

**Order Preserving Minimal Perfect Hash Functions  
and Information Retrieval**

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

**TR 91-1**



# Order Preserving Minimal Perfect Hash Functions and Information Retrieval \*

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

February 8, 1991

## Abstract

Rapid access to information is essential for a wide variety of retrieval systems and applications. Hashing has long been used when the fastest possible direct search is desired, but is generally not appropriate when sequential or range searches are also required. This paper describes a hashing method, developed for collections that are relatively static, that supports both direct and sequential access. The algorithms described give hash functions that are optimal in terms of time and hash table space utilization, and that preserve any a priori ordering desired. Furthermore, the resulting order preserving minimal perfect hash functions (OPMPHF's) can be found using time and space that are linear in the number of keys involved and so are close to optimal.

**CR Categories and Subject Descriptors:** E.2 [Data Storage Representations]: *hash table representations*; H.2.2 [Database Management]: Physical Design - *access methods*; H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing - *indexing methods*; H.3.2 [Information Storage and Retrieval]: Information Storage - *file organization*

---

\*This work was funded in part by grants or other support from the National Science Foundation (Grant IRI-8703580), Online Computer Library Center, Inc., NCR Corporation, and the VPI&SU Computing Center.

**General Terms:** Algorithms, Experimentation

**Additional Keywords and Phrases:** perfect hashing, random graph, minimal perfect hashing, indexing, dictionary structure, inverted file structures

## 1 Introduction

### 1.1 Motivation: Sources of Static Key Sets

This work was in part motivated by our investigations of optical disc technology. In the last decade, developments in this area have had a revolutionary impact on computer storage, lowering the price per unit of storage by three orders of magnitude, enabling many new computer and publishing applications, and encouraging a number of research investigations [FOX88b]. In publishing a series of CD-ROMs at VPI&SU, we have found the need for guaranteeing single-seek access to data, and have indeed included a demonstration of our earlier work with minimal perfect hash functions (MPHF's) on Virginia Disc One [FOX90].

Another reason for our work is to allow rapid access to objects in large network databases. Building upon earlier work with "intelligent" information retrieval in connection with the CODER (COMposite Document Expert/extended/effective Retrieval) system [FOX87], we observed the value of having the contents of machine readable dictionaries in an easy to manipulate computer form [FOX88a]. A large lexicon of this type should be useful to aid information retrieval by allowing automatic and semi-automatic query expansion [NUTT89]. Further, it should support a range of text understanding and other natural language processing activities [FRAN89]. However, these lexicons contain a large number of (relatively static) objects that must be rapidly located; rapid traversal of associational links is also required. We [CHEN90] elected to specify and build a Large External Network Database (LEND) and have indeed loaded over 70 megabytes of data into our current implementation. Further work is planned, showing how network databases of lexical data or other information often stored in semantic networks, as well as complex hyperbases (for hypertext and hypermedia), can be constructed to aid information retrieval [CHEN90]. All of these efforts make use of our work with MPHF's.

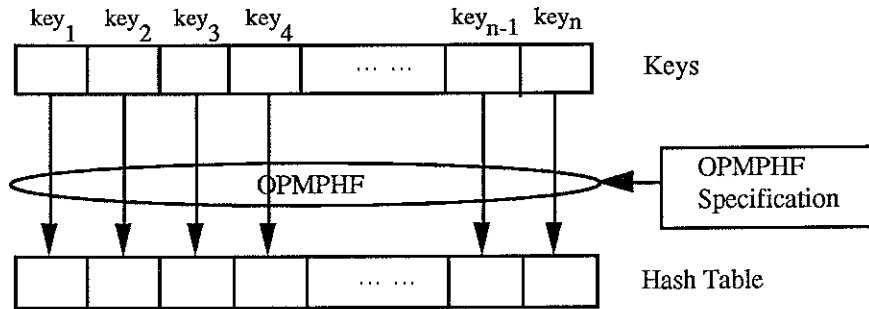


Figure 1: Order Preserving Minimal Perfect Hash Function

## 1.2 Minimal Perfect Hash Functions, Preserving Order

Our initial work with hash functions took a different tact from currently popular methods of dynamic hashing [ENBO88]. Those methods are suitable when it is acceptable to use extra space, and necessary to allow for frequent additions and deletions of records. While dynamic hashing generally does not preserve the original key ordering, there also exists order-preserving key transformations, which are appropriate for dynamic key sets as long as the key distributions are or can be made to be stable [GARG86]. In contrast, we made the very useful assumption that our key sets are static, and investigated published algorithms for finding minimal perfect hash functions (MPHF's), i.e., those where no collisions occur and where the hash table size is the same as the size of the key set (see our review of earlier work [DAT88]). Of those examined, one by Sager [SAGE85] had polynomial time complexity,  $O(n^4)$ , and seemed amenable to enhancement. With some small extensions we were able to handle thousands of keys, with an  $O(n^3)$  algorithm for  $n$  unique keys [FOX89a]. By reformulating the problem, we developed an  $O(n \log n)$  algorithm and tested it with a variety of key sets, including one with  $n = 1.2$  million [FOX89b]. We have developed and tested even better algorithms, and have submitted that work for publication.

This paper, however, focuses on MPHF's that also have the property of preserving the order of the input key set - these are denoted "OPMPHF's" since they are order preserving. Because they are of special value for information retrieval applications, we elaborate on this part of our work. To make it clear what is implied, consider Figure 1. A function must be obtained that maps keys, usually in the form of character strings or concatenations of several numeric fields, into hash table locations. In brief, the  $i^{th}$  key is mapped into the  $i^{th}$  hash table location, so the key order is preserved in the ordering of entries of the hash table.

### 1.3 Applications for Information Retrieval

While there are numerous applications for our methods, it is appropriate to consider two that are particularly well known and important for information retrieval. First, there is the dictionary. Here the object is to take a set of tokens or token strings (words, phrases, etc.) and allow rapid lookups to find associated information (number of postings of a term, the “concept number” for that entry, pointers to inverted file lists, etc.). If Order Preserving Minimal Perfect Hash Functions (OPMPHF) can be used for this purpose, in one disk access the record of any dictionary item can be identified, and it is possible to rapidly find previous or subsequent entries as well. Thus, the dictionary can be kept in lexicographic order, and can be read sequentially or accessed directly. This application is illustrated in Figure 2a, where real data from the *Collin’s English Dictionary* [HANK79] is given for illustrative purposes; this CED example is discussed later as well since some of our experimental studies were with a large set of keys in part derived from the CED.

A second application is for accessing inverted file data. Figure 2b illustrates selected data taken from the CISI collection [FOX83]. For a given term ID (identifier), it is usually necessary to find the number of postings, that is the number of documents in which the term occurs, and then to find the list of all those occurrences. All of this information has been included in a single file accessible by an OPMPHF. Normally, for a given term ID, we obtain the document and frequency (of that term in that document) pairs for all occurrences. Assuming that document numbers have value at least 1, we use the simple trick of storing the postings data in the frequency field of an entry that has a given term ID and document number set to 0. Thus, we can, for a given term ID, build a key formed by concatenating the value 0 to it, find the postings in one seek, and read the document-frequency pairs that appear directly after. Various methods using unnormalized forms of the data are possible to effect space savings. Indeed, the associated value for a key that can be located using OPMPHF can actually be an arbitrary value so that variable length records can be directly addressed [DAOUD90].

### 1.4 Summary of Earlier Work

Our earlier work has been discussed in [FOX89b], along with an overview of related work. We review the key concepts here. First, there is theoretical evidence that since MPHFs are rare in the space of all functions, a moderate amount of space is required to specify a given MPHf [MEHL82]. In a separate (submitted) paper we describe MPHf methods

	Term Id	Doc Id	Weight
	0	0	1
	0	1271	3
Aveynn	1	0	102
Bulwer-Lytton	1	11	1
Carl	1	16	3
Chunking	1	17	3
Clouet	....	....	....
Euclidean	....	....	....
Han Cities	....	....	....
Indonesia.	9999	0	5
Lagoomorpha	9999	447	1
Sabbaths	9999	939	1
antennae	9999	988	1
burrows	9999	1250	1
debris	9999	1429	2
deposited	10000	0	1
dentifrice	10000	177	1

a) Partial Dictionary  
from CED

b) Partial Inverted File  
from CISI

Figure 2: Using OPMPHF's for Information Retrieval

that require space approaching the theoretical lower bound. In this paper (see section 3.1), a proof of the lower bound for OPMPHF's is given, and that bound is approached by the current algorithm. Thus, while readers might be concerned that using space to specify a function is contrary to the spirit of hashing, it is required based on theoretical analysis.

Second, the approach we take is to use a three step process of Mapping, Ordering, and Searching — somewhat like the procedure used by Sager [SAGE85]. We map the problem of finding a MPHF into one involving working with a random bipartite graph, where each given key is represented by an edge, and where randomness allows us to make use of important results from the theory of random graphs (see, for example, [BOLL85] and [PALM85]). Since in the original problem space we must avoid collisions among keys, in our graph we must identify dependencies between edges, which result when multiple edges share a common vertex. These dependencies are captured during the Ordering phase, which makes use of properties of the dependency graph, and which leads to an ordering of levels or groups of interdependent edges. If the Ordering phase is done well, then during the subsequent Searching phase, when the actual hash values are assigned so as to avoid collisions, a viable MPHF can be quickly specified.

To facilitate subsequent discussion, we adapt notation used in [FOX89b], relating to our work with MPHF's, and list it for reference in Figure 3.

