





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

3 4

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER <b>AFOSR-TR- 83-0960</b>		2. GOVT ACCESSION NO. <b>A135299</b>	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) <b>THE PROGRAM COMPLEXITY OF SEARCHING A TABLE</b>		5. TYPE OF REPORT & PERIOD COVERED <b>TECHNICAL</b>	
7. AUTHOR(s) <b>Harry G. Mairson</b>		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS <b>Department of Computer Science Stanford University Stanford CA 94305</b>		8. CONTRACT OR GRANT NUMBER(s) <b>AFOSR-80-0212</b>	
11. CONTROLLING OFFICE NAME AND ADDRESS <b>Mathematical &amp; Information Sciences Directorate Air Force Office of Scientific Research / JMK Bolling AFB DC 20332</b>		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS <b>PE61102F; 2304/A2</b>	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE <b>OCT 83</b>	
		13. NUMBER OF PAGES <b>15</b>	
		15. SECURITY CLASS. (of this report) <b>UNCLASSIFIED</b>	
16. DISTRIBUTION STATEMENT (of this Report) <b>Approved for public release; distribution unlimited.</b>		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		S DTIC ELECTED DEC 2 1983	
18. SUPPLEMENTARY NOTES		B	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) <b>SEE REVERSE</b>			

A135299

DTIC FILE COPY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20:

Does the point X belong to the set S?

n times the log e + the base 2 (1+o(1))

ABSTRACT. Given a fixed set  $S$  of  $n$  keys, we would like to store them so that queries of the form " $x \in S$ " can be answered quickly. A commonly employed scheme to solve this problem uses a table to store the keys, and a special purpose program depending on  $S$  which probes the table. We analyze the tradeoff between the maximum number of probes allowable to answer a query, and the information-theoretic complexity of the program to do so. Perfect hashing (where the query must be answered in one probe) has a program complexity of  $(n \log_2 e (1 + o(1)))$  bits, and this lower bound can be achieved. Under a model combining perfect hashing and binary search methods, it is shown that for  $k$  probes to the table,  $nk / 2^{k-1} (1 + o(1))$  bits are necessary and sufficient to describe a table searching algorithm. This model gives some information-theoretic bounds on the complexity of searching an external memory. We examine some schemes where pointers are allowed in the table, and show that for  $k$  probes to the table, about  $\frac{n \log_2 e}{k+1} (1 + o(1))$  bits are necessary and sufficient to describe the search. Finally, we prove some lower bounds on the worst case performance of hash functions described by bounded Boolean circuits, and worst case performance of universal classes of hash functions.

$nk / 2^{k-1}$  + the  $k+1$  power  $(1+o(1))$

Accession For	
NTIS GR&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

**AFOSR-TR- 83 - 0960**

**The Program Complexity of Searching a Table**

**HARRY G. MAIRSON**  
*Department of Computer Science*  
*Stanford University*  
*Stanford, California 94305*

**ABSTRACT.** Given a fixed set  $S$  of  $n$  keys, we would like to store them so that queries of the form "Is  $x \in S$ ?" can be answered quickly. A commonly employed scheme to solve this problem uses a table to store the keys, and a special purpose program depending on  $S$  which probes the table. We analyze the tradeoff between the maximum number of probes allowable to answer a query, and the information-theoretic complexity of the program to do so. *Perfect hashing* (where the query must be answered in one probe) has a program complexity of  $n \log_2 e(1 + o(1))$  bits, and this lower bound can be achieved. Under a model combining perfect hashing and binary search methods, it is shown that for  $k$  probes to the table,  $nk/2^{k+1}(1 + o(1))$  bits are necessary and sufficient to describe a table searching algorithm. This model gives some information-theoretic bounds on the complexity of searching an external memory. We examine some schemes where pointers are allowed in the table, and show that for  $k$  probes to the table, about  $\frac{n \log_2 e}{(k+1)!}(1 + o(1))$  bits are necessary and sufficient to describe the search. Finally, we prove some lower bounds on the worst case performance of hash functions described by bounded Boolean circuits, and worst case performance of universal classes of hash functions.

This paper will appear in the IEEE Symposium on Foundations of Computer Science, November 1983.

Approved for public release;  
distribution unlimited.

**83 11 29 221**

1. Introduction

MATTHEW J. KERPER  
Chief, Technical Information Division

§1 Introduction

Given a fixed set  $S$  of  $n$  keys, we would like to store them so that *membership queries* of the form "Is  $x \in S$ ?" can be answered quickly. This *searching problem*, particularly in the static case, is clearly among the most fundamental of data structuring problems, as well as being ubiquitous in computer science applications.

This simple information-retrieval problem has generated considerable interest in recent years. The papers [Sprugnoli] and [Jaeschke] suggest several *ad hoc* hashing schemes to implement a solution. Jaeschke recommends using a hash function of the form  $h(x) = \lfloor \frac{Ax}{Bx+C} \rfloor \pmod{n}$ , where the constants  $A$ ,  $B$ , and  $C$  depend on  $S$ . This function is called "perfect" as no keys in  $S$  collide under  $h$ , and  $h$  has a very reasonable unit-cost arithmetic complexity. Its *program complexity*, on the other hand, is very large, as the number of bits needed to write  $A$ ,  $B$ , and  $C$  is  $\Omega(n)$ . By contrast, binary search has a program complexity of  $O(\log n)$  and answers queries in  $O(\log n)$  probes. This relation between *search time* (measured in number of probes) and *program complexity* (measured in bits) suggests that there should be an information-theoretic tradeoff between the two. The tradeoff intuitively corresponds to the relationship of the *performance* of a search strategy to the inherent complexity of its *description*. The relationship has appeared in a variety of disguises, including the following:

1. **Searching External Memories.** Gonnet and Larson recently examined the problem of searching an external memory with limited internal storage [Gonnet and Larson]. Since accessing external memory is very time-consuming, methods which reduce the number of accesses are very desirable, even if they increase internal processing. Gonnet and Larson examine external hashing techniques assuming random probing, where a small amount of internal storage (a *directory*) is used to help direct the search. How does the size of the directory affect the efficiency of the search? If membership queries are answered by retrieving pages of the external memory, our above described tradeoff represents the relationship between the directory size and the number of uniform-size pages needed to store  $S$ . We determine this tradeoff.

