximal degree, the algorithm is almost certainly choosing v_1 in the large component. Therefore, the algorithm selects the large component first. After that, it must process the remaining components in the same fashion. The algorithm maintains a list of those vertices not yet selected and can easily find an unselected vertex of maximal degree. Therefore, the Ordering step is able to order all vertices of degree > 0.

```
initialize(VHEAP)
v1 = a vertex of maximum degree
mark v1 SELECTED
for each w adjacent to v1 do
    insert(w, VHEAP)
i = 2
while some vertex of nonzero degree is not SELECTED do
    while VHEAP is not empty do vi = deletemax(VHEAP)
        mark vi SELECTED
        for w adjacent to vi do
            if w is not SELECTED and w is not in VHEAP then
                insert(w, VHEAP)
        i = i + 1
```

Figure 4: The Ordering Step

In our most recent implementation, we have refined the Ordering step to take advantage of the cycle structure of the graph [14]. In each component, we identify any maximal subtree that attaches to the remainder of the component through a single cut point. It is easy to verify that each edge of such a subtree can always be made to appear in a level of size 1. We identify these subtrees in linear time and place them at the end of the ordering. This improves the probability of success for the Searching step.

4.3 The Searching Step

The Searching step takes the levels produced in the Ordering step and tries to assign hash values to the keys, a level at a time. Assigning hash values to $K(v_i)$ amounts to assigning a value to $g(v_i)$, as is indicated in Section 3.3. To this end, we define a `hash-table` data structure

```
hash-table:  array [0..n − 1] of record
    key:  integer
    assigned:  boolean
    end
```

where `key` is the index to the key that has hash value i, and `assigned` is a flag as to whether the hash value i has been assigned to any key yet.

When a value is to be assigned to `vertex[i].g`, there are usually several choices for `vertex[i].g` that place all the keys in $K(v_i)$ into unassigned slots in hash-table. The analysis and the empirical results mentioned in section 3.3 indicate that an acceptable value for `vertex[i].g` should be picked *at random* rather than, for example, picking the *smallest* acceptable value for `vertex[i].g`. In looking for a value for `vertex[i].g`, the Searching step uses a random probe sequence to access the slots `0,...,n − 1` of the hash-table.

```
(1)   for i ∈ [0...n − 1] do
          hash-table[i].assigned = false
(2)   for i = 1 to t do
(3)       establish a random probe sequence s_0, s_1, ..., s_{n−1} for [0...n − 1]
          j = 0
          do
              collision = false
              if v_i ∈ [0...r − 1] then w = 1 else w = 2
              for each k ∈ K(v_i) do
(4)               h(k) = edge[k].h_0 + vertex[edge[k].h_{3−w}] + s_j   (mod n)
(5)                   if hash-table[h(k)].assigned then
                          collision = true
                              collision = true
(6)           if not collision then
                  for each k ∈ K(v_i) do
                      hash-table[h(k)].assigned = true
                      hash-table[h(k)].key = k
                  else
                  j = j + 1
(7)               if j > n − 1 then
                      fail
          while collision
```

Figure 5: The Searching Step

Figure 5 gives the algorithm for the Searching step. A random probe sequence of length n is chosen in step (3). The probe sequence actually used in our implementation has only a weak claim to randomness; that is, just a small amount of randomness is sufficient to make a good Searching step. At the beginning of the Searching step, the current implementation chooses a set of 20 small primes (or fewer if n is quite small) that do not divide n. Each time (3) is executed, one of the primes q is chosen at random to be s_1 and is used as an increment to obtain the remaining $s_j, j \geq 2$. Thus, the random probe sequence is

$$0, q, 2q, 3q, \ldots, (n-1)q.$$

A more robust random probe sequence would choose the increment q at random from $0, \ldots, n-1$ such that the greatest common division of q and n is 1. As just mentioned, such a robust sequence does not appear to be necessary.

A detail that is omitted from Figure 5 is the action taken when the Searching step is unable to insert a level into the hash table (**fail** in (7)). This is such a rare occurrence that for a large enough value of n and an appropriate choice of r, it is very unlikely to occur even once in the execution of the algorithm. Therefore, one reasonable response to this rare event is to restart

Word	h_0-value	h_1-value	h_2-value
Asgard	2	0	5
Ash	3	0	5
Ashanti	0	2	3
Ashcroft	3	1	5
Ashe	5	1	3
Asher	1	1	3

Table 2: Example: Set of Words with Associated h_0, h_1, h_2 Values

the algorithm from the beginning with new random tables for h_0, h_1, and h_2. Our current implementation actually uses a simple backtracking scheme. It assigns new g values to earlier vertices and then tries to complete the g function for the entire graph. More sophisticated backtracking schemes are possible, but, as mentioned above, it is unclear whether the added effort is justified.