$U$	=	universe of keys
$N$	=	cardinality of $U$
$k$	=	key for data record
$S$	=	subset of $U$ , i.e., the set of keys in use
$n$	=	cardinality of $S$
$T$	=	hash table, with slots numbered $0, \dots, (m - 1)$
$m$	=	number of slots in $T$
$h$	=	function to map key $k$ into hash table $T$
$ h $	=	space to store hash function
$G$	=	bipartite dependency graph
$r$	=	parameter specifying the number of vertices in one part of $G$
<i>ratio</i>	=	$2r/m$ , which specifies the relative size of $G$
$h_0, h_1, h_2$	=	three separate pseudo-random functions easily computable over the keys
		$h_0: U \rightarrow [0, \dots, n - 1]$
		$h_1: U \rightarrow [0, \dots, r - 1]$
		$h_2: U \rightarrow [r, \dots, 2r - 1]$
$g$	=	function mapping $0, \dots, (2r - 1)$ into $0, \dots, (m - 1)$
$h(k)$	=	$\{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod n$
	=	form of hashing function
$v$	=	vertex in $G$
$SG$	=	a subgraph of $G$
$VS(SG)$	=	vertex sequence produced during the Ordering phase
$t$	=	length of $VS(SG)$
$K(v)$	=	for a given $v$ in the vertex ordering, the set of keys in that ordering level

Figure 3: Terminology from Earlier Work on MPHFs



Note that when  $n = m$  the hash function is minimal, as desired, so in the following discussion  $n$  will be used instead of  $m$ . In the bipartite dependency graph  $G$  there are two parts having  $r$  vertices (numbered from 0 to  $r - 1$  and from  $r$  to  $2r - 1$ , respectively), each part connected to the other part by  $n$  edges. One end of each edge associated with key  $k$  is at the vertex numbered by  $h_1(k)$ , and the other end is at the vertex numbered by  $h_2(k)$ . Thus, each edge is uniquely defined by the associated key. The function  $h(k)$  is the one actually used with key  $k$ , and is easily computable from  $k$ , given a specification of  $g$  for all values in its domain.

Central to our algorithms is an analysis of the properties of the graph  $G$ , which is random since it is formed through use of the pseudo-random functions  $h_1()$  and  $h_2()$ . When the ratio (i.e.,  $2r/n$ ) is 1 or more, the graph has few vertices with high degree. When the ratio falls below 0.5, fewer vertices have low degree and the graph has larger connected components and more cycles. More detailed results are given in [FOX89b] for graphs with ratios as small as 0.4, but for OPMPHF's found using the current scheme, ratios are around 1.2. Other graph properties also are considered in the discussion below.

## 1.5 Outline of Paper

This paper is organized as follows. In section 2 we explain our approach, including three methods to find OPMPHF's, and then provide both details and an example for the third method. Section 3 gives analytical and experimental results, including lower bounds and other descriptive information about our methods, as well as confirming evidence from several runs with test collections. Section 4 gives timing results for our test collections, where a dictionary and an inverted file were implemented using an OPMPHF. In section 5 we give a short description of a slightly modified algorithm which takes less space to store the function. Finally, we summarize our results in section 6.

## 2 Approach

This section describes our methods to obtain an OPMPHF. In section 2.1 we outline three methods to find OPMPHF's, and then focus on the third method, which requires less space than the other two. This method is fully described in section 2.2, and is illustrated with an example in section 2.2.4.

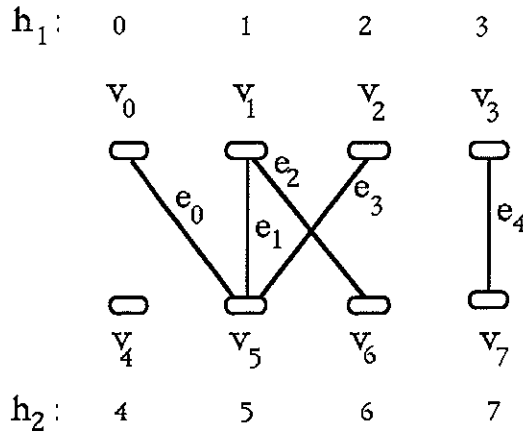


Figure 4: A Cycle Free Bipartite Graph

## 2.1 Three Methods to Find OPMPHF's

Based on our experience working on various versions of MPHf algorithms, we note that there are at least three ways to obtain an OPMPHF. The first two are straightforward extensions of our earlier research, but require a large amount of space to describe the OPMPHF. The third method, obtained after extensive study of graphs used with MPHf's, requires much less space but is rather complex.

### 2.1.1 Method 1: Acyclic Graphs

Method 1, the acyclic technique, involves constructing a bipartite graph  $G$  sufficiently large so that no cycles are present. This extends our earlier work described in [FOX89b], and is based on the use of a large ratio  $(2r/n)$  which makes the probability of having a cycle approach 0 (see proof in section 3.2.2). If there are no cycles, we have sufficient freedom during the Searching phase to select  $g$  values that will preserve any a priori key order.

Our algorithm is basically the same as that described in [FOX89b] throughout the Mapping and Ordering phases. But because  $G$  is acyclic, we obtain an ordering of non-zero degree vertices  $v$  to yield levels  $K(v)$  following certain constraints (see section 2.2.2), which only contain one edge (one key). This is achieved through an edge traversal (e.g., depth-first or breadth-first) of all components in  $G$ . Thus, in Figure 4, which shows an acyclic bipartite graph, an ordering obtained by depth-first traversal of first the left connected component and then the right might give the vertex sequence  $(VS) : [v_1, v_5, v_0, v_2, v_6, v_3, v_7]$ . The corresponding levels of edges are given in the edge sequence:  $[\{\}, \{e_1\}, \{e_0\}, \{e_3\}, \{e_2\}, \{\}, \{e_4\}]$ . Notice that in this example, each level has at most one edge, which is only possible if  $G$  is acyclic.

During the Searching phase, a single pass through the ordering can determine  $g$  values for all keys in a manner that preserves the original key ordering. This is possible since with only one edge being handled at each level, there are no interdependencies that would restrict the  $g$  value assignments.

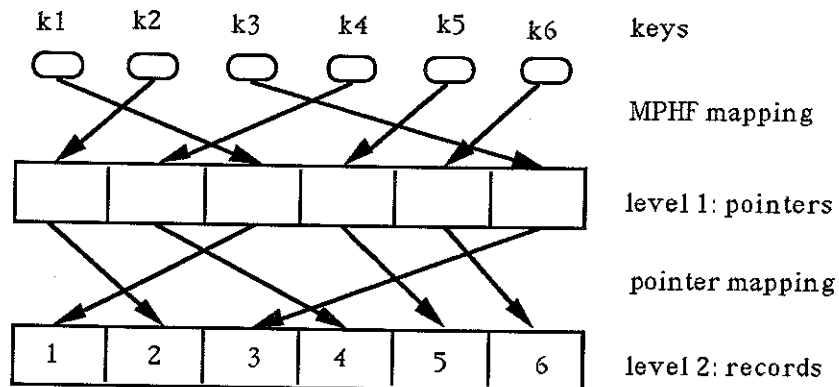


Figure 5: A Two Level OPMPHF Scheme

Although this approach is simple, it is only practical if a small acyclic graph can be found. Using our ratio,  $2r/n$ , we therefore give a lower bound on the number of vertices for a given set of  $n$  keys. Section 3.2.2 gives a detailed probabilistic account of the expected number of cycles in  $G$ , as a function of the ratio. If the average number of cycles,  $E(Y)$ , approaches 0, then by Chebyshev's inequality

$$P(Y \geq t) \leq E(Y)/t,$$

so the probability of a particular graph having cycles approaches 0. Thus, for sufficiently large ratio (i.e.,  $ratio \rightarrow \infty$ ), it will be very unlikely that  $G$  will have cycles. However, this ratio and the size of  $G$  is very much larger than values required in the other two methods described below.

### 2.1.2 Method 2: Two Level Hashing

The second idea is to use two level hashing. Here the MPHF computed through the method in [FOX89b] is in the first level and an array of pointers is in the second. A hash value from the MPHF addresses the second level where the real locations of records are kept. The records are arranged in the desired order. This method uses at the first level  $2r$ , and at the second,  $n$  computer words for the OPMPHF. For large key sets,  $2r \cong 0.4n$  is possible and feasible. Thus this method typically will use  $1.4n$  computer words. Figure 5 illustrates the two level hashing scheme. Note, however, that it is much easier and faster to find small OPMPHF's using Method 3, which is discussed next.

### 2.1.3 Method 3: Using Indirection

The third method is based on the idea of using  $G$  to store the additional information required to specify a MPHf that also preserves order. For  $n$  keys, if our graph has somewhat more than  $n$  vertices (i.e., if ratio  $> 1$ ), then there should be enough room to specify the OPMPHF. In a random graph of this size, a significant number of vertices will have zero degree; we have found a way to use those vertices. The solution is to use indirection for some of the keys; this solution can be viewed as a combination of Methods 1 and 2. This means that some keys will be mapped using indirection, in this case using the composition

$$order(k) = h(k) = g(\{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod 2r)$$

while on the other hand, the desired location of a key that is, as before, found directly is determined by:

$$order(k) = h(k) = \{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod n.$$

Note that we use the  $g$  function in two ways, one way for direct keys and the other way for keys that are handled through indirection. In any case,  $h(k)$  preserves the original ordering of keys (i.e.,  $order(k)$ ).

Let us consider more closely the distribution of  $X$ , the degree of a vertex in  $G$ . The actual distribution is binomial and can be approximated by the Poisson:

$$\begin{aligned} E(X = d) &= \{2r e^{-n/r} (n/r)^d\} / d! \\ E(X = 0) &= 2r e^{-n/r} \end{aligned}$$

When  $2r = n$ , about 13.5% of the vertices have zero-degree. If these zero-degree vertices can be used to record order information for a significant number of keys, then it is not necessary for  $G$  to be acyclic to generate an OPMPHF. In our actual algorithm, we also can use one vertex from each acyclic component in  $G$  to accept indirect keys. Figure 6 is a brief demonstration of the idea. Note that keys associated with edges  $e_0$  and  $e_1$  can be indirectly hashed into zero-degree vertices  $v_6$  and  $v_2$ . In this example it is not necessary, but if another indirect key had to be accommodated, one vertex from the set  $\{v_1, v_5, v_3, v_7\}$  could be used.