2. **Internal Searching in a constant number of probes.** Several investigations have been made in this direction, including [Tarjan and Yao] and more recently [Fredman et al.] Both of their results show practical schemes for answering queries in  $O(1)$  probes with a program complexity of  $O(n \log n)$ . Another paper [Yao] demonstrates a "canonical 2-probe structure" which always probes first to a table position containing a directory for the rest of the table, and uses information in the directory to choose the next probe. Again, the size of the directory is the program complexity of the search strategy. We analyze the worst case of hashing with separate chains, and determine the tradeoff between program complexity and chains of maximum length  $k$ , so queries are answered in no more than  $k$  probes.

3. **Probabilistic Hashing and Hash Circuits.** Recent work on probabilistic hashing by Carter and Wegman suggests choosing a hash function at random from a class of functions  $\mathcal{X}$  having a "universal" property [Carter and Wegman]. The property guarantees certain desirable *input independent* expected bounds on search time (measured in number of probes), randomizing over the choice of function. The program complexity (roughly  $\log_2 |\mathcal{X}|$ ) of these hash functions was analyzed in [Mehlhorn]. It is of interest to minimize  $\log_2 |\mathcal{X}|$  for several reasons: note that  $\log_2 |\mathcal{X}|$  is a measure of how many coin flips are needed to randomly choose a function from  $|\mathcal{X}|$ , which becomes important in some applications. We show that for such minimal  $\mathcal{X}$ , there are some worst-case input dependent lower bounds on search time

of  $\Omega\left(\frac{\log n}{\log \log n}\right)$ . Similar lower bounds can be derived for bounded-size Boolean hash circuits.

Program complexity (see [Chaitin]) is a measure which has largely been applied to problems in formal systems, "machine based" recursive function theory, and information theory. We intend to use it as a tool in concrete complexity, applying it to a particular information retrieval problem. As long as we assume a basic random access machine (RAM) model, this measure of program complexity is fundamentally independent of language implementation, so that it does not matter whether a search strategy to answer membership queries is described in assembly language, PASCAL, or French.

## §2 Model and Notation

The problem is formalized as follows. We construct a special purpose computer program  $P$  depending on  $S$  and store the elements of  $S$  in a table, possibly with some associated pointer structure. To answer "Is  $x \in S$ ?"  $P$  is allowed to probe the table in its search for  $x$ , and make auxiliary computations (following pointers if they are permitted) between probes. If  $P$  finds  $x$  in the table, it answers "yes." If it does not find  $x$ , but gathers sufficient information to certify that  $x$  is not in the table, it answers "no." In addition, we use the following notation:

$M = \{0, 1, \dots, m-1\}$ . The key space.

$M^{(n)}$ . The subsets of  $M$  of cardinality  $n$ .

$S \in M^{(n)}$ . The set in question. We assume that  $M$  is much larger than  $S$ , which is reasonable in most applications.

$N = \{0, 1, \dots, n-1\}$ . The address space of the table.

$P^{(n)}$ . The partitions of  $M$  into  $n$  parts (possibly empty parts).

$P$ . We use  $P$  ambiguously to denote both partitions of  $M$  as well as program encodings of search strategies, because there is a direct and significant correspondence between the two. Which meaning of  $P$  is intended should be clear from context.

## §3 Hashing and Partitions: Combinatorial Preliminaries

Every hash function program  $P : M \rightarrow N$  induces a partition of the key space  $M$  into parts  $M_i$ , where  $\bigcup_i M_i = M$  and  $M_i = \{x \in M | P(x) = i\}$ . We can examine a *partition property*  $A(P, S)$  of a program (partition)  $P$  and a set  $S$ , and determine whether or not the property is satisfied. For example, "no two elements of  $S$  are found in the same part of the partition defined by  $P$ " (the *perfect hashing property*), or "no more than  $k$  elements of  $S$  are found in the same part of the partition defined by  $P$ ." Given any set  $S \in M^{(n)}$  we want a program  $P$  such that  $A(P, S)$  holds. Define  $C(A)$  as the bit complexity of the program (partition)  $P$  satisfying  $A(P, S)$ .

We can then generate lower bounds through the following counting argument:

**Theorem 1.** Let

$$Q_A = \max_{P \in P^{(n)}} |\{S \in M^{(n)} | A(P, S) \text{ holds}\}|.$$

Then for any  $S \in M^{(n)}$ , the bit complexity of a program satisfying  $A(P, S)$  is bounded below by

$$C(A) \geq \log_2 \left[ \binom{m}{n} / Q_A \right].$$

The following probabilistic argument lets us construct upper bounds.

**Theorem 2.** Let  $A(P, S)$  be a partition property over the class  $P^{(n)}$  of partitions, where

$$\Pr \{A(P, S) \text{ holds}\} \geq p(m, n).$$

Then there exists a set  $\mathcal{X} = \{P_1, P_2, \dots, P_k\} \subseteq P^{(n)}$  of  $k$  partitions where  $k = \frac{n \ln m}{p}$ , and the following is true: for any  $S \in M^{(n)}$  there exists a  $P \in \mathcal{X}$  such that  $A(P, S)$  holds.

