An $O(n \log n)$ Algorithm for Finding Minimal Perfect Hash Functions

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

TR 89-10

An $O(n \log n)$ Algorithm for Finding Minimal Perfect Hash Functions *

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

June 21, 1989

Abstract

We describe the first practical algorithm for finding minimal perfect hash functions that can be used to access very large databases. 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 new $O(n \log n)$ 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. While ultimately applicable to a wide variety of data and file access needs, this algorithm has 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, and several other microcomputer, minicomputer, and parallel processor versions and applications of our algorithm have also been developed. Tests with a French word list of 420,878 entries and a library catalog key set with over 1.2 million keys have shown that our methods work with very large databases.

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

*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, and the State Council of Higher Education. AT&T and Apple Computer have provided equipment used in some of our experiments.

1 Introduction

Ubiquitous in the 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 a fast algorithm to search for MPHF's for large static key sets, and have used the resulting functions to improve access to large CD-ROM (compact disk, read-only) units 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 [FOX88]. 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 [FOX89] and in constructing a large lexicon from machine readable dictionaries [FWSCF86], [FNAEM88] 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 considering 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, the optimal case regarding space is 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; i.e., for h to be 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 fast (i.e., $O(n \log n)$ expected time) algorithm for finding MPHF's, and Section 5 illustrates the procedure using a very small but realistic example that is worked out in a fair amount of detail. Section 6 reports on some of our experimental results, giving a characterization of the internal representations required during MPHF construction, timings on several types of computers 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 efforts to use MPHFs in connection with our lexicon construction effort, design of a Large External Network Database (LEND) that involves MPHF access at the lowest level, and other applications.

2 Related Work

Hashing has been a topic of study for many years, both in regard to practical methods and analytical investigations [KNU73]. Recently there has been renewed interest in hashing due to the development of techniques suitable for dynamic collections [ED88]. 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. We consider this work in subsequent subsections,

as we unfold the various dimensions of the problem.

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 assumed to be positive integers bounded above by N. This situation was assumed, for example, by Sprugnoli [SPR78] and Jaeschke [JAE81].

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 [CHA84] used four tables based on the first and second letters of the key. Cichelli [CIC80] 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. [CKB83] 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., [CIC80]) 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 Proof

One might ask if a MPHF h for a given key set exists. Jaeschke [JAE81] 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 can similarly demonstrate that the answer is clearly yes. A constructive proof follows in terms of a sketch of a fast algorithm to define a suitable (though large!) MPHF:

Allocate an array A of length N with all values initialized to ERROR.
for $i = 1$ to n **do**
    $A[k_i] = i - 1$

The array A then defines h, where allowable keys map into hash addresses $\{0, \ldots, n-1\}$ and other keys yield ERROR.

While this indeed gives us a proper h, since $N \gg m$, the array A is mostly empty (ERROR), and this hash function is too large to be really useful. Another way to view this is to realize that perfect hash functions are rare in the set of all functions. Knuth [KNU73] 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

In the algorithm given above, A is an array of N numbers, each representable in $\log_2 m$ bits. This gives an upper bound of $O(N \log_2 m)$ bits to store the PHF. Mehlhorn shows a lower bound on the number of functions needed to make a family of PHFs:

$$\frac{\binom{N}{n}}{(N/m)^n \binom{m}{n}}.$$

Under the reasonable assumption that N grows faster than n^2, we can use the asymptotic estimate $\binom{N}{n} \sim N^n/n!$ to obtain the asymptotic lower bound

$$\frac{m^n}{n!\binom{m}{n}}.$$

In the case of minimal perfect hashing, $m = n$, so we have the asymptotic lower bound $n^n/n!$. Applying Stirling's approximation for $n!$ and taking the base 2 logarithm, we have an approximate, asymptotic lower bound of

$$\begin{aligned} \log_2 e^n/\sqrt{2\pi n} &= n \log_2 e - \log_2 \sqrt{2\pi n} \\ &\approx 1.4427n \end{aligned}$$

bits to represent an arbitrary PHF.

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 algorithm achieves MPHF size of less than $O(n)$ computer words, that is, the storage required for a MPHF increases slightly less than linearly with n.

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 [CW79] 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

Figure 1: Method to Find Perfect Hash Functions

can be found quickly. This technique has also been applied to dynamic hashing by Ramakrishna and Larson [RL89].

Sprugnoli [SPR78] 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 [JAE81] suggests a reciprocal hashing scheme with three parameters that is guaranteed to find a MPHF, but that is only practical when $n \leq 20$. Chang [CHA86] 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.

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 [CIC80] and of Cercone et al. [CKB83] 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.

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 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

Sager [SAG84,SAG85] 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

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

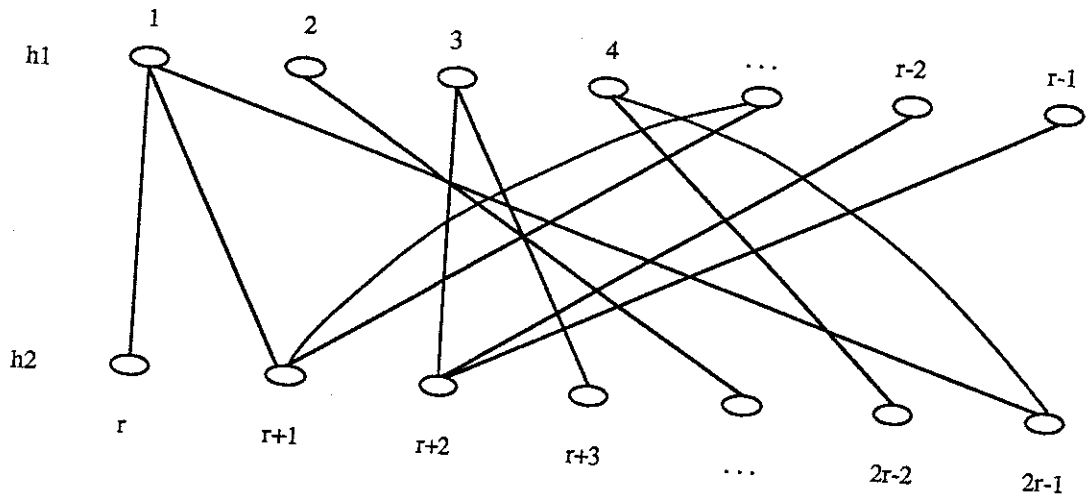


Figure 2: Dependency Graph

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 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. Sager calls the ordering of the keys into a sequence of levels a *tower*. We shall also use the term tower to mean a sequence of levels.

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.

2.6 Improvement on Sager's Method

While Sager's approach is quite flexible, use of the expensive mincycle heuristic in the Ordering step restricts the utility of his scheme to moderate size sets. 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 [DAT88]. We determined that Sager's approach did indeed require $O(m^4)$ time and also found that we could find MPHF's with $r \leq m/2$. We defined $ratio = 2r/m$ to measure the relative size of the dependency graph to the size of the key set.

After further investigation we developed a modified algorithm requiring $O(m^3)$ time that is described in [FCHD89]. In the Ordering step, mincycle is replaced by a faster two step process. First, we build a maximum spanning forest for the bipartite graph using Prim's algorithm, where the edge multiplicities serve as edge weights. Second, we use several heuristics to obtain lists ordered by edge multiplicity, number of cycles, or number of edges between subgraphs.