In general, an edge (key) is indirectly hashed when that situation is described by information associated with its two vertices, given by  $h_1(k)$  and  $h_2(k)$ . Usually, indirection can be indicated using one bit per vertex that is decided at MPHf building time and that is subsequently kept for use during function application time.

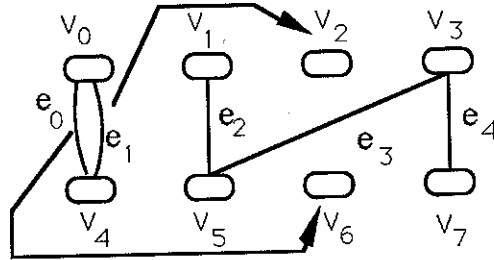


Figure 6: Zero Degree Vertices are Useful

Various schemes of indirection have been proposed and tested. In section 2.2, we describe our one-bit algorithm which is capable of finding ordered hashing functions with high probability for large key sets with *ratio*  $\cong 1.20$ .

## 2.2 Method 3: Algorithm and Data Structures

This section outlines an algorithm using one indirection bit, which is an extension of the one in [FOX89b] used to find MPHFs. Our hashing scheme uses the OPMPHF class:

$$h(k) = g(\{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod 2r),$$

when the indirection bit associated with the two vertices for this key have the same value, and otherwise uses

$$h(k) = \{h_0(k) + g(h_1(k)) + g(h_2(k))\} \bmod n.$$

The algorithm for selecting proper  $g$  values and setting indirection bits for vertices in  $G$  consists of the three steps: Mapping, Ordering and Searching. By reducing the problem of finding an OPMPHF to these three subproblems, we can more easily and rapidly identify a usable hash function. Each step, along with implementation details, will be described in a separate subsection below.

### 2.2.1 The Mapping Step

This step is essentially identical to that discussed in [FOX89b]. The only addition is that the indirection bit must be included in the vertex data structure. Readers may elect to skip to the next subsection, or to follow the discussion below which is included for completeness.

The basic concept is to generate unique triples of the form  $(h_0(k), h_1(k), h_2(k))$  for all keys  $k$ .  $h_0(), h_1(), h_2()$  are simple pseudo-random functions. Figure 7(a) illustrates one assignment of  $h_0, h_1$ , and  $h_2$  values to keys yielding the graph G shown in Figure 7(b). Since the final hash function should be perfect, all triples are distinct. Following [FOX89b], we use pseudo-random functions  $h_0, h_1, h_2$  to build the triples so as to obtain a probabilistic guarantee on the distinctness of the triples. The probability that all triples will be unique is:

$$\begin{aligned} P &= nr^2(nr^2 - 1) \dots (nr^2 - n + 1) / (nr^2)^n = (nr^2)_n / (nr^2)^n \\ &\cong e^{-n^2/2nr^2} \quad (\text{by an asymptotic estimate from [PALM85]}) \\ &= e^{-n/2r^2}. \end{aligned}$$

Since  $r$  is on the order of  $n$ ,  $P$  goes to one as  $n$  approaches infinity.

The  $h_0, h_1$  and  $h_2$  values for all keys are entered into an array *edge* defined as

```
array of [0 ... n - 1] of record
  h0, h1, h2: integer;
  nextedge1: integer;
  nextedge2: integer;
  order: integer
```

Here the combination  $h_0, h_1, h_2$  field contains the triple. The nextedge<sub>*i*</sub> field ( $i = 1, 2$ ) indicates the next entry in the *edge* array with similar  $h_i$  value to the current entry. It is used to link together all edges joined to a vertex. The order field is the desired hash location of a key.

The  $g$  function is recorded in another array *vertex* defined as follows.

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

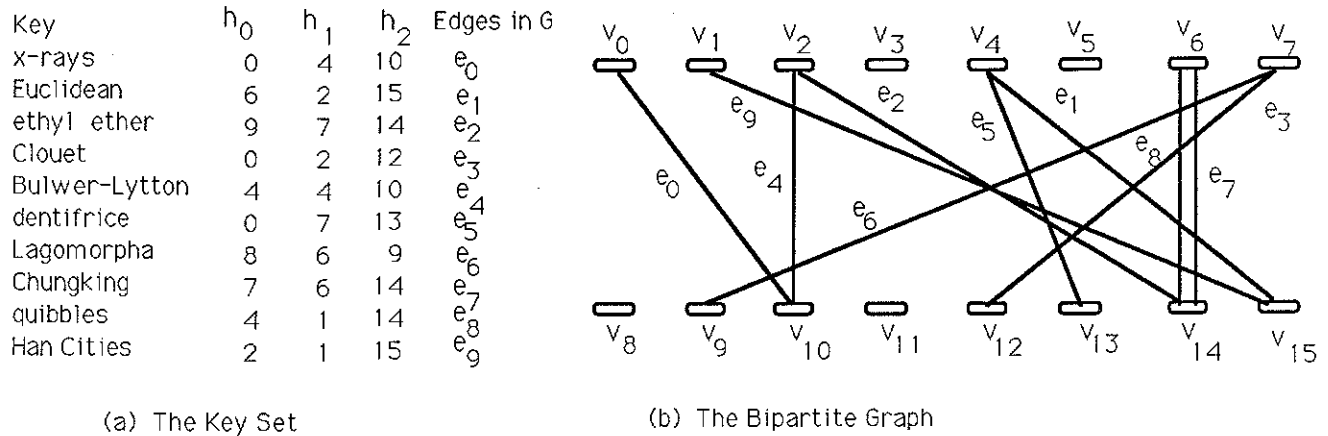


Figure 7: A Key Set and its Dependency Bipartite Graph  $G$

Here  $vertex[i].g$  records the final  $g$  value for  $h_1(k) = i$  if  $i$  is in  $[0, r - 1]$  or the final  $g$  value for  $h_2(k) = i$  if  $i$  is in  $[r, 2r - 1]$ . The mark field contains a bit of indirection information, as given above for either  $h_1(k)$  or  $h_2(k)$ . Also,  $vertex[i].firstedge$  is the header for a singly-linked list of the keys having  $h_1(k) = i$  if  $i$  is in  $[0, r - 1]$  or the keys having  $h_2(k) = i$  if  $i$  is in  $[r, 2r - 1]$ . The firstedge field actually points at an entry in the edge array indicating the start of the list and  $nextedge_i$  (for  $i = 1, 2$ ) there connects to the rest of the list. The degree field is the length of the list or equivalently the degree of the vertex. Thus, the *edge* and *vertex* arrays give a representation of a bipartite graph  $G$ .

Appendix A details the sub-steps within the Mapping Step. Step (1) builds the random tables that specify the  $h_0$ ,  $h_1$  and  $h_2$  functions. Step (2) initializes the two key (edge) related fields of the *vertex* array. Step (3) constructs the graph representation for each key  $k_i$ . Step (4) validates the distinctness of triples. Step (5) enforces the repetition of the steps from (1) to (4) under the rare circumstance that triples duplicate. It is trivial to show that steps (1), (2) and (3) all take  $O(n)$  time. Step (4) is linear on average also, because each vertex usually has quite small degree. Thus, the total Mapping step is  $O(n)$ .

### 2.2.2 The Ordering Step

In the Ordering step it is necessary to obtain a good vertex sequence  $VS$  for use later in the Searching step. Specifically,  $VS$  specifies a sequence of the vertices so that, during searching, each related set of edges can be processed independently. For a given vertex  $v_i$

in the ordering, these related edges contained in  $K(v_i)$  (i.e., at that level) are the backward edges, going to vertices that appear earlier in the ordering. Taking the bipartite graph in Figure 7 (b) as an example, one of the  $16!$  possible vertex sequences is

$$VS = [v_6, v_{14}, v_2, v_{10}, v_0, v_4, v_{15}, v_1, v_9, v_7, v_{12}]$$

with corresponding levels or edge sets

$$\begin{aligned} K(v_6) &= \{\} \\ K(v_{14}) &= \{e_7, e_8\} \\ K(v_2) &= \{e_2\} \\ K(v_{10}) &= \{e_4\} \\ K(v_0) &= \{e_0\} \\ K(v_{13}) &= \{\} \\ K(v_4) &= \{e_5\} \\ K(v_{15}) &= \{e_1\} \\ K(v_1) &= \{e_9\} \\ K(v_9) &= \{\} \\ K(v_7) &= \{e_6\} \\ K(v_{12}) &= \{e_3\} \end{aligned}$$

The graph constructed from vertices in  $VS$  plus edges in  $G$  is essentially a redrawing of  $G$  that excludes zero-degree vertices, as can be seen in Figure 8.

Finding a proper  $VS$  requires that we process vertices with many backward edges (i.e., with large  $|K(v_i)|$ ), first. Thus we employ a variety of heuristics to quickly find such vertices early. The other key issue in finding a proper  $VS$  is to handle the fact that some edges will be indirect and some direct. Since the assignment of a  $g$  value for vertex  $v_i$  fully determines the hash addresses of all keys in  $K(v_i)$ , given that the  $g$  values of each previously visited vertex has been set, it is in general true that at most one key in  $K(v_i)$  can be directly hashed for a fixed  $g$  value at  $v_i$ . Thus, we must determine exactly one such key, if the Searching step is to proceed properly. In the scheme proposed, we attach one bit (namely the mark bit) in the Ordering step as well, to each vertex for the purpose. Then, when the hashing function is used, for key  $k$  we need only consider the two indirection bits (stored in primary memory) attached to the two vertices  $h_1(k)$  and  $h_2(k)$ .