**Proof.** Suppose we choose the partitions in  $\mathcal{X}$  independently at random. Let  $A_S, S \in M^{(n)}$  be the event

$$\bigwedge_{P \in \mathcal{X}} (A(P, S) \text{ does not hold}).$$

Then

$$\Pr \{A_S\} \leq (1-p)^k,$$

and

$$\Pr \left\{ \bigvee_{S \in M^{(n)}} A_S \right\} \leq \sum_{S \in M^{(n)}} \Pr \{A_S\} \leq \binom{m}{n} (1-p)^k.$$

If

$$\binom{m}{n} (1-p)^k < 1 \quad (*)$$

then it is possible to deterministically choose an  $\mathcal{X}$  making  $\bigvee_{S \in M^{(n)}} A_S$  false, and

$$\overline{\bigvee_{S \in M^{(n)}} A_S} = \bigwedge_{S \in M^{(n)}} (\exists P \in \mathcal{X}) (A(P, S) \text{ holds}),$$

which would prove the theorem. Since  $kp = n \ln m$ , we know

$$k \ln(1-p) + n \ln m < -kp - \frac{kp^2}{2} + n \ln m = -\frac{kp^2}{2} < 0,$$

or  $m^n(1-p)^k < 1$ , which implies (\*), proving the theorem. ■

We shall see that if  $P$  is chosen randomly from  $P^{(t)}$ , and  $\Pr \{A(P, S) \text{ holds}\} \geq p(m, n)$ , then Theorem 2 will let us show upper bounds of  $C(A) \leq \log_2 \frac{n \ln m}{p} + O(\log n + \log \log m)$ .

#### §4 Some Applications

We first examine *perfect hashing*, where  $P$  may probe exactly one table location in its search to answer the query, "Is  $x \in S$ ?" If  $x$  appears there,  $P$  answers "yes"; if  $x$  does not appear,  $P$  must be justified in its saying "no" without fear of the appearance of  $x$  somewhere else in the table. We would like to know exactly how many bits are needed to define such a perfect hash function for every  $S \in M^{(n)}$ , denoted  $PHF(m, n)$ .

**Theorem 3.**

$$PHF(m, n) \geq n \log_2 e - \frac{1}{2} \log_2 2\pi n + O\left(\frac{1}{n} + \frac{n^2}{m}\right).$$

**Proof.** We use the counting argument of Theorem 1. Let  $P$  be a program which is a perfect hash function. Think of  $P$  as a function  $P: M \rightarrow N$  which maps each key in the key space onto a particular address in the table.  $P$  then induces a partition of  $M$  into parts  $M_0, M_1, \dots, M_{n-1}$ , where  $\bigcup_i M_i = M$  and  $M_i = \{x \in M | P(x) = i\}$ . If  $P$  is a perfect hash function for  $S$ , clearly no two keys in  $S$  can be in the same  $M_i$ , since  $P$  restricted to  $S$  gives the different addresses of the elements of  $S$  in the table. Define

$$\text{Perf}(P) = |\{S \in M^{(n)} | P \text{ is a perfect hash function for } S\}|.$$

Let  $\mathcal{X}$  be a set of search programs such that for any  $S \in M^{(n)}$ , there exists a  $P \in \mathcal{X}$  which is perfect for  $S$ . Then by counting,

$$\left(\max_{P \in \mathcal{X}} \text{Perf}(P)\right) |\mathcal{X}| \geq \binom{m}{n} = \text{number of sets of size } n.$$

If  $P$  partitions  $M$  into parts  $M_0, M_1, \dots, M_{n-1}$  then clearly  $\text{Perf}(P) = \prod_{0 \leq i \leq n-1} |M_i|$ , and this quantity is maximized when each  $M_i$  is of equal size, so that  $|M_i| = \frac{m}{n}$ . Then

$$\left(\frac{m}{n}\right)^n |\mathcal{X}| \geq \binom{m}{n} = \frac{m^n}{n!},$$

or

$$|\mathcal{X}| \geq \frac{n^n}{n!} \left(1 + O\left(\frac{n^2}{m}\right)\right).$$

Applying Stirling's formula, taking logarithms, and assuming  $n^2 = o(m)$ , we obtain the desired lower bound: clearly  $\log_2 |\mathcal{X}|$  bits are necessary or else we cannot uniquely identify all the programs in  $\mathcal{X}$ . ■

The counting argument of Theorem 3 shows that sets  $S \in M^{(n)}$  exist with perfect hash functions that have complexity of at least  $n \log_2 e (1 + o(1))$ . Of course, some sets, such as  $\{0, 1, 2, \dots, n-1\}$ , have perfect hash functions of low complexity. However, the counting argument, as in Chaitin-Kolmogorov complexity, is quite strong in the following sense: no more than  $2^{-k} \binom{m}{n}$  of the sets  $S \in M^{(n)}$  can have perfect hash functions with program complexity less than  $PHF(m, n) - k$ . Therefore, both in Theorem 3 and in counting arguments we will see later, a lower bound of  $C$  on program complexity for perfect hashing, or analogous search properties, indicates that at least half of the sets  $S \in M^{(n)}$  have program complexity of at least  $C - 1$ , etc.

**Theorem 4.**

$$PHF(m, n) \leq n \log_2 e + \frac{3}{2} \log_2 n + 2 \log_2 \log_2 m + O(1).$$

**Proof.** Let  $A(P, S)$  be the perfect hashing property. We know that

$$\Pr \{A(P, S) \text{ holds}\} = \frac{n! n^{m-n}}{n^m} = \frac{n!}{n^n}.$$

By Theorem 2, we know it is possible to choose a set  $\mathcal{X}$  of partitions so that every  $S \in M^{(n)}$  has a perfect hash function chosen from  $\mathcal{X}$ , where

$$|\mathcal{X}| = \frac{n^n}{n!} n \ln m,$$

so that

$$|\mathcal{X}| = \sqrt{\frac{n}{2\pi}} e^n \ln m \left(1 + O\left(\frac{1}{n}\right)\right).$$

To prove the theorem, it only remains to show that a program computing a partition in  $\mathcal{X}$  can indeed be written in about  $\log_2 |\mathcal{X}|$  bits. This can be done using a variant of the algorithm suggested in [Mehlhorn], which we now describe. The idea of the program is that every class of hash functions can be specified by an  $m \times |\mathcal{X}|$  matrix  $M$ , where  $M_{i,j} = h_j(i)$ , that is, the  $j$ -th hash function in  $\mathcal{X}$  applied to  $i$ . It turns out that very short programs can enumerate these matrices, and check them for properties like perfect hashing.

