Optimal Arrangement of Keys in a Hash Table

RONALD L. RIVEST

Massachusetts Institute of Technology, Cambridge, Massachusetts

ABSTRACT When open addressing is used to resolve collisions in a hash table, a given set of keys may be arranged in many ways, typically this depends on the order in which the keys are inserted It is shown that arrangements minimizing either the average or worst-case number of probes required to retrieve any key in the table can be found using an algorithm for the assignment problem. The worst-case retrieval time can be reduced to $O(\log_2(M))$ with probability $1 - \epsilon(M)$ when storing M keys in a table of size M, where $\epsilon(M) \to 0$ as $M \to \infty$ We also examine insertion algorithms to see how to apply these ideas for a dynamically changing set of keys

KEY WORDS AND PHRASES hashing, collision resolution, searching, assignment problem, optimal algorithms, database organization

CR CATEGORIES 3 74, 5 41

"Spread the table and contention will cease " Old English proverb [11, #272 6]

1. *Introduction*

We consider schemes to optimize the placement of keys in a hash table when open addressing is used to resolve collisions. More precisely, we begin with the observation that a given set of keys may be inserted into a hash table in many different orders, yielding arrangements of the keys in the table of varying efficiency. Typically, the user has no control over the order in which the keys are inserted; he must accept them in the order in which they arrive. However, the previous observation that there exist many different arrangements of the given set of keys raises the following questions:

(1) How can one determine that arrangement which minimizes either the average or worst-case number of probes to retrieve a key in the table? In Section 2 we show that this problem is an instance of the well-known "assignment problem," for which efficient algorithms exist.

(2) What is the expected value of the worst-case number of probes required to retrieve a key from a full table that has been optimally arranged using the assignment algorithm? In Section 3 it is proved that this value is $O(\log_2(M))$ for a table of size M containing M keys. The proof is modeled on a result by Erdös and Renyi [2] concerning the permanent of a random matrix. This result demonstrates that we can use hashing to achieve "good" (i.e. $O(\log_2(M))$) worst-case performance if we take the time to optimize the arrangement of the keys in the table. Traditionally hashing has

This research was prepared with the support of the National Science Foundation under Research Grant GJ-43534X, Contract DCR74-12997, and Research Grant MCS76-14294

Author's address Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139

been viewed as excellent on the average, but horrible in the worst case. We see therefore that this need not be so.

(3) The results mentioned above require that an $M \times M$ assignment problem be solved to optimize the placement of M keys in a table of size M. A natural question to ask is, "Is it possible to solve the assignment problem efficiently 'incrementally,' so that the new keys can be added to the table in such a way that the optimality of the overall arrangement is maintained?" In Section 4 this problem is studied and it is shown that for table densities less than approximately 0.415, it is possible to insert a key and maintain overall optimality by solving an assignment problem no larger than 10×10, whereas for larger densities the entire $M \times M$ assignment problem must apparently be solved.

Overall, we view the contribution of this paper to be the introduction of the assignment algorithm for the placement of keys in a hash table, and the demonstration that efficient worst-case retrieval can be achieved thereby, even in a full table.

We proceed now to define our terminology and to introduce the "standard" algorithm for inserting a key into a hash table. Let $\mathcal{K} = \{K_1, K_2, \ldots, K_N\}$ be a set of N keys, and let an array T_i for $1 \leq i \leq M$ be a set of M memory locations (the hash table) which will be used to store $\mathcal{K}$. Each table position may hold either a single key or the special symbol *empty*. We assume $N \leq M$. When open addressing is used to resolve collisions a "hashing function" $h: U \times \{1, 2, \ldots, M\} \rightarrow \{1, 2, \ldots, M\}$ is used, mapping the set U of all possible keys (that is, $\mathcal{K}$ may be any N-subset of U) and probe numbers into the set of memory locations. We assume for any key $K \in U$ that the sequence $h(K, 1), h(K, 2), \ldots, h(K, M)$ is a permutation of $\{1, 2, \ldots, M\}$ To store the key K in the table using the standard insertion algorithm the locations $T_{h(K,1)}, T_{h(K,2)}, \ldots$ are successively examined until an empty location is found or until K is found already present in the table. The following program makes this precise.