Given the need to quickly find the proper  $VS$  and to decide the proper indirection bits for vertices in  $VS$ , it is essential that we obtain hints from the properties of the  $K(v_i)$ ,



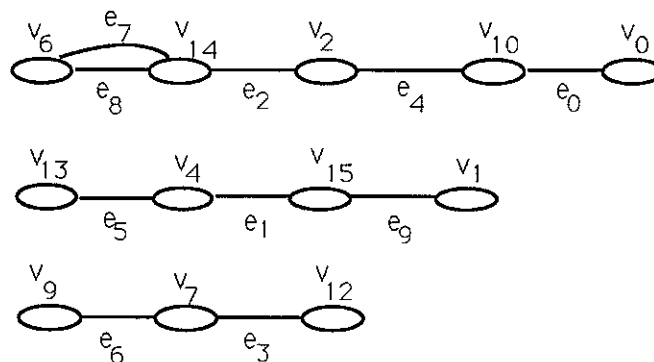


Figure 8: Redrawing of  $G$  based on a  $VS$  that excludes zero-degree vertices

such as their size. For a key  $k$  in a level where  $|K(v_i)| = 1$  and  $v_s$  is the vertex at the other side, the key can be directly hashed by setting the  $g$  value at  $v_i$  to

$$g(v_i) = [\text{order}(k) - h_0(k) - g(v_s)] \bmod n.$$

Recall that  $\text{order}(k)$  refers to the desired hash address for key  $k$ , so that we can have an order preserving function.

For keys in a level containing more than one key (i.e.,  $|K(v_i)| > 1$ ), since at most one key can be direct, hashing of the other keys requires indirection. Since in our scheme indirect hashing is indicated by the indirection bits, all such keys have those bits set accordingly and thus are indirectly hashed. After considering the two cases, we conclude that a proper  $VS$  will be one that tends to maximize the number of  $v_i$ 's with  $|K(v_i)| = 1$  and, hence, to minimize the number of  $v_i$ 's with  $|K(v_i)| > 1$ .

A practical way to obtain such a  $VS$  is to take into account the cyclic structure of  $G$ . Following standard graph terminology, we can refer to the set of edges ( $E(G)$ ) and the set of vertices ( $V(G)$ ), as given in Figure 9. Special attention must be given, though, to each connected component ( $C$ ). Clearly, edges in a tree component (denoted by  $AC$  to stand for "acyclic component") of  $G$  can be directly hashed if their vertices are included in  $VS$  by a simple depth or breadth first traversal as described in section 2.1.1. For example, in the second and third components in Figure 8, all five edges are direct. Since any vertex in an  $AC$  can be the root for a traversal and more importantly, the ordering of vertices for  $AC$  is not performed until the Searching step. At that time, the single vertex of the  $AC$  that has been used for indirection of a previous key is made the root of the traversal.

For a cyclic component (denoted by  $CCY$ ) such as the larger component at the top of

$E(G)$	=	edges of graph $G$
$V(G)$	=	vertices of graph $G$
$C$	=	connected component in $G$
$AC$	=	acyclic connected component
$CCY$	=	connected component containing one or more cycles
$CP$	=	maximal subgraph of $CCY$ containing only cut edges, each cutting $CCY$ into at least one acyclic subcomponent
$CC$	=	$CCY - CP$

Figure 9: Graph Component Terminology

Figure 8, two types of edges are distinguishable. First there are “bush” edges which are cut edges such that removal of each of them leaves at least one acyclic subcomponent. We use cycle periphery ( $CP$ ) to denote the maximal subgraph of  $CCY$  whose edges are bush edges. Note that in Figure 7(b),  $V(CP) = \{v_0, v_2, v_{10}, v_{14}\}$  and  $E(CP) = \{e_0, e_2, e_4\}$ .

All edges in a  $CP$  can be hashed directly if a vertex visiting strategy similar to that for a tree component  $AC$  is used, and the roots for visiting are vertices shared by bush edges and non-bush ones. Since the existence of  $g$  values at the root is the only precondition for assignment of  $g$  values to other vertices in  $CP$ , edges in  $CP$  should be assigned after the non-bush edges are handled.

Second, there are non-bush edges of  $CCY$ . Non-bush edges can be direct or indirect, based on a specific ordering of the vertices that these edges are connected to. We use  $CC$  to describe the component after “bush” edges are removed. In Figure 7(b), we only have indirect non-bush edges with  $V(CC) = \{v_6, v_{14}\}$  and  $E_{CC} = \{e_7, e_8\}$ . Intuitively, we see that keys where  $|K(v_i)| = 1$  should be direct and those where  $|K(v_i)| > 1$  should be indirect. However, due to the way in which the indirection bits are set, some keys where  $|K(v_i)| = 1$  may also be indirect.

In summary, our strategy to obtain a good  $VS$  involves first identifying  $ACs$ ,  $CPs$  and  $CCs$ . Second, we order vertices in  $CCs$ , then in  $CPs$  and finally in  $ACs$ . The implementation of the algorithm combines the ordering and searching for  $CPs$  and  $ACs$  in the Searching step to save one traversal of edges in  $CPs$  and  $ACs$ . In the Ordering step for vertices in  $CCs$ , a vertex whose  $K(v_i)$  set is (currently) larger is chosen next in the ordering over a vertex whose  $K(v_i)$  set is (currently) smaller. The ordering of vertices in  $CPs$  and  $ACs$  is done purely through tree traversals.

The number of vertices of  $G$  for a fixed key set is an important factor affecting the quality of  $VS$ . First,  $|V(G)|$  is theoretically bounded below by the number of keys  $n$ , as is

shown in section 3.1. Any  $G$  with smaller than  $n$  vertices has little probability of producing an OPMPHF. For  $G$  with  $|V(G)| > n$ , we have a tradeoff between the size of the OPMPHF and the ease of finding such an OPMPHF. Let  $S$  be the set of indirect keys. Then if  $G$  is large,  $|S|$  becomes small implying both an easier indirect fit for  $S$  and a larger OPMPHF. On the other hand, a small  $G$  will result in a large  $S$ , increasing the difficulty of finding an OPMPHF, though if one is found, it will be rather small. Of course, we have the final constraint that  $|S|$  must be less than the total number of  $AC$ s.

Having obtained  $VS(CC)$ , we need to mark indirection bits for all vertices in the sequence. Though not necessarily yielding an optimal marking in terms of generating a minimal number of indirect edges, the method, described in detail in Appendix B, achieves satisfactory results. Step (5) in Appendix B works as follows. Suppose we are marking all edges in  $K(v_i)$ . Without loss of generality, assume  $v_i$  is in the first side of  $G$  and  $v_j$  is a previously marked vertex adjacent to  $v_i$ .

We determine the mark bit  $v_i.mark$  using the strategy of finding as many direct keys as possible in one scan of  $VS(CC)$ . Our heuristic rule, for the several cases, is:

- a)  $v_i.mark = 1$  if  $|K(v_i)| = 0$ ; or
- b)  $v_i.mark = 1$  if  $v_j.mark = 0$  and  $|K(v_i)| = 1$ ; or
- c)  $v_i.mark = 0$  if  $v_j.mark = 1$  and  $|K(v_i)| = 1$ ; or
- d)  $v_i.mark = 0$  if  $|K(v_i)| > 1$  and all  $v_j.mark = 0$  and  $|K(v_i)| = v_i.degree$ ; or
- e)  $v_i.mark = 1$  if  $|K(v_i)| > 1$  and set all  $v_j.mark = 1$  if previously 0.

Figure 10 illustrates the marking scheme where shaded vertices are marked and the unshaded vertices are to be marked.

If  $v_i$  is on the second side, we just switch  $h_1$  and  $h_2$  for steps a) to e). A simple induction proof on the length  $i$  of  $VS(CC)$  shows that (1) a direct edge only appears in a  $|K(v_i)| = 1$  level if that edge is not forced to be indirect by e); (2) all edges in levels with  $|K(v_i)| > 1$  are indirect. Here a key  $k$  is indirect if its two mark bits are identical, and direct otherwise.

Our Ordering phase performs its job in three sub-steps (cf. Appendix B). First, all components in  $G$  are identified by assigning component IDs ( $cids$ ) as shown in Step (1) of Appendix B.  $VSTACK$  is a stack data structure that keeps all unidentified vertices adjacent to at least one identified vertex. Each time a vertex is popped from  $VSTACK$ , it gets a  $cid$  and its adjacent unidentified vertices are pushed onto  $VSTACK$ . After the identification process, all zero-degree vertices will get a 0  $cid$  and all other vertices get  $cids$  greater than 0. Step (1) can be finished in  $O(n)$  time because each non-zero vertex is in  $VSTACK$  only once, and pushing and popping operations take constant time.