*PROGRAM Perfect<sub>S</sub>(x):*

1.  $b \leftarrow \lceil \log_2 m \rceil$  written in binary;
2.  $j \leftarrow$  some number between 1 and  $|\mathcal{X}|$  depending on  $S$ ;
3. Search through all  $2^b \times 1, 2^b \times 2, 2^b \times 3, \dots$  matrices in some lexicographic order with entries in  $\{0, \dots, n-1\}$  until a "perfect matrix"  $M$  is found. Column  $j$  of  $M$  represents the perfect hash function for  $S$ : probe address  $M_{x,j}$  of the table. If  $x$  appears there, return "yes," otherwise return "no."

The length of the above program is  $\log_2 \log_2 m$  for step 1, at most  $\lceil \log_2 |\mathcal{X}| \rceil$  for step 2, and  $\log_2 n + O(1)$  for step 3, which proves the theorem. Note the same perfect matrix is

always found, since the matrices are enumerated in the same lexicographic order: this is why  $j$  can in principle be determined beforehand. ■

Theorems 3 and 4 give a  $\Omega(n + \log_2 \log_2 m)$  lower bound on the program complexity of perfect hash functions. Whether  $n$  or  $\log_2 \log_2 m$  is the significant term is dependent on the relative asymptotic growth of  $n$  and  $m$ . We will assume throughout our discussion that  $m$  asymptotically grows faster than  $n$ , but not too fast, so that the  $n$  dominates the asymptotics. More specifically, we will assume  $n^2 = o(m)$  and  $\log_2 \log_2 m = o(n)$ . These bounds allow for analysis of "intermediate" values of  $m$  and  $n$  such as  $m = 2^{\sqrt{n}}$ , suggested as an open problem in [Yao].

The lower and upper bounds described by Theorems 3 and 4 have separately and independently appeared in various places, including [Mehlhorn], [Berman et al.], and [Fredman et al.].

Now let's look at searching an external memory. Suppose a page of external memory stores exactly  $k$  keys. Decompose  $S$  into  $n/k$  pages  $B_i$ ,  $1 \leq i \leq n/k$ . Membership queries will now be answered by examining a *directory* in the internal memory containing enough information to determine in which page  $x$  is found if  $x$  is indeed in  $S$ . (If the keys in each  $B_i$  are sorted, the relevant page can be pulled and binary searched in  $\lceil \log_2(k+1) \rceil$  probes.) Let  $HF_k(m, n, t)$  denote the bit complexity of the most concise such directory, where  $t$  is the size of the table.

**Theorem 5.**

$$HF_k(m, n, n/k) \geq \frac{n \log_2 2\pi k}{2k} - \frac{1}{2} \log_2 2\pi n + O\left(\frac{n}{k^2} + \frac{n^2}{m} + \frac{1}{n}\right).$$

The above theorem has a corresponding upper bound:

**Theorem 6.**

$$HF_k(m, n, n/k) \leq \frac{n \log_2 2\pi k}{2k} + \frac{3}{2} \log_2 n + 2 \log_2 \log_2 m - \frac{1}{2} \log_2 2\pi + O\left(\frac{n}{k^2} + \frac{1}{n}\right).$$

**Proof.** We proceed as in Theorem 4. Let  $A(P, S)$  be the property " $P$  partitions  $S$  into  $n/k$  parts with exactly  $k$  elements of  $S$  in each part." Then for fixed  $S$  and randomly chosen  $P$ ,

$$\begin{aligned} \Pr \{A(P, S) \text{ holds}\} &= \left(\frac{n}{k}\right)^{-n} \binom{n}{k \ k \ \dots \ k} \\ &\geq \frac{\sqrt{2\pi n}(1 + O(\frac{1}{n}))}{(\sqrt{2\pi k}(1 + O(\frac{1}{k})))^{n/k}} = p. \end{aligned}$$

By Theorem 2, we can choose a set  $\mathcal{X}$  of partitions so that every  $S \in M^{(n)}$  has a partition  $P \in \mathcal{X}$  satisfying  $A$ , where

$$\begin{aligned} |\mathcal{X}| &= \frac{n \ln m}{p} \\ &= \sqrt{\frac{n}{2\pi}} \ln m \left(\sqrt{2\pi k} \left(1 + O\left(\frac{1}{k}\right)\right)\right)^{n/k} \left(1 + O\left(\frac{1}{n}\right)\right). \end{aligned}$$

The following program will satisfy  $A$  for  $S$  and takes  $\log_2 |X| + \log_2 n + \log_2 \log_2 m + O(1)$  bits to encode.

PROGRAM AlmostPerfect $_S$  ( $x$ ):

1.  $b \leftarrow \lceil \log_2 m \rceil$  written in binary;
2.  $j \leftarrow$  some number between 1 and  $|X|$  depending on  $S$ ;
3. Search through all  $2^b \times 1, 2^b \times 2, 2^b \times 3, \dots$  matrices in some lexicographic order with entries in  $\{0, 1, \dots, n/k - 1\}$  until a matrix  $M$  is found satisfying property  $A$ .  $M_{x,j}$  gives the block address  $B_x$  of  $x$ .
4. Binary search block  $B_x$  in  $\log_2(k+1)$  probes, and return "yes" or "no" depending on whether  $x$  is found in the block. ■

By trading off binary search and perfect hashing, then,  $\Theta(nt/2^t)$  bits are necessary and sufficient to encode a program  $P$  answering membership queries in  $t$  probes to the table.