With this algorithm we were able to find MPHF's for sets of over a thousand words, and to prepare a beta copy of the Virginia Disc One CD-ROM [FOX89] with 273 MPHFs computed on that many sets of 256 words each.

2.7 Summary of Related Work

As mentioned above, hashing has been used in many applications and, with recent development of dynamic hashing techniques, has witnessed a resurgence of interest (Enbody and Du [ED88]). However, most dynamic hashing involves low load values, on the order of 0.60 to 0.90, 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.

An examination of previous schemes for perfect hashing shows that they are generally only applicable to very small sets, or require a prohibitive amount of space to store. Thus, it may be the case that a MPHF cannot realistically be found for a large set. 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 (e.g., Fredman, Komlós and Szemerédi [FKS84] and work that builds on their approach).

When we consider these various methods, we should evaluate them according to a set of reasonable criteria. Table 1 lists a collection of criteria to evaluate MPHF approaches, where those which lead to positive responses for all questions are most desirable. In the next sections we discuss our algorithm, that possesses each of these good qualities.

3 Key Concepts of New Algorithm

After careful analysis of Sager's algorithm [SAG85] and our enhanced version [FCHD89], Heath made three crucial observations that serve as the foundation of our new algorithm:

1. Can the method work directly with character strings?
2. Can the method work with a single record per bin?
3. Can the method work without extra storage in bins?
4. Can the method work on any large set of keys?
5. Can a load factor of 1 be achieved?
6. Will a hash function be found almost always?
7. Can the hash function be found very quickly?
8. Can the hash function description be stored without taking up too much space?
9. Can the address of a record be found very quickly? in one step?
10. Can the address of a record be found without using additional data storage?

Table 1: Considerations when Evaluating Perfect Hash Function Methods

1. First, it appears that we must exploit randomness whenever possible.

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 algorithm to obtain an Ordering in $O(n \log n)$ expected time. Since Mapping and Searching have expected time $O(n)$, the time complexity of our method is $O(n \log n)$ 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. The functions suggested by Sager do not yield distinct triples when used with large key sets.

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 idea is to produce a random number for each letter/position combination in the keys, and to make use of the fact that the sum of random numbers is a random number.

The pseudo-random number generator we selected, to allow machine independence and to ensure good behavior (better than the standard routine in the C library!), is due to Park and Miller [PM88]. The random integers generated by their random number generator are between 0 and $2^{31} - 2$. The next random integer *Seed* is gotten by the formula:

$$Seed = (16,807 \times Seed) \ \ mod \ \ 2,147,483,647.$$

To obtain the random integers needed for our purposes, we take $Seed \pmod n$. 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 word. Let a key be the character string $k = k_1 k_2 \ldots k_y$ (y, the length of k, is a function of k). Then, the triple is computed using the following formulas:

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

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

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

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. Certainly, that probability must be close to 1 if we are to build a MPHF for a large key set. The derivation below demonstrates that the probability of no collisions tends to one rapidly (the probability of collision tends to zero rapidly) as key set size increases.

Let $t = nr^2$ be the size of the universe of triples. The probability that n triples chosen uniformly at random from t triples are distinct is

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

The number of computer words required to represent g is $2r$. From the lower bound on the size of g,

$$\begin{aligned} r &= \Omega(n/\log_2 n) \\ t &= nr^2 = \Omega(n^3/\log_2^2 n). \end{aligned}$$

By an asymptotic estimate from Palmer [PAL85],

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

Therefore,

$$\begin{aligned} -\frac{n^2}{2t} - \frac{n^3}{6t^2} &= O\left(\frac{\log_2^2 n}{n} \right) \\ p(n,t) &\sim 1. \end{aligned}$$

Set Size (n)	Probability of Collision, ratio=1.0	Probability of Collision, ratio=0.4
16	0.111111	0.530777
32	0.058824	0.317154
64	0.030303	0.175229
128	0.015385	0.092392
256	0.007752	0.047479
512	0.003891	0.024073
1024	0.001949	0.012121
2048	0.000976	0.006082
4096	0.000488	0.003046
8192	0.000244	0.001525
16384	0.000122	0.000763
32768	0.000061	0.000381
65536	0.000031	0.000191
130198	0.000015	0.000096

Table 2: Probability of Collision for h_0, h_1, h_2 Triples

For typical values of $r = cn$, where c is a constant,

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

so that $p(n,t)$ goes to 1 quite rapidly with n.

We can illustrate how rapidly the probability of collision, $1-p(n,t)$, decreases by considering Table 2, where set sizes from 16 through over 130,000 are considered. Clearly it is unlikely that we will have collisions in triples for large key sets, even when a small graph (e.g., $ratio = 0.4$) is involved. Note too that if by some chance an assignment of triples leads to a collision, then it is extremely unlikely that with new random tables there would be another collision. Thus, our method for finding a suitable set of h_0, h_1, h_2 functions is linear in an expected sense.

Later we will see that randomness plays a key role in other parts of the algorithm. In the next section, we discuss the distribution of vertex degrees induced by these random functions.

In the following section, we show that randomness is important in the Searching step.

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 tower.

Concentrate on a particular vertex v and the edges incident on it. The probability that a particular edge is incident on v is $p = 1/r$. Let X be the random variable that equals the degree of v. Then (Feller [FEL68]), X is binomially distributed with parameters n and p. Therefore, the mean is

$$\mathrm{E}(X) = np$$

and the variance is

$$\begin{aligned} \mathrm{Var}(X) &= np(1-p) \\ &\approx np \end{aligned}$$

since $1 - p \approx 1$. Since the case of large n is the case 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, a good approximation to the distribution of vertex degrees can be obtained. The expected number of vertices of degree d is

$$2r\Pr(X = d) \approx \frac{2r2^{-n/r}\left(\frac{n}{r}\right)^d}{d!}$$

For $ratio = 1.0$, 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. From the Poisson approximation, we see that there will be few vertices of high degree. These predictions are actually born out by experimentation; see Section 6 for more details.

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*. From an ordering of vertices, it is a simple matter to obtain an ordering of the keys into levels. 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, of course, 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 $K(v_i)$, where each nonempty $K(v_i)$ is a level of the ordering.

As discussed in the next section, it is desirable to have levels that are as small as possible and to have any large levels early in the ordering. If vertex ordering is used, 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

Our third key insight is suggested by the vertex degree distribution. If the Ordering step processes vertices with higher degree first, then the tower contains the larger groups of inter-related words early on, thereby avoiding these troublesome cases later on. In the Searching step, this translates into placing larger groups into the hash table first, and handling smaller groups later.