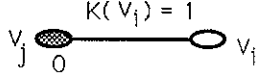
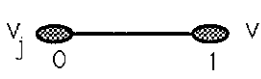
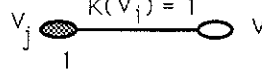
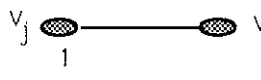
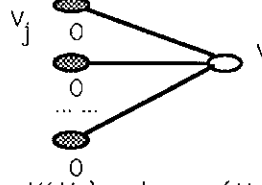
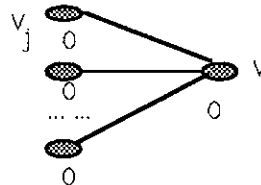
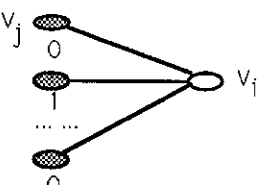
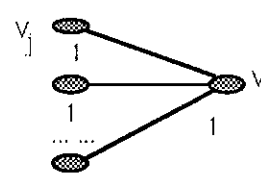
	Before marking	After marking
a)	 $v_i$	 $v_i$ 1
b)	 $v_j$ $K(v_i) = 1$ $v_i$	 $v_j$ $v_i$
c)	 $v_j$ $K(v_i) = 1$ $v_i$	 $v_j$ $v_i$
d)	 $v_j$ $K(v_i) = \text{degree}(v_i)$ $v_i$	 $v_j$ $v_i$
e)	 $v_j$ $K(v_i) \leq \text{degree}(v_i)$ $v_i$	 $v_j$ $v_i$

Figure 10: Illustration of marking process

Steps (2) and (3) recognize  $E(CP)$  in each component by manipulating the degree field. Initially, Step (2) collects all vertices of degree one into  $VSTACK$  and sets their degree field to zero. Afterwards, Step (3) takes  $VSTACK$  and tries to find more vertices whose degree could be reduced to one. Each time a vertex is popped, the degree of all its adjacent vertices is decreased. If some of them turn into degree one vertices, then they are pushed onto  $VSTACK$ . The process will continue until no more vertices can have their degree values decreased. It can be seen that each time a vertex is popped, an edge in  $E(CP)$  is found that connects the vertex to some previously popped vertex. The final non-zero vertices left are just those in  $V(CC)$ . The time complexity is easily determined. Since at most  $n$  vertices will get into  $VSTACK$  and each stack operation takes constant time, steps (2) and (3) together use  $O(n)$  time.

Next, the vertices in  $V(CC)$  are subjected to an ordering in Step (4) to generate a vertex sequence  $VS(CC)$  for each  $CCY$ . In generating  $VS(CC)$ , Step (4) uses a heap  $VHEAP$  to record vertices out of which a vertex with maximal degree is always chosen as the next vertex to be put into the sequence. The usage of  $VHEAP$  is analogous to Prim's algorithm for building a minimum spanning tree. Due to the small vertex degrees in  $G$ , step (4) takes  $O(n)$  time, on average, to finish the ordering.

Based on  $VS(CC)$ , Step (5) marks all vertices in the sequence to maximize the number of direct keys in  $|K(v)| = 1$  levels, and forces all keys in  $|K(v)| > 1$  levels to be indirect. Step (5) is linear because the number of visits to vertices in  $VS(CC)$  is bounded by the total of the degree values of those vertices.

### 2.2.3 The Searching Step

The Searching step determines the  $g$  value for each vertex so as to produce an OPMPHF. The job is done in two sub-steps. First,  $g$  values for all vertices in the  $VS(CC)$  generated by the Ordering step are decided. These  $g$  values will in turn hash all keys in  $E_{CC}$  to vertices in  $AC$ s where they find their desired hash values. Then all the edges in  $E(CP)$  and  $E(AC)$  are processed to finish the searching.

A detailed description of the Searching phase is shown in Appendix C. Step (1) straightforwardly assigns  $g$  values for  $VS(CC)$ . The random probe sequence  $s_0, s_1, \dots, s_{n-1}$ , a random permutation of the set  $[0 \dots n-1]$ , gives an ordered list of potential  $g$  values to test for each vertex. Step (1) classifies three kinds of  $v_i$  in the assignment:  $|K(v_i)| = 0$ ,  $|K(v_i)| = 1$  and each  $k$  in  $K(v_i)$  is direct, or  $|K(v_i)| > 0$  otherwise. Each case is treated separately. Step (1) will use  $O(n)$  time for a successful assignment. For the rare case when there is no satisfactory  $g$  value for some vertex, we start another run of the Mapping, Ordering and

vertex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
g value	5	0	5	0	8	0	8	7	0	1	5	0	6	7	8	7
mark bit	0	0	0	1	0	1	1	0	1	1	1	1	1	1	1	1

Table 1:  $g$  Values Assignment to Vertices in Figure 7(b)

Searching steps.

Step (2) fits edges in  $CPs$ , by a depth-first traversal. The last two steps (3) and (4) are for edges in  $ACs$  with traversal root vertices either fixed during Step (1) or in  $ACs$  that have accepted no indirect edges. Step (3) can be done in linear time. Since only one edge is directly hashed during each visit of a vertex, steps (2), (3) and (4) cannot fail.

#### 2.2.4 An Example

We show in this section an example of finding an OPMPHF for the 10 key set listed in Figure 7(a) and the corresponding bipartite graph in Figure 7(b). It can be seen from Figure 7(b) that  $G$  has one  $CCY$  consisting of vertices  $V(CCY) = \{v_0, v_2, v_6, v_{10}, v_{14}\}$  and of edges  $E(CCY) = \{e_0, e_2, e_4, e_7, e_8\}$ .  $G$  also has two trees  $AC_1$  and  $AC_2$  consisting of vertices  $V(AC_1) = \{v_1, v_4, v_{13}, v_{15}\}$  and edges  $E(AC_1) = \{e_1, e_5, e_9\}$  in  $AC_1$ , and vertices  $V(AC_2) = \{v_7, v_9, v_{12}\}$  and edges  $E(AC_2) = \{e_3, e_6\}$  in  $AC_2$ .

When the Ordering phase is carried out for  $G$ , it identifies  $CCY$ ,  $AC_1$  and  $AC_2$  during Step (1) as described in Appendix B, and truncates bush edges in  $CCY$  in steps (2) and (3), leaving a sub-graph  $CC$  which has two edges  $\{e_7, e_8\}$ . In Step (4), vertices adjacent to these two edges are subject to ordering, producing a vertex sequence  $VS(CC) = [v_6, v_{14}]$ .  $VS(CC)$  is immediately involved in a marking process in Step (5), starting at  $v_6$ . Since  $K(v_6) = 0$ , we have  $v_6.bit = 1$ .  $v_{14}$  obtains the same mark (bit 1) because  $K(v_{14})$  is of size 2 and  $v_6$  has been assigned bit 1.

During the Searching phase (Appendix C),  $g$  values will be assigned first to vertices in  $VS(CC)$  in Step (1). Both  $v_6$  and  $v_{14}$  get a random number 8 so that two edges sharing vertices  $v_6$  and  $v_{14}$  can be indirectly hashed to vertices  $v_7$  and  $v_4$ . The remaining 8 edges are all direct. Vertices  $v_2$ ,  $v_{10}$  and  $v_0$  will obtain their  $g$  values in Step (2); they are all 5. Since neither  $AC_1$  nor  $AC_2$  has accepted any indirect edges, they are processed in Step (4). The final  $g$  assignment for all vertices is illustrated in Table 1. To validate the OPMPHF based on the ranking of occurrence of keys in Figure 7(a), we list the  $h$  for each key in the fifth column of Table 2.

key	$h_0$	$h_1$	$h_2$	$h(k)$
x-rays	0	0	10	$0+5+5 \pmod{10} = 0$
Euclidean	6	4	15	$6+8+7 \pmod{10} = 1$
ethyl ether	9	2	14	$9+5+8 \pmod{10} = 2$
Clouet	0	7	12	$0+7+6 \pmod{10} = 3$
Bulwer-Lytton	4	2	10	$4+5+5 \pmod{10} = 4$
dentifrice	0	4	13	$0+8+7 \pmod{10} = 5$
Lagomorpha	8	7	9	$8+7+1 \pmod{10} = 6$
Chungking	7	6	14	$7+8+8 \pmod{16} = 7, g(7) = 7$
quibbles	4	6	14	$4+8+8 \pmod{16} = 4, g(4) = 8$
Han Cities	2	1	15	$2+0+7 \pmod{10} = 9$

Table 2: The Keys from Figure 7 and Their Final Hash Addresses

### 3 Analysis and Experimental Validation

To provide further insight into our algorithm, we present analytical and experimental results in this section. In particular, section 3.1 discusses lower bound results for OPPHFs. Section 3.2 deals with characteristics of graphs, giving formulas used to compute expected values of two random variables. Their actually observed values are also listed for comparison.

#### 3.1 A Lower Bound on the Size of OPPHFs

Following the definition of a  $(N, m, n)$  perfect class of hash functions in [MEHL82], we define a  $(N, m, n)$  *order-preserving perfect class*  $H$  of OPPHFs as a set of functions  $h$

$$h : [0 .. N - 1] \rightarrow [0 .. m - 1]$$

such that for any permutation of any subset  $S$  in  $N$  of size  $|S| = n$ , there is an  $h$  in  $H$  such that  $h$  is an OPPHF for the permutation.

**Theorem.**

$$|H| \geq \frac{\binom{N}{n} n!}{\left(\frac{N}{m}\right)^n \binom{m}{n}}.$$

The proof is based on a argument similar to that found in [MEHL82], in proving the lower bound for the  $(N, m, n)$  perfect class of PHFs.

**Proof:** Clearly, there are  $\binom{N}{n}$  distinct subsets in  $[0..N-1]$ , each of size  $n$ . For each such subset  $S$ , there are  $n!$  permutations (i.e.,  $n!$  different orderings). We need to show that at most  $\left(\frac{N}{m}\right)^n \binom{m}{n}$  permutations out of the total  $\binom{N}{n} n!$  can be order preserving and hashed by a single fixed  $h$  in  $H$ .

Let  $S = \{s_1, s_2, \dots, s_n\}$  be a subset of size  $n$  that is hashed perfectly (that is, one-to-one) by  $h$ . Denote  $h(s_i)$  by  $t_i$ . Then  $T = \{t_1, t_2, \dots, t_n\}$  is one of the  $\binom{m}{n}$  subsets of  $[0..m-1]$  of size  $n$ .

We determine  $K$ , the number of subsets of size  $n$  from  $[0..N-1]$  that are hashed perfectly by  $h$  and that have image  $T$ . Let  $|h^{-1}(t_i)| = k_i$ . Then

$$K = \prod_{i=1}^n k_i$$

while

$$N \geq P = \sum_{i=1}^n k_i.$$

$K$  is maximized when all the  $k_i$ 's are equal, i.e., when  $K = (P/n)^n$ .