### §5 Hashing with Separate Chains

Now suppose each table slot is allowed to hold a pointer to another table address as well as holding a key. We would like to know how this additional information in the table can be used to optimize the worst-case number of probes to the table.

Let each search program  $P$  initially probe one table location to answer the query "Is  $x \in S$ ?" If  $x$  is found at that location,  $P$  answers "yes." Otherwise it follows a chain of pointers until  $x$  is found on the chain, or the end of the chain is reached, answering "yes" or "no" accordingly. This scheme is intended to model the static case of hashing with separate chains. The static nature of the problem allows the folding of chains into the table, so no additional memory is needed.

We assume that static table schemes always consist of separate chains of pointers, where the program initially probes to the first key in the chain, and then follows pointers. This assumption is not restrictive in terms of finding optimal search programs and associated pointer structures.

To answer a query in  $k$  probes, it means that no chain in the table can be more than  $k$  keys long. Each program and table structure for  $S \in M^{(n)}$  now corresponds to a partition of  $M$  in which no more than  $k$  keys in  $S$  appear in any part. The analysis of this model is considerably more difficult than the pointer-free models we have already examined, as the counting problems involved are much more complicated, and we must asymptotically approximate their solution.

Let  $H_k(m, n)$  denote the number of bits required to define a search strategy as described earlier, where no chain of pointers in the table has length greater than  $k$ .

**Theorem 7.**

$$H_k(m, n) \geq n \log_2 \left( \frac{m}{1 + \sqrt{k}} \right) - \frac{1}{2} n \left( 1 + \frac{1}{\sqrt{k}} \right) + O \left( \frac{n^2}{m} + n^{-1 + 3k} \right).$$

**Proof.** (sketch) Let  $\mathcal{A}(P, S)$  be the property " $P$  partitions  $M$  into  $n$  parts with at most 2 elements of  $S$  found in any part." If  $P$  partitions  $M$  into parts of size  $p_i$ ,  $1 \leq i \leq n$ , then

$$\begin{aligned} Q_P &= \{S \in M^{(n)} \mid \mathcal{A}(P, S) \text{ holds}\} \\ &= (z^n) \prod_{1 \leq i \leq n} \left(1 + p_i z + \binom{p_i}{2} z^2\right), \end{aligned}$$

and it is not difficult to show that

$$\begin{aligned} Q_A &= \max_{P \in \mathcal{P}(n)} \{S \in M^{(n)} \mid \mathcal{A}(P, S) \text{ holds}\} \\ &= \left(\frac{m}{n}\right)^n (z^n) \left(1 + z + \frac{z^2}{2}\right)^n \\ &\leq \left(\frac{5m}{2n}\right)^n (z^n) \left(\frac{2}{5} + \frac{2}{5}z + \frac{1}{5}z^2\right)^n \\ &\leq \left(\frac{5m}{2n}\right)^n (z^n) g(z)^n, \end{aligned}$$

where  $(z^n)g(z)^n$  denotes the coefficient of  $z^n$  in  $g(z)^n$ . The function  $g(z)$  is a probability mass function, so that the coefficients of  $g(z)^n$  represent the distribution of sums resulting from  $n$  trials of the random variable described by  $g(z)$ . The coefficient of interest may then be recovered by use of the Local Limit Theorem for lattice distributions, a discrete and local form of the Central Limit Theorem (see [Feller], [Petrov]):

$$(z^{\mu n + r})g(z)^n = \frac{1}{\sigma\sqrt{2\pi n}} \exp\left(\frac{-r^2}{2\sigma^2 n}\right) + O(n^{-1+3\epsilon}),$$

where  $\mu$  and  $\sigma^2$  are the mean and variance of  $g(z)$ , and  $\epsilon$  is some small constant greater than zero. Since  $\mu \neq \sigma^2$  and  $\mu$  and  $\sigma^2$  are both constant, the above approximation is useless except very close to the mean (i.e., for small  $r$ ); otherwise the  $O(n^{-1+3\epsilon})$  term swamps everything. However, this situation can be remedied by the technique of *shifting the mean*, described in [Greene and Knuth]. We introduce a parameter  $\alpha$ , and note

$$\begin{aligned} (z^n)g(z)^n &= \left(\frac{g(\alpha)}{\alpha}\right)^n (z^n) \left(\frac{g(\alpha z)}{g(\alpha)}\right)^n \\ &= \left(\frac{g(\alpha)}{\alpha}\right)^n (z^n) G(z)^n. \end{aligned}$$

If we let  $\alpha = \sqrt{2}$ , then  $\text{mean}(G) = 1$ , and the Local Limit Theorem will provide the required asymptotic information, since we will be asking about the distribution precisely at the mean, in which case

$$(z^n)g(z)^n = \left(\frac{2\sqrt{2} + 2}{5}\right)^n \left[\frac{\sqrt{2} + \sqrt{2}}{2\sqrt{\pi n}} + O(n^{-1+3\epsilon})\right].$$

Inserting this value into the inequality  $Q_A \mid \mathcal{X} \mid \geq \binom{n}{m}$  gives the lower bound. ■

A tight upper bound can be proven using the nonconstructive argument, which we give without proof:

**Theorem 8.**

$$H_2(m, n) \leq n \log_2 \left( \frac{e}{1 + \sqrt{2}} \right) + 2 \log_2 n + \log_2 \log_2 m + O(1).$$

Probability, information, and combinatorial theory share a variety of asymptotic counting techniques. To prove Theorems 7 and 8, we use the Local Limit Theorem for counting, and not because of its relation to anything probabilistic. Given a combinatorial generating function  $f(z) = \sum f_n z^n$  which is free of singularities, the Residue Theorem from complex variable theory can be used to determine particular coefficients:  $(z^k)f(z) = \frac{1}{2\pi i} \oint \frac{f(z)}{z^{k+1}} dz$ . One proof of the Local Limit Theorem uses precisely this technique: we assume  $f(z) = g(z)^n$  where  $g(z)$  is a probability mass function, and the path of integration is the unit circle on the complex plane. Intuitively, the unit circle path tends to "focus" the value of the integral at the mean of  $g(z)^n$ . The method of shifting the mean is a heuristic which allows the saddle point of the integrand to be "moved" to an advantageous place on the path.