5 Example

We illustrate our algorithm using a key set of 6 words that has actually been processed to yield a MPHF. The set of words was drawn from the initial portion of the *Collins English Dictionary* [21]. The 6 words with their h_0, h_1, and h_2 values are given in Table 2. The h_0, h_1, and h_2 functions are the auxiliary hash functions found in the Mapping step.

From this assignment of h_1 and h_2 values, the bipartite dependency graph shown in Figure 6 is produced. Note that some vertices (1, 3, and 5) are quite "popular" while vertex 4 is left out. Each word is associated with edges; there are two pairs of words, (Asgard, Ash) and (Ashe, Asher), that each have the same endpoints. This is allowed here, since the h_0 values will allow separation between words when the final hash value is computed.

The Ordering step finds an order for the vertices 0, 1, 2, 3, and 5 (those of degree > 0). The process of selecting the order is shown in Figure 7. In 7(a), an arbitrary vertex, 1, of maximum degree (3) is selected to be v_i; the result is $v_1 = 1$. Next, in 7(b), the vertices 3 and 5 adjacent to vertex 1 are examined to find one of maximum degree; the arbitrary selection of $v_2 = 5$ has been made. In 7(c), the vertices 0 and 3 (each adjacent to one of the vertices 1 and 5) are examined; the selection $v_3 = 3$ is made because vertex 3 has higher degree than vertex 0. In 7(d), vertices 0 and 2 are examined; the selection $v_4 = 0$ is made because vertex 0 has higher degree than vertex 2. Finally, in 7(e), the selection $v_5 = 2$ is made.

The result of the Ordering step is the vertex order 1, 5, 3, 0, 2. We obtain an ordering of 4 levels, shown in Table 3. Level i corresponds to the set of keys $K(v_{i+1})$ for vertex v_{i+1}. Thus, level 2 corresponds to the set of keys $K(v_3) = \{\text{Ashe, Asher}\}$. The level sizes are bounded above by the degree of the corresponding vertex and are typically smaller. For example, the degree of v_3 is 3 but $|K(v_3)| = 2$.

The Searching step assigns g values to the vertices 1, 5, 3, 0, and 2, in that order. The

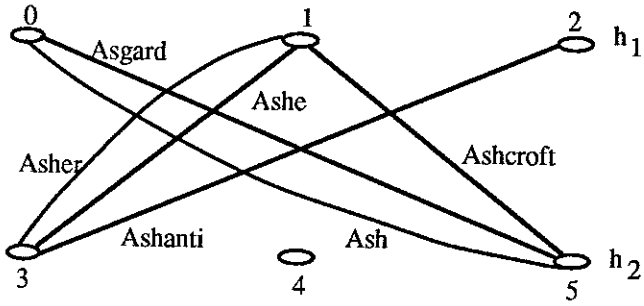
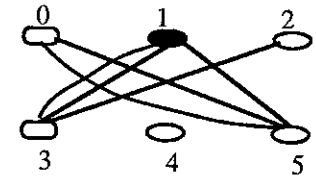
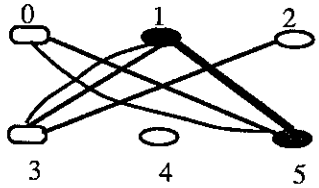


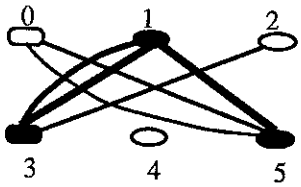
Figure 6: Example: Dependency Graph



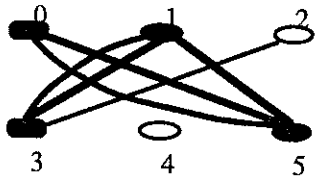
(a) Vertex selected: 1
Initialization



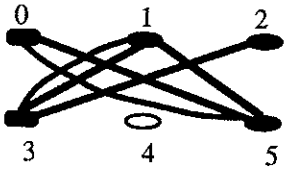
(b) Vertex selected: 5
Level: (1,5)



(c) Vertex selected: 3
Level: (1,3), (1,3)



(d) Vertex selected: 0
Level: (0,5), (0,5)



(e) Vertex selected: 2
Level: (2,3)

Figure 7: Example: Ordering Step

Level	Size of Level	Keys in This Level
1	1	Ashcroft
2	2	Ashe, Asher
3	2	Asgard, Ash
4	1	Ashanti

Table 3: Example: Levels in the ordering

assignment process is illustrated in Figure 8. The g value for $v_1 = 1$ is arbitrary; in 8(a), the assignment $g(v_1) = 2$ has been made. Vertex $v_2 = 5$ is next; $K(v_2) = \{\text{Ashcroft}\}$. We know

$$\begin{aligned} h_0(\text{Ashcroft}) &= 3 \\ g\big(h_1(\text{Ashcroft})\big) &= g(v_1) = 2. \end{aligned}$$