To view this from another perspective, consider Figure 3. On the right is a hash table that has been partially filled. This corresponds to a partially filled disk of the same size (on the left), since the h function involves placement mod n. At each iteration of the Searching step, one level of the tower 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 $v_i \in \{r, \ldots, 2r-1\}$, that is, v_i is on the h_2 side of the dependency graph. 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

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

Then

$$h(k) = b(k) + g(v_i).$$

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). Since the pattern is (mod n), it may be viewed as circular and subject to *rotation* by the amount $g(v_i)$. 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,

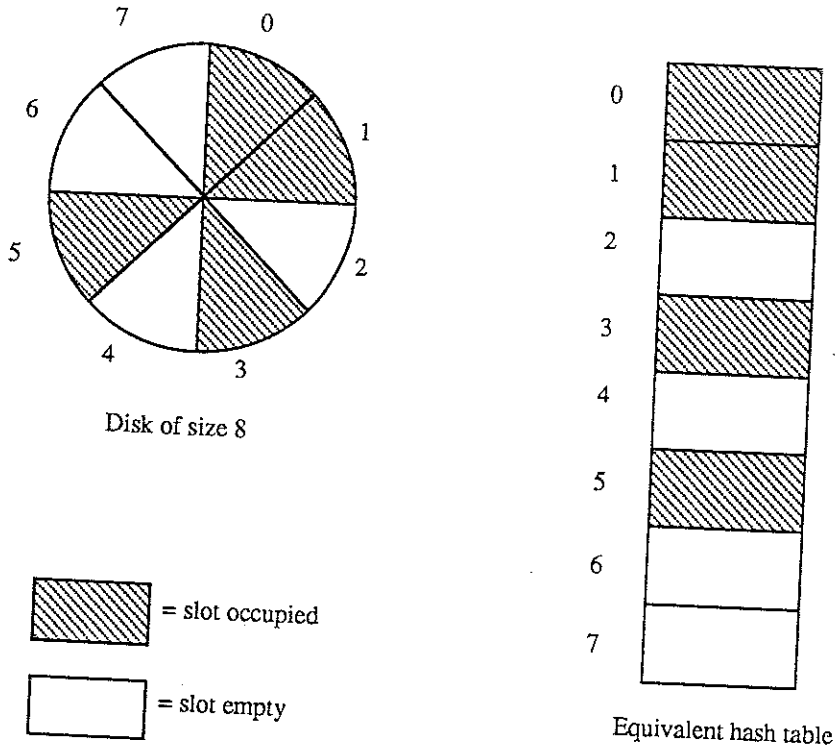


Figure 3: Typical Layout of a Partially Filled Hash Table

regardless of the size of j it is 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.

Consider the situation of Figure 3 and the problem of fitting various patterns into the disk. In Table 3 we see the various possible patterns, classified in terms of the skip distance between related entries and then into good or bad patterns, depending upon whether they can be placed in the disk. For this example with small n and with a largely full disk, even when only pairs of vertices are left to consider, fitting the remaining entries into the disk does not have a high probability of success.

It is important to show that because of the distribution of vertex degrees, we can expect to fit all of the keys into the hash table. We claim that if we have a *ratio* sufficiently close to 1.0, enter words into our hash table roughly according to a descending ordering of the degree of vertices in the graph, and repeatedly rotate the pattern corresponding to a word group till a fit in the disk is achieved, then a MPHF will almost certainly be found.

3.3.1 Probability of Fitting into Disk

Thus, we now consider the probability of fitting a pattern into a disk. (The notation introduced in the remainder of this section is restricted to this section.) Model this by a disk of M slots, f

Possible Patterns (i = [0,...,7])	Skip Distance	Good Patterns	Bad Patterns
{(i, (i+1) mod 8)}	1	{(6,7)}	{(0,1), (1,2), (2,3), (3,4), (4,5), (5,6), (7,0)}
{(i, (i+2) mod 8)}	2	{(2,4), (4,6)}	{(0,2), (1,3), (3,5), (5,7), (6,0), (7,1)}
{(i, (i+3) mod 8)}	3	{(4,7), (7,2)}	{(0,3), (1,4), (2,5), (3,6), (5,0), (6,1)}
{(i, (i+4) mod 8)}	4	{(2,6)}	{(0,4), (1,5), 3,7)}

Note: A pattern in this case is defined as an integer pair (x,y). Fitting into the hash table requires both slots x and y to be empty. If they are not, the next slot pair ((x+1) mod 8, (y+1) mod 8) is tried. If eventually both are empty, slots x and y are filled.

Table 3: Pattern Classification for Filling Disk

of which are filled, and a pattern of size j. The problem of fitting the pattern into the table is to find a rotation of the pattern that fits into the disk without hitting any occupied slots. As $M \gg j$, assume j is a fixed small constant and that the pattern is any fixed pattern of size j. Also assume that the f occupied slots are chosen uniformly at random from the $\binom{M}{f}$ possible patterns of occupied slots.

We wish to estimate the probability, as $M \to \infty$, that the pattern of size j *fails* to fit. There are M rotations of the pattern. Let $X_i, 0 \le i \le M-1$, be the random variable that is 1 if the ith rotation of the pattern fits and 0 otherwise. Define the random variable

$$X = \sum_{i=0}^{M-1} X_i$$

to be the number of rotations for which the pattern fits. The probability that the pattern does not fit under any rotation is $\Pr(X = 0)$. Following Feller [FEL68], Chapter 4, define the Rth binomial moment, $0 \le R \le M$, as

$$S_R = \sum_{0 \le i_1 \le i_2 \le \cdots \le i_R \le M-1} \mathrm{E}(X_{i_1} X_{i_2} \cdots X_{i_R})$$

where the sum is over all $\binom{M}{R}$ subsets $\{i_1, i_2, \ldots, i_R\}$ of size R of the set $\{0, 1, \ldots, M-1\}$. $\mathrm{E}(X_{i_1} X_{i_2} \cdots X_{i_R})$ is the same as the probability that the pattern fits under *all* the R rotations $i_1, i_2, \ldots, i_R$. Fix R and $i_1, i_2, \ldots, i_R$ for the moment. Suppose the R rotations of the pattern hit Rj distinct slots (that is, no two rotations hit the same slot). Then

$$\mathrm{E}(X_{i_1} X_{i_2} \cdots X_{i_R}) = \binom{M-Rj}{f} \Big/ \binom{M}{f}.$$

As $M \to \infty$, the probability that the R rotations of the pattern collide (that is, hit fewer than Rj slots) goes to 0. Therefore,

$$\begin{aligned}
S_R &\to \binom{M}{R}\binom{M-Rj}{f} \Big/ \binom{M}{f} \\
&= \frac{M!(M-Rj)!f!(M-f)!}{R!(M-R)!f!(M-Rj-f)!M!} \\
&= \frac{1}{R!} \cdot \frac{(M-f)(M-f-1)\cdots(M-Rj-f+1)}{(M-R)(M-R-1)\cdots(M-Rj+1)} \\
&= \frac{1}{R!} \prod_{i=0}^{Rj-R-1} \left(\frac{M-f-i}{M-R-i}\right) \prod_{i=Rj-R}^{Rj-1} (M-f-i) \\
&= \frac{1}{R!} \prod_{i=1}^{Rj-R-1} \left(1 - \frac{f-R}{M-R-i}\right) \prod_{i=0}^{R-1} (M-f-i-Rj+R)
\end{aligned}$$

$$\rightarrow \quad \frac{1}{R!}\left(1-\frac{f}{M}\right)^{(j-1)R}(M-f)^R$$

when $M \rightarrow \infty$ and $M - f \rightarrow \infty$. By the Principle of Inclusion and Exclusion,

$$\begin{aligned}
\Pr(X=0) &= \sum_{R=0}^{M}(-1)^R S_R \\
&\rightarrow \sum_{R=0}^{M}(-1)^R \frac{1}{R!}\left(1-\frac{f}{M}\right)^{(j-1)R}(M-f)^R \\
&\rightarrow \sum_{R=0}^{\infty}(-1)^R \frac{1}{R!}\left(\left(1-\frac{f}{M}\right)^{j-1}(M-f)\right)^R \\
&= e^{-\mu}
\end{aligned}$$

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 $\mu \rightarrow \infty$ and $\Pr(X=0) \rightarrow 0$. 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.