We now generalize the above methods to prove a tradeoff for arbitrary  $k$ .

**Theorem 9.**

$$H_k(m, n) = \frac{n \log_2 e}{e} \left( \left( \sum_{j \geq k+1} \frac{1}{j!} \right) + O\left(\frac{k}{k!^2}\right) \right) + O(\log n + \log \log m).$$

**Proof.** (sketch) Let  $A(P, S)$  be the property " $P$  partitions  $M$  into  $n$  parts with no more than  $k$  elements of  $S$  found in any part." The quantity  $Q_A$  defined in Theorem 1 (and needed to determine a lower bound) can be bounded as

$$Q_A \leq \left(\frac{m}{n}\right)^n (z^n) \left(1 + z + \frac{z^2}{2!} + \cdots + \frac{z^k}{k!}\right)^n.$$

In showing an upper bound, we find that for randomly chosen  $P \in P^{(n)}$ ,

$$\Pr \{A(P, S) \text{ holds}\} = \frac{n!}{n^n} (z^n) \left(1 + z + \frac{z^2}{2!} + \cdots + \frac{z^k}{k!}\right)^n.$$

Surprisingly, the same generating function, given different combinatorial interpretations, appears in both the upper and lower bounds. The appearance of the above generating function in the lower bound is related to the fact that  $Q_A$  is maximized by partitions of the key space into equal sized parts. This equipartition property, which gives a bound on the entropy of the search program, is analogous to the equipartition property of information theory which maximizes entropy in source coding.

To recover the coefficient of interest in the above truncated exponential, denoted as  $G(z)$ , we proceed as follows:

- $\langle z^n \rangle G(z)^n$  is replaced by  $G(1)^n \langle z^n \rangle g(z)^n$ , where  $g(z) = G(z)/G(1)$  is now a probability mass function.  $\langle z^n \rangle g(z)^n$  is then rewritten as  $\langle z^n \rangle g(z)^n = \left(\frac{g(\alpha)}{\alpha}\right)^n \langle z^n \rangle \bar{g}(z)^n$  where  $\bar{g}(z) = g(\alpha z)/g(\alpha)$ .
- By the Residue Theorem,  $\langle z^n \rangle \bar{g}(z)^n = \frac{1}{2\pi i} \oint_{\frac{\beta}{z^{n+1}}} \bar{g}(z)^n dz$ . We select as a contour of integration the path  $z = e^{it}$  around the pole at  $z = 0$ . We now choose  $\alpha$  so that  $\alpha g'(\alpha) = g(\alpha)$ , which shifts the mean of  $\bar{g}$  to 1, and thus the saddle point of the contour integral to  $z = 1$ . To choose  $\alpha$  satisfying these constraints, we must find the positive real zero of a polynomial of degree  $k$ , which (unless Galois was wrong) cannot in general be done. In this case, though, we can compute an asymptotic approximation for the solution of  $\alpha = 1 + \frac{1}{ek} + O\left(\frac{k}{k^2}\right)$ .
- Since  $k$  may depend on  $n$ , the Local Limit Theorem cannot be used as in Theorem 5. Instead, we use the saddle-point method of complex variables [deBruijn]. We show the existence of a neighborhood around  $z = 1$  where  $\ln \bar{g}(z)$  converges, so that  $\exp(\ln \bar{g}(z))$  can be expanded in a convergent power series, and derive

$$\langle z^n \rangle \bar{g}(z)^n = \frac{1}{2\pi} \int_{-\delta}^{\delta} \exp\left((\mu-1)itn - \frac{\sigma^2 t^2}{2!} - \frac{i\kappa_3 t^3}{3!} + \dots\right) dt + O(\beta^n),$$

where  $\delta > 0$  is a small constant,  $0 < \beta < 1$  is a constant, and  $(\mu, \sigma^2, \kappa_3, \dots)$  are the semi-invariants of  $\bar{g}(z)$ . Since  $\mu = \bar{g}'(1) = 1$ , the first term  $(\mu-1)itn$  is zero. The proof is completed by use of Laplace's method for integrals around the saddle point. ■

We note that similar asymptotic analysis has been used by Philippe Flajolet to analyze the expected behavior of extendible hashing and trie searching [Flajolet].

**Corollary 10.** *If  $k = O(1)$ , so that we insist on answering queries in a constant number of probes, then  $H_k(m, n) = \Omega(n)$ .*

Theorem 9 demonstrates that when  $n \approx (k+1)!$ , or equivalently  $k \approx \frac{\ln n}{\ln \ln n}$ , the size of  $\mathcal{X}$  is about a constant. Then for a fixed set  $S \in M^{(n)}$ , and randomly chosen  $P \in P^{(n)}$ , the probability that no more than  $\frac{\ln n}{\ln \ln n}$  elements of  $S$  are found in any part in  $P$  is large. This fact is not altogether surprising, since it is closely related to the following classical problem in random allocations: when throwing  $n$  balls at random into  $n$  boxes, what is the expected value of the maximum number of balls in any box? [Kolchin et al.][Diaconis and Freedman]. It turns out that the expected value is about  $\frac{\ln n}{\ln \ln n}$ . In terms of hashing with separate chains, this statistic can be interpreted as the expected length of the longest probe sequence, which has been closely analyzed in [Gonnet].