THE "STANDARD" INSERTION ALGORITHM

Input: A key K, a hash table T, a hash function h
Output None T is modified to contain K, unless K is already present
Procedure

```
j := 0,
repeat j = j + 1,
       i = h(K, j),
       if T_i = empty then T_i = K
until  T_i = K,
```

Note that T must contain at least one empty location if K is not already in the table, if the loop is to terminate properly The value of j at termination, which is the number of probes required to insert K, is taken to be the cost of inserting K.

A similar procedure searches for the presence of a key K in T (replace the assignment statement "$T_i := K$" by "return (K not present)") If the **repeat** loop terminates normally then T_i contains the previously stored key K. The value of j at termination is taken to be the cost of searching for K.

Knuth [6] studies hashing algorithms in detail, giving alternative methods for handling "collisions" (the case when $h(K_i, 1) = h(K_j, 1)$ for $K_i \neq K_j$) and several open-addressing hash functions h. The reader who is unfamiliar with hashing algorithms should find it profitable to consult his text.

2 *Optimal Arrangements*

In this section we give precise definitions of when an arrangement minimizes the average or worst-case retrieval time, and then show that there always exists some ordering such that if the keys had been inserted by the standard algorithm in that order, the optimal arrangement results Then it is shown that the assignment algorithm

can be used to arrange the keys so as to minimize either the average or worst-case retrieval time.

The arrangement of the keys $\mathcal{K}$ in the hash table depends on the order in which they were inserted, if the standard insertion algorithm is used. For example, let U be the set of natural numbers and let $h(K, j)$ be the jth decimal digit of K. Inserting the set $\mathcal{K} = \{1423, 1234, 3412, 2341\}$ into an empty table in that order results in the arrangement α:

Location:	1	2	3	4
Contents:	1423	1234	3412	2341

whereas inserting them in the order 1234, 2341, 1423, 3412 results in α':

Location:	1	2	3	4
Contents	1234	2341	3412	1423

Let $\alpha: \mathcal{K} \to \{1, 2, \ldots, M\}$ be called an *arrangement*; $\alpha(K_i) = j$ means that $T_j = K_i$. Of course α must be one-to-one. Let $A(\mathcal{K}, M)$ denote the set of all arrangements of $\mathcal{K}$ in $T_1, \ldots, T_M$.

Let $p(K, \alpha)$ denote the number of probes required to retrieve a key K under arrangement α; the average $\text{avg}(\alpha) = (1/N) \sum_{K \in \mathcal{K}} p(K, \alpha)$ and worst-case $\text{wc}(\alpha) = \max\{p(K, \alpha) \mid K \in \mathcal{K}\}$ number of probes to retrieve any key in T are then definable. We have $\text{avg}(\alpha) = 7/4$, $\text{wc}(\alpha) = 3$, $\text{avg}(\alpha') = 5/4$, and $\text{wc}(\alpha') = 2$ in the above examples.

Define an arrangement $\alpha \in A(\mathcal{K}, M)$ to be *valid* if all the positions $h(K, 1), h(K, 2), \ldots, h(K, p(K, \alpha) - 1)$ are nonempty for every key K in $\mathcal{K}$. An arrangement is valid iff every key K in $\mathcal{K}$ is retrievable using the search algorithm of Section 1 Similarly define an arrangement to be *feasible* if it is the result of inserting the keys in $\mathcal{K}$ into an empty table sequentially in some order; necessarily every feasible arrangement is valid.

Valid arrangements which are not feasible are possible; consider the following arrangement using the hash function h from our previous example:

Location:	1	2	3	4
Contents:	empty	empty	4321	3412

The number of feasible arrangements depends on $\mathcal{K}$ and h. It is no larger than $N!$ (the number of ways to enter the keys), but may be as low as 1 if no collisions occur. Similarly the number of valid arrangements can vary between 1 and $N!$. For example, only one valid arrangement exists if no collisions occur and $h(K_i, 1) \neq h(K_j, 2)$ for all K_i, K_j in $\mathcal{K}$. The upper bound of $N!$ on the number of valid arrangements is obtained by induction on N, using the fact that $p(K, \alpha) \leq N$ for any valid arrangement and all keys $K \in \mathcal{K}$. We may store K_N in any of N positions $h(K_N, i)$ for $1 \leq i \leq N$; if we then delete K_N from $\mathcal{K}$ and $h(K_N, i)$ from the probe sequence $h(K_j, 1), \ldots, h(K_j, M)$ for every $j < N$ we see that every valid arrangement of $\mathcal{K}$ induces a valid arrangement of $\mathcal{K}-\{K_N\}$ in locations $\{j \mid 1 \leq j \leq M \text{ and } j \neq h(K_N, i)\}$ using the modified probe sequences.