Now let  $T_j$ ,  $j = 1, \dots, \binom{m}{n}$  be the possible choices of  $T$ , and let  $P_j$  denote the corresponding value of  $P$ . The total number of sets that are hashed perfectly by  $h$  is

$$Q = \sum_{j=1}^{\binom{m}{n}} (P_j/n)^n$$

This is maximized when all the  $P_j$ 's are equal. This occurs when the inverse images under  $h$  of each element of  $[0..m-1]$  have equal size, that is,  $N/m$ . Then each  $P_j = Nn/m$  and

$$Q = \binom{m}{n} \left(\frac{N}{m}\right)^n.$$



From this, the theorem follows.

In the case of our OPMPHF scheme, we have  $n = m$  and  $N = nr^2$ . Thus

$$|H| \geq \frac{\binom{nr^2}{n} n!}{(r^2)^n}$$

Using the asymptotic estimate  $\binom{N}{n} \approx \frac{N^n}{n!}$

$$|H| \approx \frac{(nr^2)^n}{r^{2n}} = n^n$$

or  $\log_2 |H| \approx n \log_2 n$ . Therefore,  $O(n \log_2 n)$  bits of space are required for  $|h|$  or, equivalently, the number of  $g$  values should be at least  $n$ .

## 3.2 Characteristics of $G$

This section gives probabilistic analysis on various random variables dealing with the characteristics of  $G$ . The actual values of these measures for a particular set of random graphs will also be given after each analysis.

### 3.2.1 Average Number of Cycles

In the following, we bound the expected number of cycles in  $G$  — a bipartite graph having  $2r$  vertices on each side and having  $n$  random edges. Let  $\Pr(2i)$  be the probability of having a cycle of length  $2i$  formed in a particular vertex set of  $2i$  vertices, with  $i$  (where  $i \geq 2$ ) vertices being on each side. There are  $i!(i-1)/2$  ways to form distinct cycles out of these  $2i$  vertices and  $\binom{n}{2i}(2i)!$  ways to select  $2i$  edges to form such a cycle. The remaining  $n - 2i$  edges can go into  $G$  in  $(r^2)^{n-2i}$  different ways. Thus in total there are  $i!(i-1)/2 \cdot \binom{n}{2i} \cdot (2i)! \cdot (r^2)^{n-2i}$  ways to form a cycle of length  $2i$ . We have, given that there are a total of  $(r^2)^n$  possibilities,

$$\begin{aligned} \Pr(2i) &= \frac{\frac{i!(i-1)!}{2} \cdot \binom{n}{2i} \cdot (2i)! \cdot (r^2)^{n-2i}}{(r^2)^n} \\ &= \frac{i!(i-1)! \cdot \binom{n}{2i} \cdot (2i)!}{2r^{4i}} \end{aligned}$$

For the case  $i = 1$ , we have

$$\begin{aligned}\Pr(2) &= \frac{\binom{n}{2} \cdot (r^2)^{n-2}}{(r^2)^n} \\ &= \frac{n(n-1)}{2r^4}\end{aligned}$$

Let  $Z_{ij}$  be an indicator random variable.  $Z_{ij} = 1$  if there is a  $2i$  edge cycle in the  $j^{\text{th}}$  vertex set of  $2i$  vertices,  $Z_{ij} = 0$  otherwise. Clearly, there are  $\binom{r}{i}^2$  such sets in  $G$ . Each vertex set has the same probability of having  $2i$  edge cycles.

Let  $X_i$  be a random variable counting the number of  $2i$  edge cycles in  $G$ . We have

$$X_i = \sum_{j=1}^{\binom{r}{i}^2} Z_{ij} = \binom{r}{i}^2 \cdot \Pr(2i).$$

Define  $Y_c = \sum_{i=1}^r X_i$  as another random variable counting the number of cycles in  $G$  of length from 2 to  $\min(2r, n)$ .

$$\begin{aligned}\mathbb{E}(Y_c) &= \sum_{i=1}^r \mathbb{E}(X_i) \\ &= \sum_{i=1}^{\min(2r, n)} \binom{r}{i}^2 \cdot \Pr(2i) \\ &= \binom{r}{1}^2 \cdot \Pr(2) + \sum_{i=2}^{\min(2r, n)} \binom{r}{i}^2 \cdot \frac{i! \cdot (i-1)! \cdot (2i)!}{2 \cdot r^{4i}} \cdot \binom{n}{2i} \\ &\approx \frac{n(n-1)}{2r^2} + \sum_{i=2}^{\min(2r, n)} \left( \frac{\binom{r}{i}}{i!} \cdot e^{-\frac{i^2}{2r}} \right)^2 \cdot \frac{i! \cdot (i-1)! \cdot (2i)!}{2 \cdot r^{4i}} \cdot \left( \frac{n^{2i}}{(2i)!} \cdot e^{-\frac{(2i)^2}{2n}} \right) \\ &= \frac{n(n-1)}{2r^2} + \sum_{i=2}^{\min(2r, n)} \frac{1}{2i} \cdot \left( \frac{n}{r} \right)^{2i} \cdot e^{-i^2 \cdot (\frac{1}{r} + \frac{2}{n})} \tag{1} \\ &\leq \sum_{i=1}^{\min(2r, n)} \frac{1}{2} \left( \frac{n}{r} \right)^{2i} \\ &\leq \sum_{i=1}^{\infty} \frac{1}{2} \left( \frac{n}{r} \right)^{2i}\end{aligned}$$

Ratio	Vertices	Fraction of $G$ 's having cycles	$E(Y_c)^*$
2.2	2252	0.46	2.18
2.4	2458	0.42	0.96
2.6	2662	0.41	0.58
2.8	2868	0.31	0.39
3.0	3072	0.25	0.28
3.2	3276	0.25	0.22

Table 3: The fraction of  $G$ 's having cycles (\*computed from formula 1)

$$= \frac{1}{2} \cdot \frac{\left(\frac{n}{r}\right)^2}{1 - \left(\frac{n}{r}\right)^2}$$

Then,

$$E(Y_c) \leq \frac{1}{\left(\frac{r}{n}\right)^2 - 1}.$$

When  $\frac{r}{n} \rightarrow \infty$ , then  $E(Y_c) \rightarrow 0$ .

Table 3 shows the percentage of  $G$ 's with number of edges fixed at 1024. The number of vertices varies from 2252 to 3276 with the ratio ranging from 2.2 to 3.2.

### 3.2.2 Average Number of Trees

This subsection includes a derivation of a formula counting the number of tree components in  $G$ , excluding zero-degree vertex components. Following [AUST60], we have the number of different trees in a bipartite graph  $G'$ :

$$R_{ij} = j^{i-1} \cdot i^{j-1}$$

Here the total  $i + j$  distinct vertices are split into two groups:  $i$  vertices in one and the remaining  $j$  vertices in the other. These vertices are connected by  $i + j - 1$  indistinguishable edges to form a tree. The formula counts the number of different such trees.

No. Edges	E(TR)	Average No. of Trees
16	1.78	1.23
32	2.74	2.59
64	5.28	4.93
128	10.39	9.82
256	20.19	20.09
512	39.82	38.76
1024	79.52	79.35

Table 4: Expected vs. Average Number of Trees (ratio is set at 1.3)

The expected number of trees of distinct edges of size from 1 to  $\min(n, 2r - 1)$  in a bipartite graph  $G$  with  $r$  vertices on each side is

$$E(\text{TR}) = \sum_i \sum_j \frac{\binom{r}{i} \binom{r}{j} \cdot R_{ij} \cdot \binom{n}{i+j-1} \cdot (i+j-1)! \cdot (r^2 + i \cdot j - r \cdot (i+j))^{n-i-j+1}}{r^{2n}}$$

where  $i$  and  $j$  should satisfy the constraints  $n - i \geq 1$  and  $n - j \geq 1$  when  $i + j - 1 < n$ , or  $n - i \geq 0$  and  $n - j \geq 0$  when  $i + j - 1 = n$ . It is easy to see that the term  $\binom{r}{i} \cdot \binom{r}{j}$  is the number of ways to have all the combinations of  $i$  and  $j$  vertices on both sides. The following term  $j^{i-1} \cdot i^{j-1}$  is the number of different trees constructible from these  $i + j$  vertices. The next term  $\binom{n}{i+j-1}$  allows us to select  $(i + j - 1)$  distinct edges (keys) to participate in the tree. Since these keys are distinct, there are  $(i + j - 1)!$  ways to have the actual tree distinct. The next term  $\{r^2 + i \cdot j - r \cdot (i + j)\}^{(n-i-j+1)}$  is the number of ways to have the remaining  $n - i - j + 1$  edges freely go into  $G$  without being adjacent to any tree vertices. The last term, the denominator  $r^{2n}$ , is the total number of ways to put  $n$  edges into  $G$ .

Table 4 shows the average number of trees in  $G$  with various numbers of vertices and edges, and the expected values computed by the E(TR) formula.

### 3.2.3 Observed Number of Indirect Edges

An adequate number of indirect edges is vital to a successful OPMPHF. Table 5 summarizes the observed components, observed trees, observed number of indirect edges generated by our one bit marking scheme, observed total number of zero degree vertices, and the observed total number of trees in  $G$  generated from a CISI vector collection of 74264 keys. The different ratios for  $G$  that were tested are 1.20, 1.25, 1.3, 1.4 and 1.5.

Ratio	Components	Trees	Zero degree vertices	Indirect edges
1.20	4258	4257	16725	16536
1.25	4922	4921	18773	13914
1.30	5831	5830	20724	11789
1.40	7409	7407	24914	7365
1.50	9361	9360	29349	4510

Table 5: Number of Indirect Edges in a 74264 Edge  $G$

It can be seen that most  $G$  will have only one or a few big cycle components (see last column of Table 3) and many smaller tree components (see last column of Table 4). Notice also that the number of indirect edges varies inversely with the size of  $G$  (see Table 5). This means more edges need to be indirect as  $G$  becomes smaller. On the other hand, a small  $G$  will have fewer vertices of zero degree or in tree components. Consequently, for our scheme to be successful we have to select a  $G$  that is not too small. A rough bound on the number of allowed indirect edges (keys) for our algorithm to be successful is  $E(AC)$ , i.e., the total number of zero vertices plus the total number of trees. The maximum number of indirect edges in a particular  $G$  is the total number of edges in non-tree components.