## §6 The Effect of Table Expansion

We have thus far analyzed two kinds of tradeoffs. In Theorem 6, a tradeoff was effected by synthesizing binary search and perfect hashing. Using this "paged hashing" scheme, we can answer membership queries in  $t$  probes with a program complexity of  $\frac{n^t}{2^{t+1}}(1 + o(1))$

bits. Another tradeoff scheme was analyzed in Theorem 9, where we considered hashing with separate chains of length at most  $t$ , and a consequent maximum search time of  $t$  probes. This "chained hashing" method has a program complexity of  $\frac{n \log_2 e}{e^{t+1}}(1 + o(1))$  bits, which is smaller than the complexity of paged hashing, though  $O(n \log n)$  bits are needed to store pointers for the chains.

What differences between these schemes could cause their differing program complexities? The paged hashing method has an address space of  $n/k$ , with exactly  $k$  keys stored at each address (page). The chained hashing method has an address space of  $n$ , with at most  $k$  keys stored at any address.

Suppose we modified paged hashing so that its address space was expanded, say, from  $n/k$  to  $n$ , and maintained the constraint that exactly  $k$  keys are stored at some  $n/k$  of the  $n$  addresses. How would this modification alter the lower bound on program complexity? Mehlhorn has shown that perfect hashing with a load average of  $\beta$  has a program complexity of  $\Theta(\beta n + \log \log m)$ . However, when the paged hashing method is used with similar table expansion, and  $k$  grows asymptotically large, no such similar reduction occurs in the program complexity. In particular, we can show the following theorem:

**Theorem 11.** Let  $HF_k(m, n, t)$  be the program complexity of the paged hashing method. Then for increasing  $k$ ,  $HF_k(m, n, t) \geq \frac{n \log_2 k}{2k}(1 + o(1))$ .

Theorem 11 demonstrates that the program complexity of the paging method is not significantly reduced by an expansion of the address space, since the lower bound is asymptotically equivalent to the lower bound found in Theorem 5, i.e.,  $HF_k(m, n, t) = \Omega(HF_k(m, n, n/k))$  for all  $t \geq n/k$ .

## §7 Probabilistic Hashing and Hash Circuits

We now give an application of the above analysis to showing a lower bound on the worst-case behavior of universal classes of hash functions (see [Carter and Wegman]).

A class  $\mathcal{X}$  of hash functions is called *universal* if for any  $x, y \in M$ , no more than  $|\mathcal{X}|/n$  of the functions  $h \in \mathcal{X}$  satisfy  $h(x) = h(y)$ . Carter and Wegman essentially showed that by choosing an  $h \in \mathcal{X}$  randomly, we can answer "Is  $x \in S$ ?" in  $O(1)$  probes on the average. More specifically, they showed that for any  $x \in M$  and randomly chosen  $h \in \mathcal{X}$ , the expected number of  $y \in S$  colliding with  $x$  under  $h$  is 1. In various applications it is advantageous to minimize the size of the class of hash functions: in [Mehlhorn] it is shown that the smallest universal class of hash functions has size  $\Theta(n \log_n m)$ . We can then show the following theorem:

**Theorem 12.** Let  $\mathcal{X}$  be a class of universal hash functions of minimal size. Then there exists a set  $S \in M^{(n)}$  depending on  $\mathcal{X}$  such that the following is true: for any  $h \in \mathcal{X}$  there exists a set  $S' \subseteq S$  where  $x, y \in S'$  implies  $h(x) = h(y)$ , and  $|S'| = \Omega(\log n / \log \log n)$ .

This theorem implies that if we use a "minimal" class of universal hash functions, where we choose any function from the class and construct an accompanying table structure, there will always be queries that take  $\Omega\left(\frac{\log n}{\log \log n}\right)$  probes to answer, far worse than the  $O(1)$  expected time derived from choosing randomly. Probabilistic algorithms are designed to give

a good expected time behavior *independent of the input* by randomization in the algorithm. Theorem 12 shows how in some sense the randomization can be "beat," since the proof gives a lower bound on choosing the best function as opposed to a random one, and choosing randomly cannot do any better than choosing the best function. It also suggests that a "minimal" program cannot do much better than binary search, which was shown by [Yao] for a very large key space.

The lower bound of Theorem 12 is on a very worst case measure of the performance of universal classes of hash functions, but does show that their expected behavior is not completely input independent. These functions still can perform well in an amortized sense (say over a sequence of queries over all  $x$  in  $S$ ), but this is just a hidden form of averaging over an input sequence.

A similar bound can be proven about certain classes of hash circuits. Define a *computation-bounded hash circuit* as a directed acyclic graph where the vertices of the graph represent Boolean operations with fan-in bounded by a constant, and fan-out of 1. The computation is additionally bounded by requiring that the circuit can use an input bit (i.e., a bit from the key  $x$ ) no more than a constant number of times. We envision the latter condition by providing a constant number of copies of each input bit to the circuit, which may be "used" or "ignored" by the circuit. The outputs of some fixed  $\log_2 n$  of the vertices are chosen as the bits of the hash address.

**Theorem 13.** *Let  $\mathcal{X}$  be a class of computation-bounded hash circuits, and assume that  $\log_2 m \log_2 \log_2 m = o(n)$ . Then there exists a set  $S \in M^{(n)}$  depending on  $\mathcal{X}$  such that the following is true: for any  $h \in \mathcal{X}$  there exists a set  $S' \subseteq S$  where  $x, y \in S'$  implies  $h(x) = h(y)$ , and  $|S'| = \Omega(\log n / \log \log n)$ .*

## §8 Open Problems

We conjecture that an easy-to-compute hash function with small program complexity can be constructed for any fixed set  $S \in M^{(n)}$  which has no more than  $O(\log n)$  collisions to any address.