We define an arrangement $\alpha(\mathcal{K}, M)$ to be *optimal* if either $\text{avg}(\alpha)$ or $\text{wc}(\alpha)$ is minimal over all arrangements in $A(\mathcal{K}, M)$; the terms average-optimal and worst-case-optimal will distinguish these cases.

PROPOSITION 1. *A feasible optimal arrangement always exists.*

PROOF If a minimal arrangement α is not feasible, then there exists a set $\{K_{i_0}, K_{i_1}, \ldots, K_{i_{r-1}}\}$ of keys, none of which can be entered first since they form a "blocking cycle": There is a set of integers t_j for $0 \leq j \leq r - 1$ such that $h(K_{i_j}, p(K_{i_j}, \alpha)) = h(K_{i_{(j+1) \bmod r}}, t_{(j+1) \bmod r})$ and $t_j < p(K_{i_j}, \alpha)$ for $0 \leq j \leq r - 1$. But clearly $p(K_{i_j}, \alpha)$ can be reduced by setting $\alpha(K_{i_j})$ to $h(K_{i_j}, t_j)$ for $0 \leq j \leq r - 1$. Since $\text{avg}(\alpha)$ strictly decreases, a feasible optimal arrangement can always be found after a finite number of blocking cycles have been removed in this fashion. □

Proposition 1 suggests an algorithm for finding optimal arrangements: enumerating all feasible arrangements; however, better methods exist.

PROPOSITION 2 *Optimal arrangements can be found by using an algorithm for the assignment problem*

PROOF. The assignment problem [7] can be stated as follows.

Let N and M be given, with $N \leq M$, and let $\{a_{ij} | 1 \leq i \leq N, 1 \leq j \leq M\}$ be a matrix of nonnegative real numbers. The classic example specifies for each of M men and N jobs, the "inefficiency" a_{ij} of man j in job i. The objective is to find an assignment $i \to \alpha(i)$ of jobs to men such that the sum $\sum_{1 \leq i \leq N} a_{i,\alpha(i)}$ is minimized, subject to the constraint that no man is assigned to more than one job.

We can apply this directly to the problem of finding average-optimal arrangements by letting a_{ij} be the integer such that $h(K_i, a_{ij}) = j$, denoting the cost of assigning K_i to T_j. The average number of probes required to retrieve a key in the optimized table is then just the total "inefficiency" divided by N We observe that if the various keys have associated retrieval probabilities, then the arrangement that minimizes the expected retrieval cost can be found in the same manner; we need only multiply each a_{ij} by the probability that K_i will be retrieved.

Similarly, we can minimize the worst-case cost by choosing a_{ij} to be N^l, where l is the integer such that $h(K_i, l) = j$. Since the key with highest cost determines the order of the total cost, minimizing the total cost here minimizes the worst-case cost. □

Having observed that our problem can be formulated as an instance of the assignment problem, it is of interest to know how quickly a solution can be determined. The general $N \times M$ assignment problem can be solved in time $O(NM^2)$ [8]; the space required is $O(N + M)$ if the matrix entries a_{ij} can be computed in constant time from K_i, h, and j. When all the matrix entries are small integers (as when we are finding the average-optimal arrangement), it may be possible to improve this time bound somewhat, but the author was unable to find a more efficient procedure.

Worst-case optimal arrangements can be determined in time $O(BM(M, N) \cdot \log_2(N))$, where $BM(M, N)$ is the time required to solve an $M \times N$ bipartite matching problem. The procedure, pointed out to the author by Vuillemin, is to use binary search on the worst-case cost: It is possible to test if the optimal worst-case cost is less than or equal to a given value w by solving the corresponding maximal matching problem. The graph used has N vertices x_i, M vertices y_j, and an edge (x_i, y_j) iff $a_{ij} \leq w$. Intuitively, there is an edge from x_i to y_j if and only if table position T_j is one of the first w positions in the probe sequence for K_i. There will be a matching of size N in this graph if and only if there is an arrangement of the keys in the table such that every key can be retrieved with no more than w probes. Since $BM(M, M) = O(M^{2.5})$, we obtain an $O(M^{2.5} \log(M))$ algorithm for the case $N = M$.