In 8(b), slot 1 has been selected for Ashcroft. Therefore,

$$\begin{aligned} g(v_2) &= h(\text{Ashcroft}) - h_0(\text{Ashcroft}) - g\big(h_1(\text{Ashcroft})\big) \pmod 6 \\ &= 1 - 3 - 2 \pmod 6 = 2. \end{aligned}$$

The next vertex is $v_3 = 3$; $K(v_3) = \{\text{Ashe}, \text{Asher}\}$. We calculate

$$\begin{aligned} b(\text{Ashe}) &= h_0(\text{Ashe}) + g\big(h_1(\text{Ashe})\big) \pmod 6 \\ &= 5 + 2 \pmod 6 = 1 \end{aligned}$$

and

$$b(\text{Asher}) = h_0(\text{Asher}) + g\big(h_1(\text{Asher})\big) \pmod 6 = 3.$$

Therefore, this level gives a pattern $\{1, 3\}$ to fit into the hash table. There are, of course, many values of $g(v_3)$ that make the pattern fit. In 8(c), the random value selected is $g(v_3) = 3$, which makes

$$h(\text{Ashe}) = 1 + 3 \pmod 6 = 4$$

and

$$h(\text{Asher}) = 3 + 3 \pmod 6 = 0.$$

The next vertex is $v_4 = 0$; $K(v_4) = \{\text{Ash}, \text{Asgard}\}$. The pattern for this level is $\{4, 5\}$. There is only one value for $g(v_4)$ that fits this pattern into the hash table. In 8(d), the value $g(v_4) = 4$ is selected, which fits the pattern in slots 2 and 3.

The last vertex is $v_5 = 2$; $K(v_5) = \{\text{Ashanti}\}$. Slot 5 is the only one remaining in the hash table. The selection $g(v_5) = 2$ is necessary to place Ashanti in slot 5.

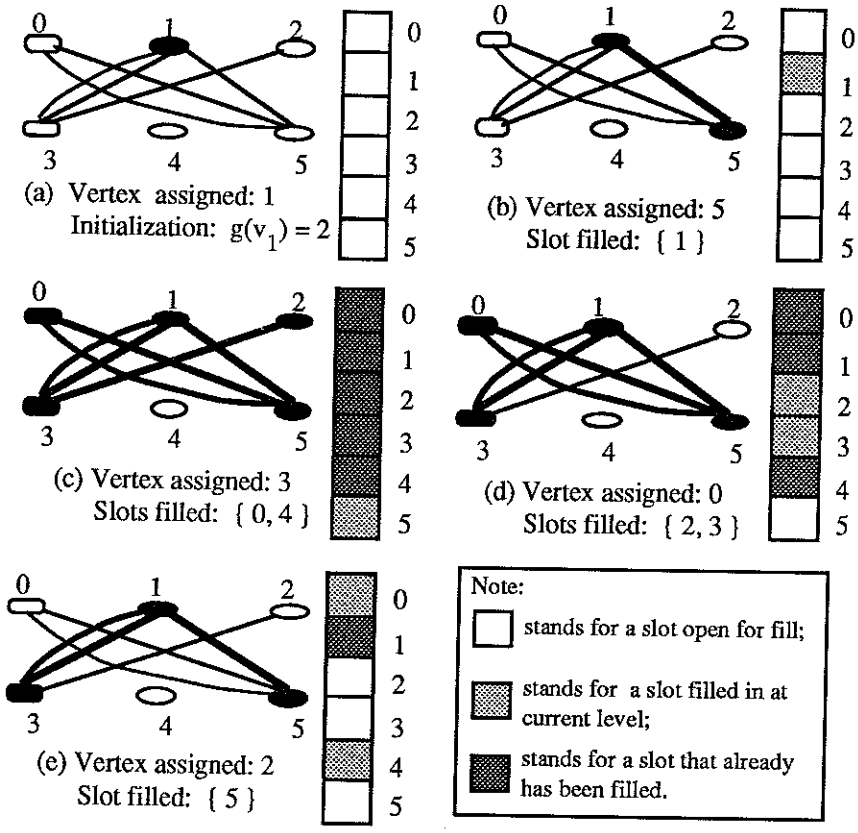


Figure 8: Example: Searching Step

Vertex	g Value
0	4
1	2
2	2
3	3
4	2
5	2

Table 4: Example: Vertices with Computed g Values

Keys	Hash Address
Asgard	2
Ash	3
Ashanti	5
Ashcroft	1
Ashe	4
Asher	0

Table 5: Example: Keys with Computed Hash Addresses

The selected g values are summarized in Table 4. Note that an arbitrary g value has been assigned to vertex 4, even though it was not in the sequence of ordered vertices, because it is of degree 0. In general, the Searching step assigns an arbitrary g value to each vertex of degree 0 so that g (and, hence, h) is a total function. No keys in the set S will actually access the g value of a vertex of degree 0.