## 4 Test Collections and Timing Statistics

Section 1.3 explains two applications of OPMPHF's for information retrieval. The first involves dictionary structures, and has led to our experimentation with dictionary key sets derived in part from the CED. Timing and other descriptive statistics from these runs with our OPMPHF algorithm are given in Table 6. On the other hand, Table 7 shows results for a set of 67,974 keys based on the inverted files data for the CISI test collection. All of these runs were made on a DECStation 5000 Model 200 in the Department of Computer Science at VPI&SU. Times were measured in seconds using the UNIX "times()" routine, and so are accurate up to 1/60th of a second.

In Table 6, we show timings for runs on graphs with different numbers of edges (varying from 32 to 16,384). We notice from Table 6 that the timing is approximately linear in the size of the key set, as is expected from our analysis. In Table 7, we list timings for runs on the 67,974 edge graph (CISI vector collection). Table 7 shows that as the ratio gets smaller (from 1.218 to 1.186), the Searching step takes more time to finish. This is because more

No.	Settings		Time in Seconds			
Keys	Ratio	Overhead	Mapping	Ordering	Searching	Total
32	1.25	2.30	0.22	0	0	0.22
64	1.23	2.45	0.22	0	0	0.22
128	1.19	2.40	0.22	0	0.02	0.24
256	1.22	2.80	0.25	0	0.02	0.27
512	1.22	3.00	0.25	0	0.05	0.30
1024	1.21	3.20	0.27	0.03	0.07	0.37
2048	1.22	3.45	0.32	0.03	0.08	0.45
4096	1.20	3.45	0.42	0.10	0.57	1.28
8192	1.19	3.60	0.63	0.25	0.92	1.80
16384	1.22	4.10	1.08	0.53	1.37	2.98

Table 6: Timing Results for Dictionary Collection

		Time in Seconds			
Ratio	Overhead	Mapping	Ordering	Searching	Total
1.186	4.0	3.67	2.28	15.87	21.82
1.193	4.1	3.72	2.27	13.92	19.90
1.199	4.2	3.62	2.25	11.20	17.07
1.205	4.3	3.72	2.25	19.05	15.02
1.211	4.4	3.73	2.27	7.90	13.90

Table 7: Timing Results for Inverted File Data

indirect edges have to be packed into a smaller number of zero-degree or tree vertices. The overhead columns in Table 6 and Table 7 shows the number of bits more than is required ( $\log n$ ) for each key.

## 5 A Second Algorithm

This section sketches a second algorithm for finding OPMPHF's with a different marking strategy, a different function and a more efficient searching step. This algorithm allows

ratio	Overhead (bits/key)	$N_{acyclic}$	$N_{indirect}$	Mapping (seconds)	Ordering (seconds)	Searching (seconds)	Total (seconds)
1.40	7.8	56743	9325	7.82	4.95	9.92	22.69
1.20	4.4	37050	21895	7.70	4.43	13.57	25.70
1.14	3.4	31846	26902	7.52	4.25	32.35	44.12
1.13	3.2	31019	27797	8.00	4.50	53.72	66.22

Table 8: OPMPHF Timing Results, Key Set Size  $n=130198$

external storage of static sets in any order desired and requires  $\approx n(\log n + 3)$  bits for the function specification. Since the lower bound is  $n \log n$ , we introduce a new metric called *OPMPHF overhead*, defined as the number of bits/key beyond the lower bound that is required to specify the hash function.

The class of functions searched is :

$$h(k) = \begin{cases} g(h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod{2r}) & \text{if } \text{mark}(h_1(k)) \text{ OR } \text{mark}(h_2(k)) \\ h_0(k) + g(h_1(k)) + g(h_2(k)) \pmod{n} & \text{otherwise} \end{cases}$$

The marking scheme simply sets the mark bit if  $|K(v_i)| > 1$  and resets it otherwise. This has the advantage of not indirecting many edges that can be treated as direct. (This differs from substep 5 in Appendix B.) The Searching step tries to assign a value for `vertex[i].g` that places all the indirect keys in  $K(v_i)$  into acyclic components. In looking for a value for `vertex[i].g`, the Searching step uses a random probe sequence to access the slots  $0, \dots, 2r - 1$  of the  $g$  table. The direct edges in these acyclic components are hashed to their desired locations as well. After hashing all indirect edges according to the vertex ordering, the rest of the direct edges in unassigned acyclic components are processed to finish the searching.

Table 8 gives results for a moderate size key set taken largely from the CED. It shows the number of acyclic components  $N_{acyclic}$ , and the number of indirect edges  $N_{indirect}$ . The timing statistics include mapping time, ordering time, searching time, and total time for various ratios. The overhead is also shown to indicate the efficiency of the algorithm.

## 6 Conclusion

In this paper, a practical algorithm for finding order-preserving minimal perfect hash functions is described. The method is able to find OPMPHF's for various sizes of key sets in almost linear time with the function size remaining within reasonable bounds. The application of the method to dictionary and inverted file construction is also illustrated. Several probabilistic analysis results on the characteristics of the random graph  $G$  are given. They are useful in guiding the proper selection of various parameters and providing insights on the design of the three main steps of the algorithm.

More experiments with the algorithm are planned. One direction is to find ways to make more edges direct so that an OPMPHF can be specified using a smaller ratio setting. Other possible interests are concerned with applications. Currently, we are using the scheme to index graph structured data. More benefits can be obtained when the scheme is applied to other fields.

Other experimentation is proceeding with a wide range of key sets. We are experimenting with a key set provided by OCLC that has more than 4 million unique keys, and so will be able to validate our approach with what are clearly very large databases.

Additional work with MPHF and OPMPHF algorithms is underway, using several related approaches. We have preliminary results regarding a MPHF method that uses much smaller function specifications and is quite fast. Subsequent papers will discuss this and other findings.

## References

- [AUST60] Austin T. L. The Enumeration of Point Labeled Chromatic Graphs and Trees. *Canadian Journal of Mathematics* **12**, 1960: 535-545.
- [BOLL85] Bollobas, B. Random Graphs. Academic Press, London, 1985.
- [CHEN90] Chen, Qi Fan. A high-performance object oriented database system for information retrieval applications. Dissertation proposal, Department of Computer Science, Virginia Polytechnic Institute & State University, 1990.
- [DAT88] Datta, S. Implementation of a Perfect Hash Function Schemes. Master's report, Department of Computer Science, Virginia Polytechnic Institute &



State University, 1988.

- [DAOUD90] Daoud, Amjad M. Efficient Data Structures for Information Retrieval Systems. Dissertation proposal, Department of Computer Science, Virginia Polytechnic Institute & State University, 1990.
- [ENBO88] Enbody, R. J. and Du H.C. Dynamic hashing schemes. *ACM Computing Surveys* **20**, 1988: 85-113.
- [FOX90] Fox, E.A., editor and project manager. Virginia Disc One. Produced by Nimbus Records, 1990. Blacksburg, VA: VPI&SU Press.
- [FOX89a] Fox, E.A., Chen, Q. F., Heath, L. and Datta, S. A More Cost Effective Algorithm for Finding Perfect Hash Functions. *Proceedings of the Seventeenth Annual ACM Computer Science Conference*, 1989, 114-122.
- [FOX89b] Fox, E.A., Heath, L.S. and Chen, Q. F. An  $O(n \log n)$  Algorithm for Finding Minimal Perfect Hash Functions. TR 89-10, Department of Computer Science, Virginia Polytechnic Institute & State University.
- [FOX88a] Fox, E.A., J. Nutter, T. Ahlswede, M. Evens, and J. Markowitz. Building a Large Thesaurus for Information Retrieval. *Proceedings Second Conference on Applied Natural Language Processing*, Austin, TX, Feb. 9-12, 1988: 101-108.
- [FOX88b] 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, **23**, 1988: 85-124.
- [FOX87] Fox, E.A. Development of the CODER System: a Testbed for Artificial Intelligence Methods in Information Retrieval. *Information Processing and Management* **23**, 1987: 341-366.
- [FOX83] Fox, E.A. Characterization of Two New Experimental Collections in Computer and Information Science Containing Textual and Bibliographic Concepts. TR 83-561, Department of Computer Science, Cornell University, Ithaca, NY, Sept. 1983.
- [FRAN89] France, R.K., E. Fox, J.T. Nutter, and Q.F. Chen. Building A Relational Lexicon for Text Understanding and Retrieval. *Proceedings First International Language Acquisition Workshop*, Aug. 21, 1989, Detroit, MI. 6 pages.

- [GARG86] Garg, Anil K. and C. C. Gotlieb Order-Preserving Key Transformations. *ACM Transactions on Database Systems*, 11(2):213-234, June 1986.
- [HANK79] Hanks, P., editor. Collins English Dictionary. William Collins Sons & Co., London, 1979.
- [MEHL82] Mehlhorn, K. G. On the Program Size of Perfect and Universal Hash Functions. *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982: 170-175.
- [NUTT89] Nutter, J.T., Fox, E.A., and Evens, M. Building a Lexicon from Machine-Readable Dictionaries for Improved Information Retrieval. *The Dynamic Text: 16th ALLC and 9th ICCH International Conferences*, Toronto, Ontario, June 6-9, 1989, revised version to appear in *Literary and Linguistic Computing*.
- [PALM85] Palmer, E. M. Graphical Evolution: An Introduction to the Theory of Random Graphs. John Wiley & Sons, New York, 1985.
- [SAGE85] Sager, T. J. A Polynomial Time Generator for Minimal Perfect Hash Functions *Communications of the ACM*, 28, 1985, 523-532.