3 *Efficiency of the Worst-Case Optimal Arrangements*

In this section we prove that even if the hash table is full ($N = M$), we can expect the worst-case optimal arrangement to have a worst-case cost of $O(\log(M))$ with a probability approaching one very rapidly as $M \to \infty$. Although a worst-case cost of $O(\log(M))$ can obviously not be guaranteed (since there is a finite chance that all keys have the same probe sequence, for example), the odds are overwhelming that with a random hash function and a random set of keys, there is some arrangement of those keys yielding a worst-case cost of $O(\log(M))$. This compares favorably with standard techniques such as binary search trees which also require $O(\log_2(N))$ time to retrieve a key, especially in situations where the set of keys is static (since updating an optimized hash table can be expensive).

The proof is modeled very closely after a similar result of Erdos and Renyi [2], who show that a random $n \times n$ matrix of 0's and 1's containing $N(n)$ 1's has a nonzero permanent with probability approaching 1 as $n \to \infty$ if $\lim_{n \to \infty}(N(n) - \log(n))/n = \infty$. The permanent of an $n \times n$ matrix $\{a_{ij}\}$ is defined to be $\sum a_{1i_1} a_{2i_2} \cdots a_{ni_n}$, where the summation is over all permutations $(i_1, \ldots, i_n)$ of $\{1, \ldots, n\}$. The permanent of a 0-1

matrix $\{a_{ij}\}$ is the number of matchings of size n in a bipartite graph whose adjacency matrix is $\{a_{ij}\}$ Ryser [10] discusses the permanent in some detail

Let $\mathcal{M}(M, N, w)$ denote the set of all 0-1 matrices with M columns, N rows, and exactly w 1's per row. Obviously $|\mathcal{M}(M, N, w)| = \binom{M}{w}^N$. We say a matrix $\{m_{ij}\} \in \mathcal{M}(M, N, w)$ contains N independent 1's iff there exists a function $\alpha:\{1, \ldots, N\} \to \{1, \ldots, M\}$ such that $\alpha(i) \neq \alpha(j)$ for $i \neq j$ and $m_{i,\alpha(i)} = 1$ for $1 \leq i \leq N$. Let $P(M, N, w)$ denote the probability that a matrix in $\mathcal{M}(M, N, w)$ contains N independent ones.

The interpretation to matrices of $\mathcal{M}(M, N, w)$ is as follows. Each such matrix has N rows (corresponding to a set of N keys) and M columns (one for each position in the hash table) Position i, j will be a 1 *iff* key i can be stored in position j with a retrieval cost of w or less. Therefore each row has exactly w 1's Such a matrix is the adjacency matrix of one of the bipartite graphs described in the last paragraph of Section 2 A matrix in $\mathcal{M}(M, N, w)$ will have N independent ones iff its corresponding bipartite graph has a matching of size N. This will happen iff there exists an arrangement of the keys so that every one can be retrieved with w probes or less.

We identify $P(M, N, w)$ with the probability that a random set of N keys can be arranged in a hash table of size M so that the worst-case retrieval cost is at most w. This will be accurate if every set of w locations is equally likely to be the set of w locations first probed for a random key k This will happen, for example, if every permutation of $\{1, \ldots, M\}$ is equally likely to be a probe sequence. Each matrix in $\mathcal{M}(M, N, w)$ then corresponds in a natural fashion to the characteristic matrix describing, for a random set of N keys, which locations are usable if the worst-case cost is constrained to be at most w The existence of N independent 1's corresponds to the existence of an arrangement with worst-case cost of at most w; and by Proposition 1 the existence of a feasible, valid arrangement with worst-case cost at most w is thereby implied.

We have $P(M, N, w) \geq P(M, M, w)$ for $1 \leq N \leq M$ since the first N rows of a matrix in $\mathcal{M}(M, M, w)$ which contains M independent 1's must contain N independent 1's. We therefore proceed to show the following.