The ultimate hash values for the six keys are given in Table 5. As required, the six hash values are distinct (we have a perfect hash function h), and the hash values are all less than 6 (h is a **minimal** perfect hash function).

6 Experimental Results

To support our claims regarding the theory and practice relating to our approach to MPHF determination, we collected distribution data regarding the vertex degrees in dependency graphs and level sizes in the ordering. For a detailed discussion of these data, refer to [16].

A variety of experimental tests have been made with versions of Algorithm 1, using ports to Apple Macintosh, IBM PC compatible, IBM PS/2, Cray XMP, and other systems. A comparison for Macintosh and Sequent Symmetry is given in [16]. For consistency, we focus below on timings using a Sequent Symmetry with 32 megabytes of main memory, running on a single (80386) processor running at about 4 MIPS, reporting UNIX "times()" results.

Table 6 shows timings for each phase of the algorithm, omitting only time required to double check that an MPHF has been found. Total time is given, along with two metrics for the size of the hash function, which we here fix by selecting $r = 0.3n$. First, since we claim less than $O(n)$ words/key is required, words/key is fixed at 0.6. Second, since $O(n)$ bits/key is the lower bound, we show bits/key. Note that key set sizes are given as successive powers of 2, from 2^5 through 2^{19}, so it is easy to see the time required is $O(n)$. By our selection, $O(n)$ words/key is required; we have also found that fewer words/key are needed as the key set size increases, suggesting $< O(n)$ words/key.

In summary, our algorithm processes large key sets well, and while there is some variation in processing time because of the probabilistic nature of the operations, with an appropriate value for r the algorithm finds a MPHF with high probability.

n	bits/key	Mapping	Ordering	Searching	Total
32	3.0	0.37	0.02	0.03	0.42
64	3.6	0.75	0.03	0.07	0.85
128	4.2	0.60	0.05	0.08	0.73
256	4.8	0.97	0.05	0.18	1.20
512	5.4	1.23	0.08	0.35	1.67
1024	6.0	1.50	0.17	0.67	2.33
2048	6.6	2.42	0.30	1.67	4.38
4096	7.2	3.47	0.62	3.13	7.22
8192	7.8	5.53	1.27	5.92	12.72
16384	8.4	9.87	2.52	12.05	24.43
32768	9.0	18.78	5.05	24.62	43.45
65536	9.6	35.68	10.20	50.02	95.90
131072	10.2	69.70	20.15	101.08	190.93
262144	10.8	137.97	40.30	201.57	379.83
524288	11.4	275.25	81.23	406.58	763.07

Note: for all runs,
Words/key = 0.6;
Number of Stacks = 12;
Machine = Sequent;
Time (CPU) is in seconds.

Table 6: Timing Results for Algorithm 1

6.1 Large Key Sets

The largest sets of keys we have been able to handle directly with our initial algorithm are a collection of over 420,878 French words and a set of $2^{19} = 524,288$ names taken from the Online Computer Library Center (OCLC) catalog, since our data structures have been tuned to require a maximum of $9n$ computer words. Our Sequent has sufficient primary memory for this to work well. For the French words, the graph used has $r = .25n$, so $|h|$ required 0.5 words/key. The Ordering step was modified to use 12 stacks, one for each vertex degree between 1 and 12. This modification led to a fast Ordering time of 53 seconds, while the time for the Searching step was 608 seconds. The total time, including the Mapping step, for our algorithm to find a MPHF for the 420,878 words was 812 seconds. For the OCLC keys, total time was 763 seconds.

For very large key sets, the primary limitation on the efficiency of our algorithm comes from the size of main memory. If little of the dependency graph can fit in the main memory of a virtual memory machine, then swapping occurs with a large proportion of the references to the dependency graph. This is because the graph is a random one and, therefore, violates the Principle of Locality.

To accommodate very large key sets, we have modified the implementation of the Mapping step to effectively partition the dependency graph into connected subgraphs of manageable size. With the modified implementation, a MPHF for a key set consisting of 1.2 million words was constructed on the Sequent Symmetry. The dependency graph was partitioned into 6 subgraphs of approximately equal size. In one run $r = .30n$, so $|h|$ required 0.60 words/key. The total time to construct the MPHF was 4804 seconds.

6.2 CD-ROM Versions

As mentioned earlier, one application for our MPHF method is to improve access time on CD-ROMs where records addressed by single keys are sought. We have prepared a demonstration of this for Virginia Disc One [13]. In particular, we took a 130,198 word collection and stored the g function and other parameters, along with the word strings corresponding to the hash table, on the CD-ROM. Users can ask for words in the *Collins English Dictionary* [21] or in files we have extracted from the AIList Digest (distributed over the Internet), the two sources for this set, and be told instantly what hash value has been assigned.