3.3.2 Simulations of Fitting into Disk

To clarify the process of filling a disk, we have performed simulations on various size sets of keys with various values of *ratio*. The purpose of the simulations is to estimate the probability of fitting the set of keys into the slots of the disk (hash table). Each set of keys was processed by the Mapping and Ordering steps of our algorithm to produce a tower of levels. The simulation program took the tower and calculated the probability of fitting a tower with that sequence of levels into the disk for each level in the tower. All patterns of the size of that level are considered in the calculation. When the size of the disk makes the number of patterns prohibitively large, the program takes a random sample of 200 patterns to make the calculation. Two different strategies, sequential and random probing, for placing a level in the disk were explored.

Table 4 gives the probabilities for sequential probing, the method that corresponds to Sager's Searching step. Table 5 gives the probabilities for random probing, the method used in our Searching step. In all cases, random probing is superior. In fact, Table 5 indicates that random probing is likely to be highly successful for moderately large to very large sets of keys.

4 Algorithm Outline

Our algorithm for finding MPHFs for large key sets is an extension of earlier work by Sager [SAG85] and built 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

n \ Ratio	1.0	0.7	0.5
32	0.635556	0.030341	0.038892
64	0.121471	0.004376	0.000037
128	0.064129	0.000192	$\approx$ 0.0
256	0.001330	$\approx$ 0.0	$\approx$ 0.0
512	0.007108	$\approx$ 0.0	$\approx$ 0.0
1024	1.0	0.000193	$\approx$ 0.0
2048	1.0	1.0	$\approx$ 0.0
4096	1.0	1.0	0.002559
8192	1.0	1.0	1.0

Table 4: Simulation of Filling Disk - Estimates of Probability of Success - Sequential Probing

n \ Ratio	1.0	0.7	0.5
32	1.0	0.426667	0.396231
64	0.967742	0.784599	0.014669
128	1.0	0.984127	0.001169
256	1.0	0.945746	0.103897
512	1.0	1.0	0.058072
1024	1.0	1.0	0.413002
2048	1.0	1.0	0.950918
4096	1.0	1.0	0.932089
8192	1.0	1.0	1.0

Table 5: Simulation of Filling Disk - Estimates of Probability of Success - Random Probing

$U =$	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 table, with slots numbered $0, \ldots, (m-1)$
$m =$	number of slots in T
$h =$	function to map key k into hash table T
$\|h\| =$	space to store hash function
$r =$	parameter specifying the number of vertices in one part of the dependency graph
$ratio =$	$2r/m$, which specifies the relative size 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 tower

Table 6: Summary of Terminology

refer to Table 6 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

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

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.

The algorithm for selecting h consists of the three steps: Mapping, Ordering, and Searching. Each step is described in a separate section. Data structures are introduced as they are needed.

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

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

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.

There are two data structures that constitute 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 to 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
    h0, h1, h2: integer;
    nextedge1: integer;
    nextedge2: integer;
    end
```

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

Figure 4 details the Mapping step. Let $\mathtt{k_1}, \mathtt{k_2}, \ldots, \mathtt{k_n}$ be the set of keys. The $\mathtt{h_0}$, $\mathtt{h_1}$, and $\mathtt{h_2}$ 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. Note in passing that in (3) when edges are added to the appropriate linked list, that values for $\mathtt{nextedge_i}$ in the `edge` array are updated as needed. (4) examines sets of edges having the same $\mathtt{h_1}$ value to check for distinct ($\mathtt{h_0}$, $\mathtt{h_1}$, $\mathtt{h_2}$) 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 $\mathtt{h_0}$, $\mathtt{h_1}$, and $\mathtt{h_2}$ functions. The probability that random tables must be generated more than twice is exceedingly small. Therefore, the expected time for the Mapping step is $O(n)$.

(1) build random tables for h_0, h_1, and h_2
(2) **for** each $v \in [0 \ldots 2r-1]$ **do**
    `vertex[v].firstedge` $= 0$
    `vertex[v].degree` $= 0$
(3) **for** each $i \in [1 \ldots n]$ **do**
    `edge[i].`h_0 = $h_0(k_i)$
    `edge[i].`h_1 = $h_1(k_i)$
    `edge[i].`h_2 = $h_2(k_i)$
    `edge[i].nextedge`$_1 = 0$
    add `edge[i]` to linked list with header `vertex[`$h_1(k_i)$`].firstedge`
    increment `vertex[`$h_1(k_i)$`].degree`
    `edge[i].nextedge`$_2 = 0$
    add `edge[i]` to linked list with header `vertex[`$h_2(k_i)$`].firstedge`
    increment `vertex[`$h_2(k_i)$`].degree`
(4) **for** each $v \in [0 \ldots r-1]$ **do**
    check that all edges in linked list `vertex[v].firstedge` have
        distinct (h_0, h_1, h_2) triples.
(5) **if** triples not distinct **then**
    repeat from step (1).

Figure 4: The Mapping Step

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

$$K(v_i) = \{k_j | h_1(k_j) = v_i, h_2(k_j) = v_s, s < i\}$$

Similarly, if $r \leq v_i \leq 2r-1$, then

$$K(v_i) = \{k_j | h_2(k_j) = v_i, h_1(k_i) = v_s, s < i\}.$$

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

An analogy with Prim's algorithm 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 maximum 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 maximum 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 maximum 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 5 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 four stacks and one traditional heap. Most vertices have degree 1, 2, 3, or 4. One stack is provided for each of these 4 degrees. A traditional 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. Because there are so few vertices of degree ≥ 5, the heap operations on `VOHEAP` require time much less than $O(\log n)$ each. The time for the Ordering step is thus $O(n \log n)$ and is actually close to linear.

There is one issue not addressed in Figure 5: the dependency graph may not be connected. Typically, the dependency graph consists of one large connected component and a number of

```
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 5: The Ordering Step

smaller components. By choosing v_1 as a vertex of maximum 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 that have not been selected and can easily find an unselected vertex of maximum degree. Therefore, the Ordering step is able to order all vertices of degree > 0.

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 from 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 hash-table.

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

Figure 6 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 6 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 *ratio*, 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 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 justifies the small gain in success.