## Appendices

### A The Mapping Phase

- | <u>Step</u> | <u>Description of Algorithm Step</u>  |
|-------------|---|
| 1.          | build random table for $h_0$ , $h_1$ and $h_2$ .  |
| 2.          | for each $v$ in $[0 \dots 2r - 1]$ do $\text{vertex}[v].\text{firstedge} = 0$ ; $\text{vertex}[v].\text{degree} = 0$  |
| 3.          | for each $i$ in $[1 \dots n]$ do<br>$\text{edge}[i].h_0 = h_0(k_i)$ ; $\text{edge}[i].h_1 = h_1(k_i)$ ; $\text{edge}[i].h_2 = h_2(k_i)$<br>$\text{edge}[i].\text{nextedge}_1 = 0$<br>add $\text{edge}[i]$ to linked list with header $\text{vertex}[h_1(k_i)].\text{firstedge}$ ; |

- increment  $\text{vertex}[h_1(k_i)].\text{degree}$
    - add  $\text{edge}[i]$  to linked list with header  $\text{vertex}[h_2(k_i)].\text{firstedge}$ ;
    - increment  $\text{vertex}[h_2(k_i)].\text{degree}$
  - 4. for each  $v$  in  $[0 \dots r - 1]$  do
    - check that all edges in linked list  $\text{vertex}[v].\text{firstedge}$  have distinct  $(h_0, h_1, h_2)$  triples.
  - 5. if triples are not distinct then repeat from step(1).

## B The Ordering Phase

Step      Description of Algorithm Step

```

order()
{
  for  $i$  in  $[0 \dots 2r - 1]$  do /*assign unique nonzero IDs to CCYs and ACs. */
  {
     $\text{vertex}[i].\text{cid} = 0$ ;
     $\text{vertex}[i].\text{CP} = 0$ ;
     $\text{vertex}[i].\text{stacked} = 0$ ;
  }

1.  $\text{cid} = 1$ ;
   for  $i$  in  $[0 \dots 2r - 1]$  do {
     if ( $\text{vertex}[i].\text{degree} > 0$ ) and ( $\text{vertex}[i].\text{cid} = 0$ ) then {
       empty(VSTACK) ; /* process one component. */
        $\text{vertex}[i].\text{stacked} = 1$  ;
       push( $i$ , VSTACK) ; /* save the first vertex of the component. */
       do {
          $j = \text{pop}(\text{VSTACK})$  ;
          $\text{vertex}[j].\text{cid} = \text{cid}$  ;
         for each  $\text{vertex}[w]$  adjacent to  $\text{vertex}[i]$  do
           /* if there are vertices unassigned, put them into VSTACK. */
           if ( $\text{vertex}[w].\text{cid} = 0$ ) and ( $\text{vertex}[w].\text{stacked} = 0$ ) then {
             push( $w$ , VSTACK) ;
           }
         }
     }
   }

```

```

        vertex[w].stacked = 1;
    } /* end of inner if */
} while notempty(VSTACK);
} /* end of outer if */
cid = cid + 1; /* increment ID for next component. */
} /* end of for */

2. for i in [0...2r - 1] do /* get all one-degree vertices into VSTACK. */
    if (vertex[i].degree = 1) then {
        push(i, VSTACK); vertex[i].stacked = 1; vertex[i].degree = 0;
    } else vertex[i].stacked = 0;

3. while notempty(VSTACK) do { /* visit and truncate all edges in E(CP). */
    i = pop(VSTACK); vertex[i].CP = 1; vertex[i].stacked = 0;
    for each vertex[w] adjacent to vertex[i] do
        if (vertex[w].degree > 0) then {
            decrement vertex[w].degree
            if (vertex[w].degree = 1) and (vertex[w].stacked = 0) then {
                push(w, VSTACK);
                vertex[i].stacked = 1;
            } /* end of if */
        } /* end of while */

4. for i in [0...2r - 1] do { vertex[i].inVS = 0; vertex[i].heaped = 0; }
    /* obtain a VS(CC) for all V(CC) vertices. */
    empty(VS);
    for i in [0...2r - 1] do {
        select a vertex vertex[i] such that:
            vertex[i].inVS = 0 and vertex[i].heaped = 0
            and vertex[i].degree is maximal
        empty(VHEAP); insert(i, VHEAP); vertex[i].heaped = 1;
        do {
            j = deletemax(VHEAP); vertex[j].inVS = 1;
            append(VS, j);
            for each vertex[w] adjacent to vertex[j] do
                if (vertex[w].inVS = 0) and (vertex[w].heaped = 0) do {

```

```

        insert(w, VHEAP);
        vertex[w].heaped = 1;
    } /* end of if */
} while notempty(VHEAP);
} /* end of for */

5. for i in [0...2r - 1] do { vertex[i].marked = 0; vertex[i].mark = 0}
    /* decide indirection bit for all vertices in V(CC) */ .
    make a traversal over VS(CC), select one vertex (vertex[i]) each time
    Let vertex[w] be any marked vertex adjacent to vertex[i]
    switch (|K(vertex[i])|) {
        case 0: vertex[i].mark = 1;
        case 1: if (vertex[w].mark = 0) then
            vertex[i].mark = 1;
        else
            vertex[i].mark = 0;
        default:
            if (vertex[w].mark = 0 for all w) then
                vertex[i].mark = 0;
            else {
                for all w do
                    if (vertex[w].mark = 0) then vertex[w].mark = 1;
                    vertex[i].mark = 1;
            }
    } /* end of switch */
    vertex[i].marked = 1;

```

## C The Searching Phase

Step	Description of Algorithm Step
------	-------------------------------

```

search()
{

```

1. *empty*(*R*); *empty*(*S*)
  - /\* S is the set of cid's of those occupied trees. \*/*
  - /\* R records the root vertices of trees in S. \*/*
  - for *i* in  $[0 \dots 2r - 1]$  do { *vertex*[*i*].*occupied* = 0; *vertex*[*i*].*fitted* = 0;}
    - establish a probe sequence  $s_0, s_1, \dots, s_{n-1}$  where each  $s_j$  is a random number in set  $\{0 \dots n - 1\}$ . All  $s_j$  are distinct and put in the sequence randomly.

Let *i* be the index of the first element in *VS*(*CC*)

```

do {
  j = 0
  do forever {
    vertex[i].g =  $s_j$  ;
    if (fit(i) = false) then /* if this  $s_j$  can't fit  $K(\text{vertex}[i])$ , try next  $s_j$  */
      j = j + 1;
      if (j = n) then { search = fail; return; }
    } else vertex[i].fitted = 1
  }
  i = succeeding vertex array index in VS(CC);
} while (i ≠ undefined)
2. process_CPs(); /* process CP */
3. process_R(); /* process components that have accepted indirected keys */
4. process_ACs(); /* process components that haven't accepted any indirected keys */
   search = success; return;
}

```

```

fit(i) /* fit keys in  $K(\text{vertex}[i])$  into hash table */
{

```

Let *vertex*[*w*] be any vertex adjacent to *vertex*[*i*] such that *vertex*[*w*].*fitted* = 1;

switch ( $|K(\text{vertex}[i])|$ ) {

case 0: return true;

default:

if ( $(|K(\text{vertex}[i])| = 1)$  and (*vertex*[*w*].*mark* ≠ *vertex*[*i*].*mark*) then {

*/\* direct hash case \*/*

let  $K(\text{vertex}[i]) = \{k\}$

```

    vertex[i].g = {edge[k].order - edge[k].h0 - vertex[w].g} mod n;
    return true;
} else { /* indirect hash case */
    for each k in K(vertex[i]) do /* test collision */
        if ((vertex[h(k)].cid = 0) and (vertex[h(k)].occupied = 1))
            or (vertex[h(k)].cid in S) then
                return false;

        /* no collision, assign desired hash addresses to accepting vertices */
    for each k in K(vi) do {
        h(k) = {edge[k].h0 + vertex[edge[k].h1].g + vertex[edge[k].h2].g} mod 2r
        if (vertex[h(k)].cid = 0) then
            vertex[h(k)].occupied = 1;
        else {
            S = S ∪ {vertex[h(k)].cid};
            R = R ∪ {vertex[h(k)]};
        }
        vertex[h(k)].g = edge[k].order;
    } /* end of inner if */
    return true;
} /* end of outer if */
} /* end of switch */
}

process_CPs() /* process CP edges */
{
    empty(VSTACK);
    for i in [0...n-1] do vertex[i].stacked = 0

    for i in [0...n-1] do {
        if (vertex[i].fitted = 1) and (vertex[i].CP = 1) and
            (vertex[i].degree > 0) then
            /* identify starting vertices */
            for (all vertex[w] adjacent to vertex[i] such that vertex[w].CP = 1) and
                (vertex[w].stacked = 0) do {
                push(w, VSTACK);
            }
        }
    }
}

```

```

        vertex[w].stacked = 1;
    }
    while notempty(VSTACK) do { /*directly hash all bush edges */
        j = pop(VSTACK);
        vertex[j].traversed = 1;
        for all vertex[w] adjacent to vertex[j] do
            if (vertex[w].fitted = 1) then {
                let k join vertex[w] and vertex[j];
                vertex[j].g = {edge[k].order - edge[k].h0 - vertex[w].g} mod n;
            } else
                if (vertex[w].stacked = 0) then {
                    push(w, VSTACK);
                    vertex[w].stacked = 1;
                }
        } /* end of while */
    } /* end of for */

```

```

process_R()
{
    Same as process_CPs(). Each vertex in R will act as i.
}

```

```

process_ACs()
{
    Same as process_CPs(). Each vertex that has not accepted
    any indirected edge will act as i.
}

```