7 Algorithm 2

Algorithm 1, developed in 1989 [16] and summarized in sections 4-6 above, has $O(n)$ time complexity and less than $O(n)$ words space complexity. An extension of that algorithm, yielding order preserving minimal perfect hash functions, is discussed in [14]. This development led to further work in Spring 1990, leading to Algorithm 2, which has $O(n)$ time complexity and $O(n)$ bits space complexity [8]. An overview of this method is given below.

7.1 Basic Concepts

Algorithm 2 still applies the concepts explained in section 3, but adds several more insights. First, to reduce the size of the hash function to approach the theoretical lower bound, the domain of g is taken as $\{0,\ldots,r-1\}$ where $r = \lceil cn/log_2 n \rceil$, and c is a constant typically less than 4. The class of hash functions searched is:

$$h(k) = \begin{cases} \left(h_0(k)g(h_1(k)) + h_2(k)g^2(h_1(k))\right) mod\ n & \text{if } mark(h_1(k)) = 1 \\ g(h_1(k)) & \text{if } mark(h_1(k) = 0 \end{cases}$$

where

$$\begin{array}{rccl} h_0: & U & \rightarrow & \{0,\ldots,n-1\} \\ h_1: & U & \rightarrow & \{0,\ldots,r-1\} \\ h_2: & U & \rightarrow & \{0,\ldots,n-1\} \\ g: & \{0,\ldots,r-1\} & \rightarrow & \{0,\ldots,n-1\} \\ mark: & \{0,\ldots,r-1\} & \rightarrow & \{0,1\}. \end{array}$$

Here, h_0, h_1 and h_2 are random mapping functions as before, while g and $mark$ are parameters to be determined. Total space for these functions is $cn(1 + 1/log_2 n)$ bits, so space is $O(n)$ bits, typically $< 4n$ bits.

Second, the form of h is such that h_1 and h_2 play very different roles. This means that there is no dependency graph. Use of primary memory is greatly reduced, allowing easier MPHF constructions for very large key sets.

Third, quadratic hashing is utilized. This leads to the use of the mark bits, and requires an extra multiplication in cases where the mark bit is set. Yet the power of quadratic hashing allows smaller MPHFs to be found very quickly.

7.2 Overview of Steps

As in Algorithm 1, Mapping, Ordering, and Searching are required. This is no dependency graph, however, so Ordering and Searching simplified.

7.2.1 The Mapping Step

The Mapping step is like the procedure described in section 4.1 but h_0, h_1, and h_2 are taken modulo n, r, and n, respectively. The mark bits are set as part of this step, too.

7.2.2 The Ordering Step

Ordering involves two passes through the triplets $(h_0(k), h_1(k), h_2(k))$. In the first pass, h_1 values are used to build a one dimensional array A of linked lists labeled $0,\ldots,r-1$. The set of keys stored in linked list A_i, the i^{th} record of A, $0 \leq i \leq r-1$ is

$$K(A_i) = \{k | h_1(k) = i\}$$

The A array is

```
A: array [0..r-1] of record
    firstkey: integer;
    degree: integer;
    g: integer;
    mark: boolean;
    end
```

`firstkey` is the header for a singly-linked list of keys with $h_1(k) = i$. `degree` is the cardinality of $K(A_i)$. `g` is the g value for A_i, which is assigned later in the Searching step. `mark` is set if $|K(A_i)| > 1$ and is reset if $|K(A_i)| \leq 1$. More formally

$$mark(i) = \begin{cases} 1 & \text{if } |K(A_i)| > 1 \\ 0 & \text{if } |K(A_i)| \leq 1 \end{cases}$$

The `keys` array is

```
keys: array [1..n] of record
    h0, h1, h2: integer;
    nextkey: integer;
    end
```

where `nextkey` points to the next key in the linked list whose head is given by `firstkey` in the A array. In the second pass for Ordering, a number of stacks equal to the maximum cardinality, `MaxCard`, of the A_i are initialized. Each stack is assigned to store entries of a certain size. For example, Stack[1] is assigned to store entries of the A_i array of cardinality equal to 1, and so on. The A_i array is scanned and all elements are pushed into their corresponding stacks Stack[A_i.degree]. For very large key sets, these stacks are maintained on external storage.

7.2.3 The Searching Step

As explained in Section 3.3, the set of keys $k \in K(A_i)$ constitute a *pattern*. The Searching step determines an offset $g(i)$ that fits all elements of this pattern in empty slots of the hash table *simultaneously*. The Searching step pops entries of A_i from stacks and assigns hash values to all keys in $K(A_i)$, in descending order of A_i.degree. Thus, stacks for larger patterns are processed before stacks with smaller patterns. Since $h(k) = h_0(k)g(i) + h_2(k)g^2(i)$, $i = h_1(k)$, assigning hash values to all keys in $K(A_i)$ is done by assigning a value to $g(i)$. This $h(k)$ is a variation of quadratic hashing and has the advantage of eliminating both elementary and secondary clustering in the hash table. When all patterns of size greater than 1 are processed, assigning the rest of the patterns of size 1 in A is done by setting g to the addresses of the remaining empty slots in the hash table using the hash function $h(k) = g(i)$. The $mark(h_1(k))$ function is used to distinguish which hash function has been used to hash k.