5 Example

We illustrate our algorithm using a key set of 6 words that has actually been processed to yield a MPHF. While this small example illustrates **how** the algorithm works, it does not explain (as was done in Section 3) **why** the algorithm works well for a **large** set of keys. The set of words was drawn from the initial portion of the Collins English Dictionary [HAN79]. The 6 words with their h_0, h_1, and h_2 values are given in Table 7. The h_0, h_1, and h_2 functions are the auxiliary hash functions found in the Mapping step.

```
(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
                  for each k ∈ K(v_i) do
(4)                   h(k) = edge[k].h_0 + vertex[edge[k].h_2] + s_j   (mod n)
(5)                       if hash-table[h(k)].assigned then
                              collision = true
              else
                  for each k ∈ K(v_i) do
                      h(k) = edge[k].h_0 + vertex[edge[k].h_1] + s_j   (mod n)
                          if hash-table[h(k)].assigned then
                              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 6: The Searching Step

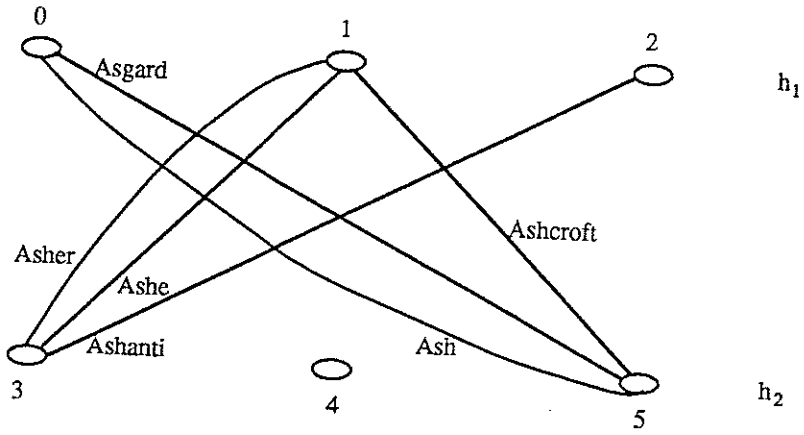


Figure 7: Example: Dependency Graph

From this assignment of h_1 and h_2 values, the bipartite dependency graph shown in Figure 7 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.

After initialization, the vertex and edge data structures for this example have contents as shown in Figure 8. We see a symbolic representation of the linked lists involved, and all of the h_0, h_1, and h_2 values. Of course, the g values (left blank here) are to be determined in the Searching step.

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 9. In 9(a), an arbitrary vertex, 1, of maximum degree, 3, is selected to be v_i; the result is $v_1 = 1$. Next, in 9(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 9(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 9(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 9(e), the selection $v_5 = 2$ is made.

The result of the Ordering step is the vertex order

$$1\ 5\ 3\ 0\ 2.$$

In an order containing 5 vertices, we obtain a tower of 4 levels. The resulting tower is shown in Table 8. 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

$$K(v_3) = \{\text{Ashe}, \text{Asher}\}.$$

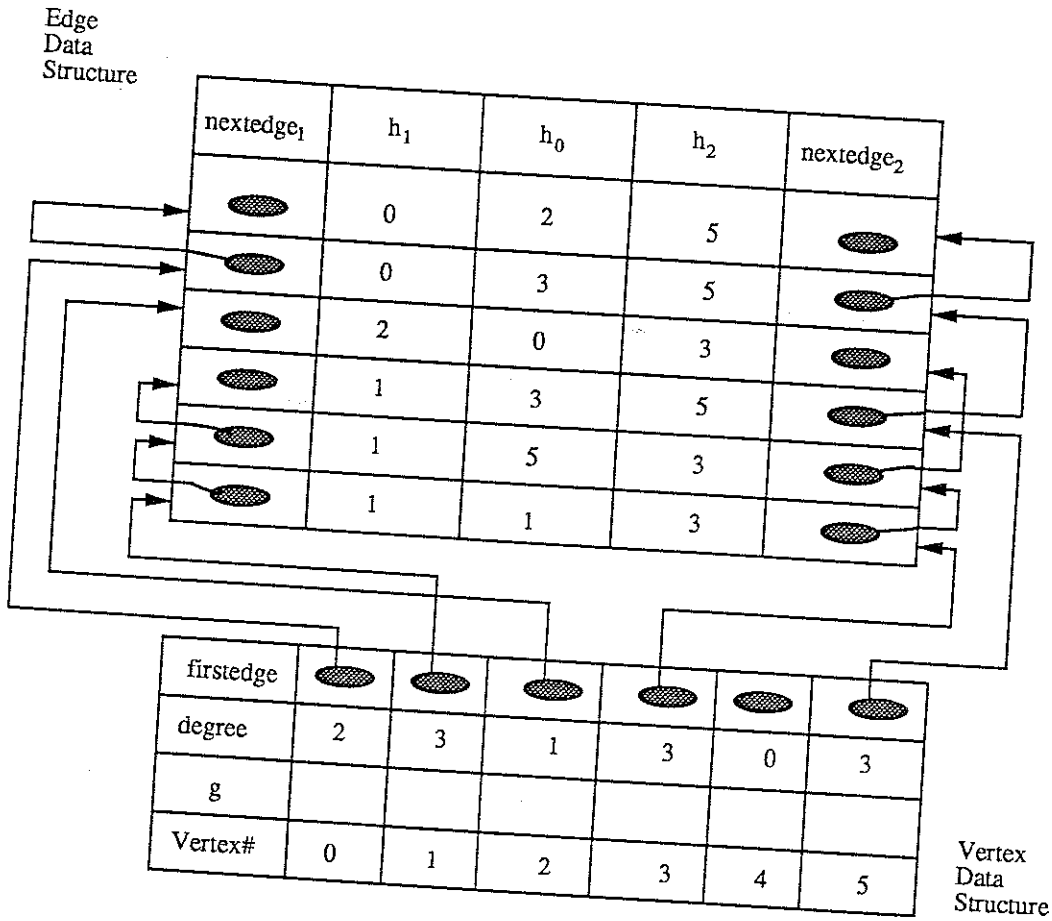
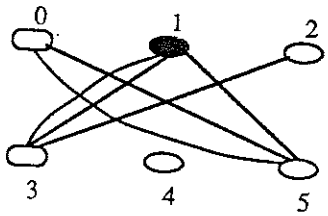


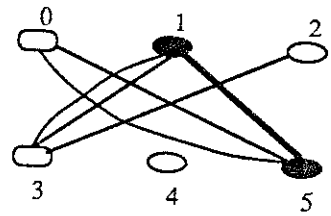
Figure 8: Example: Vertex and Edge Data Structures

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

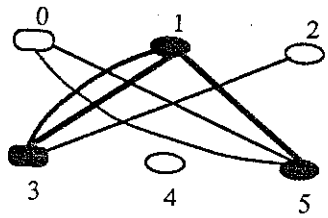
Table 8: Example: Levels in the Tower



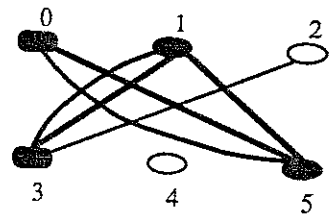
(a) Vertex selected: 1
Initialization



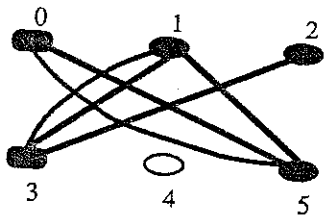
(b) Vertex selected: 5
Level: (1,5)



(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 9: Example: Ordering Step

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 assignment process is illustrated in Figure 10. The g value for $v_1 = 1$ is arbitrary; in 10(a), the assignment $g(v_1) = 2$ has been made. Vertex $v_2 = 5$ is next; $K(v_2) = \{\text{Ashcroft}\}$. We know

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

In 10(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

$$\begin{aligned} b(\text{Asher}) &= h_0(\text{Asher}) + g\big(h_1(\text{Asher})\big) \pmod 6 \\ &= 1 + 2 \pmod 6 \\ &= 3. \end{aligned}$$

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 10(c), the random value selected is $g(v_3) = 3$, which makes

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

and

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

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 10(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.

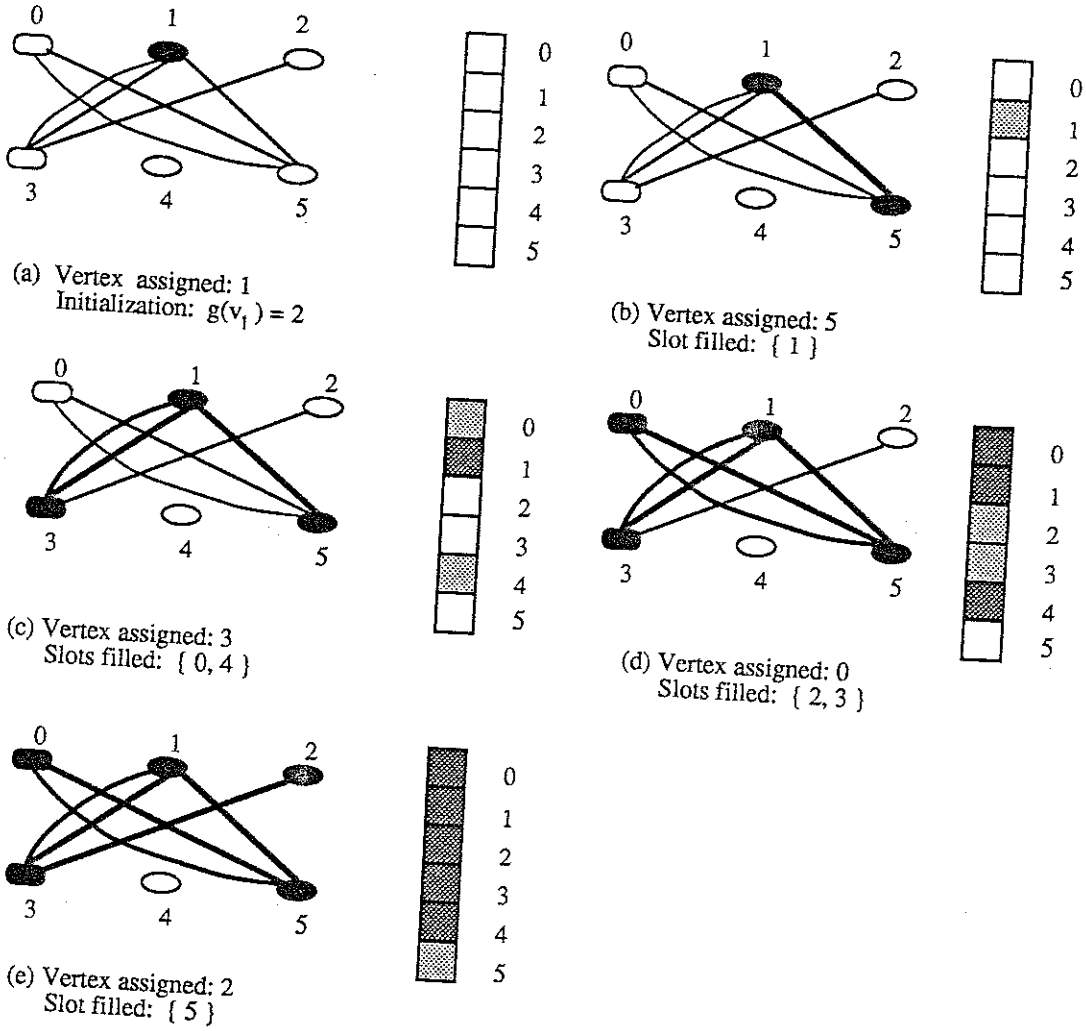


Figure 10: Example: Searching Step

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

Table 9: Example: Vertices with Computed g Values

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

Table 10: Example: Keys with Computed Hash Addresses

The selected g values are summarized in Table 9. 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 10. 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 present a series of results based on some of our experimental studies. We begin by considering the dependency graph and the associated tower.

6.1 Dependency Graph and Tower

The largest set of keys available during our initial development contained 130,198 entries. Table 11 deals with this key set, and shows the vertex degree distribution for various values of our ratio, from 0.4 through 1.0. Note that the value $ratio = 1.0$ leads to the graph that has the

most desirable distribution, but that our algorithm was successful in finding a MPHF for this set even when *ratio* was 0.4. This is rather important, since the space required to specify the MPHF is directly proportional to the ratio.