Another topic that merits further investigation is *adaptive tradeoffs*. The tradeoff schemes we have considered have mostly been nonadaptive. For example, in hashing with separate chains, the information stored in the search program  $P$  is used to choose the right chain, but subsequent probes do not adapt to information gained by earlier, failed probes (except that they failed). The lower bound in [Yao] analyzes some *adaptive search models*. These models do not allow the search program to have knowledge in advance about the table it searches, and so the analysis is incompatible with the approach taken here. What can be done to unify these two approaches? One promising line of attack might be to extend the approach of Gonnet and Larson: consider an algorithm used for insertion, and show some entropic bound on its behavior which indicates how much information must be passed to the search algorithm.

The analysis of adaptive search leads naturally to more dynamic considerations. What can be said about the program complexity of search strategies that modify themselves as keys are inserted and deleted from the table? This question is probably very difficult.

Hash circuits is an additional area where open problems remain. How strong is the lower bound of Theorem 5.1? If our above conjecture about easy to compute number-theoretic

functions is true, the lower bound should be quite strong. However, VLSI models suggest chip area and time, rather than gate complexity, as measures of the performance of circuits. Can an  $AT^2$  lower bound be shown for, say, a circuit which is a perfect hash function? What kind of information transfer must occur in a hash circuit?

**Acknowledgements.** I would like to thank Dan Greene, Don Knuth, Ernst Mayr, Kurt Mehlhorn, Mark Soldate, Jeff Ullman, and Andy Yao for their many extremely helpful comments and suggestions. This research was supported by National Science Foundation grants MCS-82-03405 and MCS-83-00984, Office of Naval Research contract N00014-81-K-0269, and Air Force contract AFOSR-80-0212.

### §9 References

[Berman et al.]

Berman, Francine, Mary Ellen Bock, Eric Dittert, Michael J. O'Donnell, Darrell Plank, *Collections of Functions for Perfect Hashing*. Preprint, Purdue University, July 1982.

[deBruijn]

deBruijn, Nicolaas G. *Asymptotic Methods in Analysis*. Amsterdam: North-Holland, 1970.

[Carter and Wegman]

Carter, J. Lawrence, and Mark N. Wegman, *Universal Classes of Hash Functions*. *J. Comput. System Sci.* 18, No. 2 (April 1979), pp. 143-154.

[Chaitin]

Chaitin, Gregory J. *A Theory of Program Size Formally Identical to Information Theory*. *J. ACM* 22:3 (July 1975) pp. 329-340.

[Diaconis and Freedman]

Diaconis, Persi, and David Freedman, *The distribution of the mode of an empirical histogram*, Technical Report 105, Department of Statistics, Stanford University, January 1978.

[Feller]

Feller, William. *An Introduction to Probability Theory and Its Applications*, vol. 2, second edition. New York: Wiley, 1971.

[Flajolet]

Flajolet, Phillipe. *On the Performance Evaluation of Extendible Hashing and Trie Searching*. IBM Research Report RJ3258, October 1981.

[Fredman et al.]

Fredman, Michael L., János Komlós, and Endre Szemerédi, *Storing a Sparse Table with  $O(1)$  Worst Case Access Time*. 23rd IEEE Symposium on Foundations of Computer Science, November 1982, pp. 165-169.

[Gonnet]

Gonnet, Gaston H. *Expected Length of the Longest Probe Sequence in Hash Code Searching*. *J. ACM* 28:2 (April 1981), pp. 289-304.

[Gonnet and Larson]

Gonnet, Gaston H., and Per-Åke Larson, *External hashing with limited internal storage*. ACM Symposium on Principles of Database Systems, 1982, pp. 256-261.

[Greene and Knuth]

Greene, Daniel H., and Donald E. Knuth, *Mathematics for the Analysis of Algorithms*. Boston: Birkhauser, 1982.

[Jaeschke]

Jaeschke, G. *Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions*. *Comm. ACM* 24:12 (Dec. 1981), pp. 829-833.

[Kolchin et al.]

Kolchin, Valentin F., Boris A. Sevast'yanov, Vladimir P. Chistyakov, *Random Allocations*. New York: Wiley, 1978.

[Knuth III]

Knuth, Donald E. *The Art of Computer Programming*, vol. 3. Reading: Addison-Wesley, 1973.

[Mehlhorn]

Mehlhorn, Kurt. *On the Program Size of Perfect and Universal Hashfunctions*. Preprint, University of Saarlandes, 1982.

[Petrov]

Petrov, Valentin V. *Sums of Independent Random Variables*. New York: Springer-Verlag, 1975.

[Sprugnoli]

Sprugnoli, Renzo. *Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets*. *Comm. ACM* 20:11 (Nov. 1977), pp. 841-850.

[Tarjan and Yao]

Tarjan, Robert E., and Andrew C. Yao. *Storing a Sparse Table*. *Comm. ACM* 22:11 (Nov. 1979), pp. 606-611.

[Wegman and Carter]

Wegman, Mark N., and J. Lawrence Carter, *New Hash Functions and Their Use in Authentication and Set Equality*. *J. Comput. System Sci.* 22, 1981, pp. 265-279.

[Yao]

Yao, Andrew C. *Should Tables Be Sorted?* *J. ACM* 28:3 (July 1981), pp. 615-628.

[Zion]

Zion, Gene. *Harry the Dirty Dog*. Illustrated by Margaret Bloy Graham. New York: Harper and Row, 1956.

DATE  
FILMED  
8