7.3 Experimental Results

In Table 7, we exhibit the strengths of Algorithm 2 in two different ways. The second and third columns give timing results for our two algorithms under a uniform space assumption of 0.6

n	Algorithm 1 Totals (words/key =0.6)	Algorithm 2 Totals (words/key =0.6)	Near lowest bits/key achieved by Algorithm 2				
			bits/key	Mapping	Ordering	Searching	Total
32	0.10	2.18	2.28	2.15	0.02	0.02	2.19
1024	1.33	2.95	3.19	2.58	0.05	1.13	3.76
131072	189.28	98.93	3.24	58.18	7.43	14569.12	14634.73
262144	374.09	194.43	3.61	117.63	15.82	4606.75	4740.20
524288	808.34	383.57	3.60	228.10	34.43	14129.87	14831.73

Note: for all runs,
Machine = Sequent;
Time (CPU) is in seconds.

Table 7: Timing Results for Algorithms 1 and 2 using internal memory

words/key. While Algorithm 1 is faster for small key sets, Algorithm 2 proves itself significantly faster for the large key sets. The last 5 columns show the results of pushing Algorithm 2 to require as few bits as possible. The experiments indicated by these rows were conducted as follows. Lower and lower bits/key values were tried until Algorithm 2 failed to find an MPHF consistently. The time for those runs are reported in Table 7. The run for $n = 262144$ is anomalous, indicating that bits/key can actually be pushed lower.

Algorithm 2 is also capable of running efficiently using external storage when there is not sufficient internal storage. Timing results for some very large runs using external storage are reported in Table 8. Note that the dominant contribution to time is the Searching step. This time can be decreased dramatically by a modest increase in the bits/key values (see Figure 10).

Figure 9 illustrates the linear time complexity, giving total time (generally dominated by Searching) for various set sizes. Figure 10 illustrates that few bits/key are required, regardless of set size, but that time to find a MPHF increases rapidly as the theoretical lower bound on space is approached.

8 Applications

As mentioned at the beginning of this paper, the most exciting aspect of this work is the wide range of applications expected for the MPHF scheme. There is utility for MPHFs in regard to hypertext, hypermedia, semantic networks, file managers, database management systems, object managers, information retrieval systems, compilers, etc.

We have begun to exploit this potential in a number of areas. One, related to our work of building a large lexicon from machine readable dictionaries, is to construct a general dictionary manager able to handle large keys set with associated fixed or variable length records. Our MPHF system has been suitably embedded into a dictionary manager that can support a variety of functions as can be seen in Figure 11. A version of this is used in connection with an X.500

n	Near lowest bits/key achieved by Algorithm 2				
	bits/key	Mapping	Ordering	Searching	Total
524288	3.59	447.47	597.25	11476.13	12520.85
1200502	3.60	1060.62	1432.48	48118.00	50611.10
3875766	4.58	2114.15	2955.60	28243.25	33313.00

Note: for all runs except for n=3,875,766,
Machine = Sequent;
for n=3,875,766,
Machine=NeXt
Time (CPU) is in seconds.