Table 11 gives the expected number of vertices of each given degree, as computed using the equations given earlier, along with the actual numbers of vertices of that degree on the h_1 and the h_2 sides of the graph. There is a relatively close fit in all cases, so our assumptions of randomness and our derivations are realistic. We further note that as the ratio decreases from 1.0 to 0.4, the maximum degree of a vertex increases from 10 to 15. Also, the number of vertices of low degree decreases, and the number of vertices of higher degree increases. One could predict, and we have observed, that for this size set, when *ratio* is much below 0.4, our algorithm cannot reliably find a MPHF. This is because the vertex distribution no longer gives a high probability of success for the Mapping and Ordering steps of our current algorithm. Nevertheless we are pleased that we have managed to find a MPHF with $ratio \leq 0.5$ since then only half the usual amount of space is required.

As mentioned earlier, the size of a level in the tower is bounded above by the degree of the associated vertex. To better understand the actual distribution of level sizes we therefore collected data from several PHF runs. In Table 12 we see the behavior for a key set of size 32, where a ratio of 0.6 was used. The number of occurrences for each level size, shown in the last column, seems quite reasonable. We have also recorded the absolute position in the processing, and the relative measure of % Full, for the situation when a level of a given size is first encountered, and when it is last encountered. This data gives a rough indication of when levels of various sizes are actually processed.

Table 13 reports the results for the same type experiment for a set of 1024 keys with $ratio = 0.6$. The last level of size 4 was handled when the disk was 41% full, the last level of size 3 when the disk was 71% full, and the last level of size 2 was handled when the disk was 88% full.

In Table 14 we see, for $ratio = 0.4$ and set size over 130,000, a similar distribution. Because of the low ratio, the algorithm encounters large levels in some cases rather late in the processing, but because the table is so large, still is successful in finding a MPHF. Note that patterns of size 4 still must fit when the table is 74% full, and that patterns of size 2 must still fit when the table is 97% full. Clearly, our probabilistic approach has led to rapid solutions in rather difficult situations.

6.2 Timing Statistics

As can be deduced from the preceding tables, we have computed MPHFs for a large number of key sets of various sizes. Table 15 illustrates these results. We compare the algorithm due to Sager as re-implemented initially by Datta and discussed in [DAT88], the improved algorithm presented at the 1989 ACM Computer Science Conference (CSC) [FCHD89], and a recent version of our algorithm. All of these runs were made using an Apple Macintosh II system running A/UX (Apple's version of UNIX) with 2 megabytes of standard memory and 8 megabytes of NU-bus memory from National Semiconductor. This machine is rated at roughly 2 MIPS.

Times are given in seconds as measured using the A/UX "times()" routine. It can be seen that Sager's method required large amounts of time when sets of around a hundred were involved. For sets of sizes 500 and 1000, the implementation of Sager's algorithm had not completed the

Ratio	Degree	# h_1 Vertices	# h_2 Vertices	Expected # of vertices
0.4	1	882	823	877.12
	2	2263	2152	2192.91
	3	3566	3754	3655.01
	4	4562	4585	4568.94
	5	4596	4592	4569.08
	6	3821	3760	3807.66
	7	2724	2741	2719.80
	8	1657	1706	1699.89
	9	921	951	944.38
	10	499	430	472.19
	11	232	219	214.63
	12	84	90	89.43
	13	35	40	34.39
	14	13	9	12.28
	15	6	8	4.09
0.5	1	2407	2422	2384.44
	2	4661	4773	4769.06
	3	6344	6308	6358.95
	4	6469	6355	6359.09
	5	5160	5014	5087.35
	6	3381	3456	3391.59
	7	1957	1922	1938.05
	8	900	980	969.02
	9	434	418	430.67
	10	171	192	172.26
	11	52	67	62.64
	12	21	18	20.88
	13	6	6	6.42
	14	3	2	1.84
	15	1	3	0.49
0.6	1	4609	4719	4644.44
	2	7767	7663	7740.96
	3	8548	8563	8601.24
	4	7234	7187	7167.79
	5	4750	4852	4778.55
	6	2736	2611	2654.75
	7	1188	1253	1264.15
	8	526	522	526.72
	9	183	200	195.08
	10	73	62	65.02
	11	25	33	19.70
	12	10	5	5.47
	13	1	0	1.40
0.7	1	7531	7552	7477.40
	2	10598	10479	10682.22
	3	10293	10300	10173.68
	4	7299	7181	7266.95
	5	4062	4186	4152.54
	6	1940	1942	1977.38
	7	843	831	807.08
	8	293	315	288.24
	9	96	90	91.50
	10	27	22	26.14
	11	5	7	6.79
	12	1	1	1.62
	13	0	2	0.36
0.8	1	10654	10771	10687.15
	2	13364	13331	13359.14
	3	11071	11002	11132.70
	4	7122	7051	6957.94
	5	3355	3508	3478.94
	6	1436	1458	1449.54
	7	539	468	517.68
	8	165	179	161.77
	9	51	41	44.93
	10	16	13	11.23
	11	0	2	2.55
	12	1	1	0.53
	13	0	2	0.10
0.9	1	14079	14003	14109.22
	2	15796	15790	15677.08
	3	11383	11468	11612.69
	4	6552	6538	6451.47
	5	2844	2868	2867.29
	6	1101	1063	1061.94
	7	332	331	337.11
	8	95	87	93.64
	9	21	32	23.12
	10	6	4	5.14
	11	1	0	1.04
1.0	1	17570	17705	17620.38
	2	17714	17334	17620.52
	3	11884	11817	11747.01
	4	5793	5959	5873.46
	5	2321	2359	2349.35
	6	772	742	783.10
	7	230	249	223.74
	8	56	50	55.93
	9	9	12	12.43
	10	0	4	2.49

Table 11: Degree Distribution for 130,198 Case

Level Size	First Occurrence		Last Occurrence		Total # of Occurrences
	Position	% Full	Position	% Full	
1	1	6.0	12	67.0	6
2	4	22.0	18	100.0	10
3	3	17.0	8	44.0	2

Note:

n = Key Set Size = 32
m = Hash Table Size = 32, ratio = 0.6
2r = # of nodes in the bipartite graph = 0.6 * n ≈ 20

Table 12: Distribution of Level Sizes for Set of Size 32

Level Size	First Occurrence		Last Occurrence		Total # of Occurrences
	Position	% Full	Position	% Full	
1	1	0.0	588	100.0	280
2	5	1.0	516	88.0	195
3	119	20.0	419	71.0	98
4	117	20.0	244	41.0	15

Note:

n = Key Set Size = 1024
m = Hash Table Size = 1024, ratio = 0.6
2r = # of nodes in the bipartite graph = 0.6 * n ≈ 614

Table 13: Distribution of Level Sizes for Set of Size 1024

Level Size	First Occurrence		Last Occurrence		Total # of Occurrences
	Position	% Full	Position	% Full	
1	1	0.0	51718	100.0	12479
2	583	1.0	50018	97.0	12756
3	1244	2.0	45603	88.0	15830
4	1907	4.0	38282	74.0	8676
5	2553	5.0	29116	56.0	1850
6	8938	17.0	19840	38.0	126
7	11256	22.0	11256	22.0	1

Note:

n = Key Set Size = 130,198
m = Hash Table Size = 130,198, ratio = 0.4
2r = # of nodes in the bipartite graph = 0.4 * n ≈ 52,080

Table 14: Distribution of Level Sizes for Set of Size 130,198

Set Size	Ordering Step			Searching Step		
	Sager	ACM CSC	New Algorithm	Sager	ACM CSC	New Algorithm
10	0.05	0.10	0.05	0.02	0.08	0.02
20	1.05	0.27	0.05	0.07	0.32	0.20
30	3.40	0.43	0.07	0.17	0.38	0.33
40	16.55	0.72	0.07	0.42	0.50	0.08
50	28.20	1.05	0.08	0.68	0.43	0.52
60	54.23	1.40	0.10	1.10	1.00	0.15
70	120.26	2.00	0.12	1.47	0.85	0.17
80	241.72	2.57	0.12	2.33	1.37	0.22
90	323.32	3.43	0.15	3.68	1.30	0.20
100	519.85	4.23	0.17	6.88	2.15	0.25
110	385.80	5.77	0.17	9.40	2.62	0.27
120	861.42	6.47	0.18	11.34	2.63	0.30
500	§	189.48	0.52	§	264.39	0.80
1000	§	2101.03	0.63	§	413.13	8.13

Notes:

Times are measured in seconds using the Mac II A/UX system routine "times()."

§ Runs for the two largest sets using Sager's algorithm had not completed the Ordering step after 5 hours.

Table 15: Timing Results, Sager vs. ACM CSC Paper Version vs. New Algorithm, Set Sizes 10 to 1000

Set Size	Mapping (seconds)	Ordering (seconds)	Searching (seconds)	Checking (seconds)	Total (seconds)
16	0.27	0.05	0.02	0.02	0.35
32	0.33	0.07	0.05	0.03	0.48
64	0.50	0.10	0.12	0.07	0.78
128	0.57	0.18	0.20	0.12	1.02
256	0.75	0.43	0.72	0.40	2.30
512	1.28	0.80	7.92	0.43	10.43
1024	1.40	1.45	2.37	1.23	6.45
2048	3.55	2.47	4.85	1.38	12.25
4096	4.27	4.92	20.97	2.12	32.27
8192	8.58	17.85	20.55	5.48	52.47
16384	20.18	72.23	142.50	13.67	248.58
32768	39.63	280.40	494.47	27.35	841.85
65536	85.03	1184.08	731.90	60.02	2061.03
130198	166.07	4802.75	1100.03	159.68	6228.53

Notes:

(1) Each g value is stored in $\log n$ bits. Therefore, if $n < 32{,}768$, each g value fits in a short integer.

(2) hash table size = set size, i.e., load factor = 1.

(3) Mapping time is the time to compute (h_0, h_1, h_2) for each key
Ordering time is the time to build the tower
Searching time is the time to search for a phf from the tower
Checking time is the time to verify that the phf is indeed perfect

Table 16: Timing Results, New Algorithm, MacII, $ratio = 1$, Set Sizes 16 to 130,198

Ratio	Mapping		Ordering		Searching		Checking		Total	
	Mac II	Seq.	Mac II	Seq.	Mac II	Seq.	Mac II	Seq.	Mac II	Seq.
0.4	157.48	47.77	90.50	26.58	2133.95	561.57	113.92	36.37	2495.85	672.28
0.5	165.37	48.38	260.22	40.23	2021.00	512.92	98.20	36.43	2544.78	637.97
0.6	162.85	48.83	548.83	99.55	3315.75	801.02	115.45	36.53	4142.88	985.93
0.7	160.62	49.43	812.28	217.78	927.62	183.15	114.83	36.40	2015.35	486.77
0.8	161.12	50.43	1656.40	501.10	915.58	183.23	112.33	36.53	2845.53	771.30
0.9	164.98	50.65	2762.10	946.48	2212.68	584.20	123.72	37.15	5263.48	1618.48
1.0	166.07	51.23	4802.75	1728.17	1100.03	252.27	159.68	36.67	6228.53	2068.33

Table 17: Timing Results, New Algorithm, $ratio = 0.4, \ldots, 1.0$, MacII vs. Sequent, Set Size 130,198

Ordering step after 5 hours and was aborted. The extended algorithm reported at ACM CSC succeeded with sets of size 1000. However, it is clear that the new algorithm discussed in this paper was significantly better in all cases.

Table 16 details the behavior of the new algorithm as set sizes vary from 16 to 130,198 words. We fix $ratio = 1.0$ and work with a Mac II in all cases. Our claim of an $O(n \log n)$ algorithm is supported.

We further focus on the results of the algorithm described in this paper in Table 17. Here we consider the times for the set with over 130,000 keys running both on the Mac II system described above and on the Sequent Symmetry in our Computer Science Department, with 10 processors each rated at 4 MIPS. The Sequent has 32 megabytes of main memory. The code run on the Sequent was not changed to exploit the parallel architecture; the speed-up shown is simply due to faster processing (on the single processor involved).

We notice some fluctuation in times as *ratio* varies. With smaller *ratio* the graph is smaller so Mapping is slightly faster, and Ordering is much faster. However, Searching is usually more expensive, because more rotations are required due to having higher degree vertices to process later on. Also, the sequential probing used by Sager is not as fast or as successful as the random probing used in our Searching step (refer to Tables 4 and 5.) The Ordering time for larger graphs is clearly so large for bigger *ratio* values that the total time seems to be almost linear with the

ratio chosen. While it generally appears safe to use a *ratio* about 0.7, which gives reasonably rapid processing and some space saving, a wise strategy for this size set might be to start looking for a MPHF with *ratio* about 0.4 and then only increase the *ratio* if the graph that results is too difficult to process.

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, it does seem that with an appropriate value for *ratio* the algorithm finds a MPHF with high probability, in time $O(n \log n)$.

6.3 A Very Large Key Set