Table 8: Timing Results for Algorithms 1 and 2 using external storage

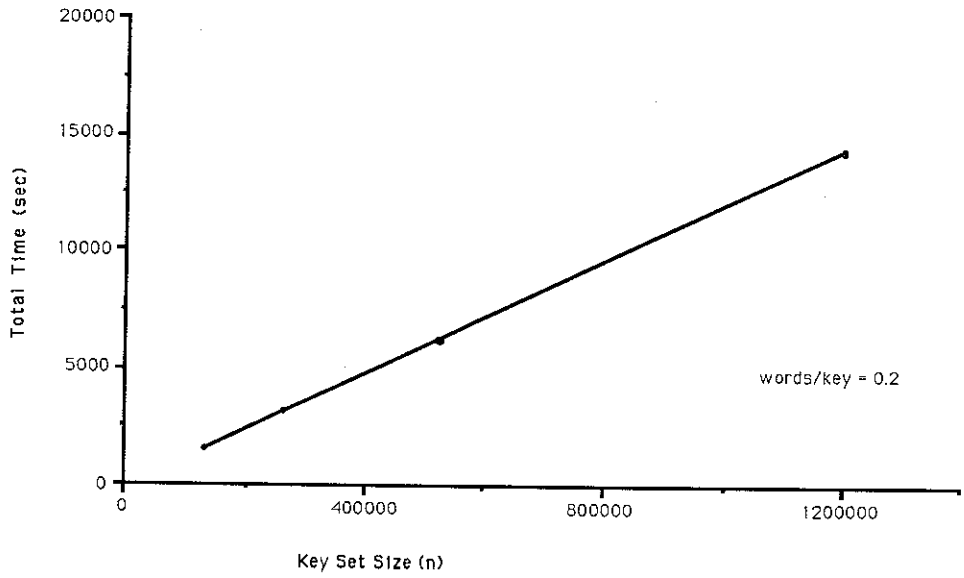


Figure 9: Total Time vs. Key Set Size

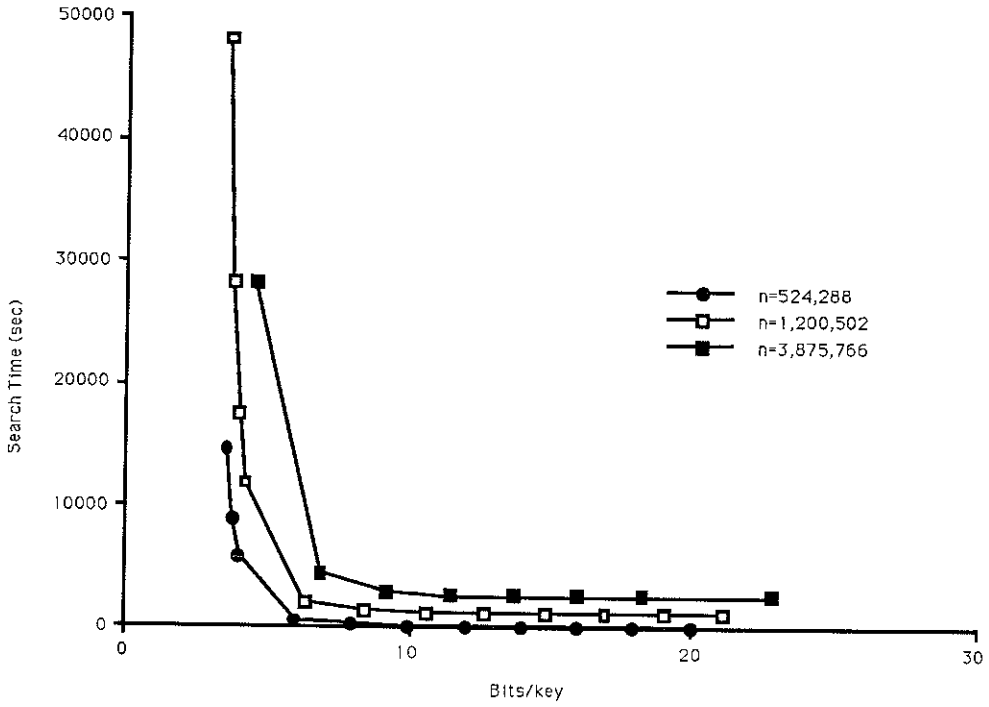


Figure 10: Search Time vs. Bits/key

directory server implemented at the VPI&SU Computing Center.

We have integrated MPHF code with a tree manager so that large data collections that are relatively static can be slowly updated or extended; when a sufficient number of changes (which are recorded in the tree) have been made the system automatically empties the tree and builds a new MPHF for the current data.

As work continues on our Large External Network Database (LEND), we expect to be able to apply our MPHF scheme to support a variety of applications involving large semantic networks, hypertext, hypermedia, and other large object bases. We believe that significant benefit will result in connection with CD-ROM, optical disc, magnetic disk, and even primary memory based retrieval.

9 Conclusion

This paper gives theoretical and experimental validation of practical new algorithms for finding minimal perfect hash functions, suitable for key sets ranging in size from small to very large (over a million). Recent efforts have led to versions requiring time and space close to theoretical lower bounds, and to methods to identify order preserving minimal perfect hash functions. These algorithms are in use for a variety of applications, and should be of value in many situations where single access is required for static key collections.

1. Define tables of n-fields. Each field may be an integer, a string of maximum length less than N, or a string of length without limit. MPHF indexing on several fields may be specified.
2. Load each table with data. MPHF indexing is performed on fields defined to be MPHF indexed. Duplication on field values can be handled.
3. Look up rows that have values for MPHF indexed fields that are equal to query terms.
4. Retrieve a field value given row number and field name.
5. Retrieve a whole row of a table.
6. Delete/update a field value of a row.

Figure 11: Functions Supported by Dictionary Manager

10 Acknowledgments

We are grateful to Collins Publishers for allowing us to work with the machine readable copy of the *Collins English Dictionary.* Professor Abraham Bookstein arranged for us to obtain the large French word list in use at the ARTFL Project at the University of Chicago. Dr. Martin Dillon of OCLC in Dublin, Ohio, arranged for us to obtain the 1.2 million key file from their catalog records. Nimbus Records has pressed various versions of Virginia Disc One that demonstrate our algorithm running on a PC with attached CD-ROM drive.

References

[1] Brain, M.D., and Tharp, A.L. Near-perfect hashing of large word sets. *Software – Practice and Experience 19* (1989), 967-978.

[2] Carter, J.L., and Wegman, M.N. Universal classes of hash functions. *Journal of Computer and System Sciences 18* (1979), 143-154.

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

[4] Chang, C.C. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM 27* (1984), 384-387.

[5] Chang, C.C. Letter oriented reciprocal hashing scheme. *Information Sciences 38* (1986), 243-255.

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

[7] Cormack, G.V., Horspool, R.N.S., and Kaiserswerth, M. Practical perfect hashing. *The Computer Journal 28* (1985), 54-58.

[8] Daoud, A.M. Efficient data structures for information retrieval systems. Dissertation proposal, Department of Computer Science, Virginia Polytechnic Institute & State University, July, 1990.

[9] Datta, S. Implementation of a perfect hash function scheme. Master's Report, Department of Computer Science, Virginia Polytechnic Institute & State University, 1988. Available as Technical Report TR-89-9.

[10] Enbody, R.J., and Du, H.C. Dynamic hashing schemes. *ACM Computing Surveys 20* (1988), 85-113.

[11] Feller, W. *An Introduction to Probability Theory and its Applications, Volume 1.* John Wiley and Sons, New York, 1968.

[12] Fox, E.A. Optical disks and CD-ROM: publishing and access. In *Annual Review of Information Science and Technology,* Martha E. Williams, ed. ASIS/Elsevier Science Publishers B. V., Amsterdam, Vol. 23, 1988, 85-124.

[13] Fox, E.A. Virginia Disc One. CD-ROM developed at Virginia Polytechnic Institute & State University and produced by Nimbus Records, Ruckersville, VA, 1990.

[14] Fox, E.A., Chen, Q., Daoud, A.M. and Heath, L. Finding order preserving minimal perfect hash functions in average linear time and applying them to information retrieval, SIGIR '90, 1990.

[15] Fox, E.A., Chen, Q., Heath, L. and Datta, S. A more cost effective algorithm for finding perfect hash functions. In *Proceedings of the Seventeenth Annual ACM Computer Science Conference,* Louisville, KY, 1989, 114-122.

[16] Fox, E.A., Chen, Q., and Heath, L. An $O(n \log n)$ algorithm for finding minimal perfect hash functions. TR 89-10, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA, 1989.

[17] Fox, E.A., Nutter, J.T., Ahlswede, T., Evens, M. and Markowitz, J. Building a large thesaurus for information retrieval. In *Proceedings of the Second Conference on Applied Natural Language Processing,* Austin, TX, 1988, 101-108.

[18] Fox, E., Wohlwend, R., Sheldon, P., Chen, Q. and France, R. Building the CODER lexicon: the Collins English Dictionary and its adverb definitions. Technical Report TR-86-23, Department of Computer Science, Virginia Polytechnic Institute & State University, 1986.

[19] Fredman, M.L., Komlós, J. and Szemerédi, E. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM 31* (1984), 538-544.

[20] Gonnet, G.L. and Larson, P. External hashing with limited internal storage. *Journal of the ACM* 35 (1988), pp. 161-184.

[21] Hanks, P., editor. *Collins English Dictionary.* William Collins Sons & Co., London, 1979.

[22] Jaeschke, G. Reciprocal hashing—a method for generating minimal perfect hash functions. *Communications of the ACM 24* (1981), 829-833.

[23] Knuth, D.E. *The Art of Computer Programming,* Volume 3, *Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[24] Mairson, H.G. The program complexity of searching a table, *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science,* 1983, 40-47.

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

[26] Palmer, E.M. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[27] Park, S.K. and Miller, K.W. Random number generators: good ones are hard to find. *Communications of the ACM 31* (1988), 1192-1201.

[28] Pearson, P.K. Fast hashing of variable-length text strings. *Communications of the ACM 33* (1990), 677-680.

[29] Ramakrishna, M.V. and Larson, P. File organization using composite perfect hashing. *ACM Transactions on Database Systems 14* (1989), 231-263.

[30] Sager, T.J. A new method for generating minimal perfect hashing functions. Technical Report CSc-84-15, Department of Computer Science, University of Missouri-Rolla, Missouri, 1984.

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

[31] Schmidt, J.P., and Siegel, A. On aspects of universality and performance for closed hashing. *Proceedings of the 21st ACM Symposium on Theory of Computing,* 1989, 355-366.

[33] Sedgewick, R. *Algorithms,* Addison-Wesley Publishing Company, Reading, MA, 1988.

[32] Sprugnoli, R. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM 20* (1978), 841-850.