The largest set of keys we have been able to handle directly with our algorithm is a collection of over 420,878 French words, since our data structures have been tuned to require a maximum of $5n$ computer words. Our Sequent has sufficient primary memory for this to work well. The run used $ratio = 0.5$. The Mapping step was modified to use 8 stacks, one for each vertex degree between 1 and 8. This modification led to a fast build time of 49.07 seconds, while the time for the Searching step was 1409.33 seconds. The total time for our algorithm to find on the Sequent a MPHF for the 420,878 words was 1623.83 seconds.

For very large key sets, the primary limitation on the efficiency of our algorithm comes from the size of the 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 16 subgraphs of approximately equal size. The *ratio* was 0.38. The total time to construct the MPHF was 6216.73 seconds. It is interesting to note that this time is only roughly 3 times the corresponding time on the Sequent Symmetry for the smaller set of 130,000 words. This is evidence that paging significantly slows the algorithm for a set of as few as 130,000 keys.

6.4 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 [FOX89]. In particular, we took the 130,198 word collection discussed earlier 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 [HAN79] or in files we have extracted from the AIList Digest (distributed over the Internet), the two sources for this set, and be instantly told what hash value has been assigned.

Table 18 illustrates some of the results obtained. In particular we have given the string and hash table address for the first, middle, and last ten entries in the file.

Word	Hash Address
x-rays	0
Euclidean	1
ethyl ether	2
Clouet	3
Bulwer-Lytton	4
dentifrice	5
Lagomorpha	6
Chungking	7
quibbles	8
Han Cities	9
reflexes	65095
encumbered	65096
kenaf	65097
Benedikt	65098
erythromycin	65099
endowing	65100
radically	65101
dictation	65102
soft-shelled turtle	65103
Theropoda	65104
Ilya	130188
Nez Perc	130189
Zieyler catalyst	130190
Georgia	130191
epic simile	130192
storybook	130193
postglacial	130194
cyanate	130195
wildfire	130196
unstated	130197

Table 18: Keys and Hash Addresses on Virginia Disc One (excerpts)

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

7 Future Work

7.1 Improvements

The algorithm has undergone a small amount of tuning, and more is planned. Because the g values are in the range $0 \ldots n-1$, each g value requires only $\log n$ bits. Other ways to reduce the storage required to specify a MPHF are under investigation. A parallel version of the algorithm is likely to produce significant improvements in speed. Promising initial results have been obtained.

7.2 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 begin 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.

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.

8 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

[CW79] Carter, J. L. and Wegman, M. N. Universal classes of hash functions. *Journal of Computer and System Sciences* **18,** 1979, 143-154.

[CKB83] 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.

[CHA84] Chang, C. C. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM* **27,** 1984, 384-387.

[CHA86] Chang, C. C. Letter oriented reciprocal hashing scheme. *Information Sciences* **38,** 1986, 243-255.

[CIC80] Cichelli, R. J. Minimal perfect hash functions made simple. *Communications of the ACM* **23,** 1980, 17-19.

[DAT88] 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.

[ED88] Enbody, R. J. and Du H. C. Dynamic hashing schemes. *ACM Computing Surveys* **20,** 1988, 85-113.

[FEL68] Feller, W. *An Introduction to Probability Theory and Its Applications, Volume 1.* John Wiley and Sons, New York, 1968.

[FOX88] 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, 1988, Vol. 23, 85-124.

[FOX89] Fox, E. A. Virginia Disc One. CD-ROM developed at Virginia Polytechnic Institute & State University and produced by Nimbus Records, Ruckersville, VA. 1989.

[FCHD89] 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,* Feb. 21-23, 1989, Louisville, KY, 114-122

[FNAEM88] 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,* Feb. 9-12, 1988, Austin, TX, 101-108.

[FWSCF86] 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.

[FKS84] 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.

[HAN79] Hanks, P., editor. *Collins English Dictionary.* William Collins Sons & Co., London, 1979.

[JAE81] Jaeschke, G. Reciprocal hashing—a method for generating minimal perfect hash functions. *Communications of the ACM* **24,** 1981, 829-833.

[KNU73] Knuth, D. E. *The Art of Computer Programming,* Volume 3, *Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[PAL85] Palmer, E. M. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[PM88] Park, S. K. and Miller, K. W. Random number generators: good ones are hard to find. *Communications of the ACM* **31,** 1988, 1192-1201.

[RL89] Ramakrishna, M. V. and Larson, P. File organization using composite perfect hashing. To be published in *ACM Transactions on Database Systems.* 1989.

[SAG84] 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.

[SAG85] Sager, T. J. A polynomial time generator for minimal perfect hash functions. *Communications of the ACM* **28,** 1985, 523-532.

[SPR78] Sprugnoli, R. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM* **20,** 1978, 841-850.