PROPOSITION 3. $\lim_{M\to\infty} P(M, M, 4 \log(M)) = 1$.

PROOF. This result says that we can expect to find an arrangement of M keys in a table of size M such that no key requires more than 4 $\log(M)$ probes to be retrieved. By the theorems of Frobenius [3] and König [7], $1 - P(M, M, w)$ is equal to the probability that a matrix in $\mathcal{M}(M, M, w)$ has k rows (or columns) and $M - k - 1$ columns (or rows) that contain all the 1's, for some k, $0 \leq k \leq M - 1$. (The result of Frobenius and König says that in an $M \times M$ matrix of 0's and 1's the minimal number of lines (i.e. rows or columns) which contain all the 1's is equal to the size of the maximum set of 1's which can be found which are pairwise independent (no two in the same line).) Thus $1 - P(M, M, w)$ is the probability that there are $M - 1$ or fewer lines which contain all the 1's.

Let $Q_k(M, N, w)$ denote the probability that a matrix in $\mathcal{M}(M, N, w)$ has k rows (or columns) and $N - k - 1$ columns (or rows) containing all the 1's, and k is the least such number for $0 \leq k \leq M/2$. Then

$$1 - P(M, N, w) = \sum_{k=0}^{\lfloor M/2 \rfloor} Q_k(M, N, w).$$

We show that for all k, $0 \leq k \leq \lfloor m/2 \rfloor$, if $w \geq 4 \log_2(M)$ then $Q_k(M, M, w) \to 0$. To do this we divide Q_k into two parts,

$$Q_k(M, M, w) = f_k(M, M, w) + g_k(M, M, w),$$

where f_k is the probability that k rows and $M - k - 1$ columns cover all the 1's and g_k is the probability that k columns and $M - k - 1$ rows cover all the 1's (k is each case being minimal).

Case 1. k rows and $M - k - 1$ columns contain all the 1's, for some $k \leq M/2$.

Those matrices in $\mathcal{M}(M, M, w)$ having a minimal number k of rows and $M - k - 1$ columns containing all the 1's can be displayed as in Figure 1, after an appropriate permutation of the rows and columns. Each row of submatrix B must contain two 1's under our assumption that k is minimal (if not, we could include the column, and exclude the row, of the 1 in matrix B which is in a row of B containing no other 1's). The fraction $f_k(M, M, w)$ of matrices of this type is less than

$$\left(\binom{M}{k}\binom{M}{k+1}\binom{M-k-1}{w}^{M-k}\left(\binom{M}{w} - \binom{M-k-1}{w} - (k+1)\binom{M-k-1}{w-1}\right)^{k}\right) \Big/ \binom{M}{w}^{M},$$

whose logarithm is bounded above by

$$[(2k+1) - w(M-k)]\log(M) + w(M-k)\log(M-k-1) - k\log(k) - (k+1)\log(k+1) \leq (2k+1)\log(M) - w(k+1)/2.$$

Thus if $w \geq 4\log(M)$, $Q_k(M, M, w) \to 0$ as $M \to \infty$.

Case 2 k columns and $M - k - 1$ rows contain all the 1's, for some $k \leq M/2$ (Figure 2).

The fraction $g_k(M, M, w)$ of matrices of this type is less than

$$\binom{M}{k}\binom{M}{k+1}\binom{M}{w}^{-(k+1)}\binom{k}{w}^{k+1},$$

whose logarithm is bounded above by

$$(2k+1)\log(M) - w(k+1)\log(w),$$

so that $g_k(M, M, w) \to 0$ with M if $w = 2\log(M)$. Since $Q_k(M, M, w) = f_k(M, M, w) + g_k(M, M, w)$, we are finished with the proof. □

This result says that in a full table arranged so as to minimize the worst-case retrieval time, the worst-case retrieval time should be $O(\log(M))$. This follows from Proposition 3 since the existence of a set of M independent 1's in a matrix in $P(M, M, w)$ corresponds to an arrangement of M keys in a table of size M with worst-case retrieval time no more than w. This result is the best possible (up to a constant multiplicative factor) due to a result of Gonnet [4]: The worst-case retrieval time must be at least $\ln(M) + O(1)$.

A study of the related question of the expected value of the average number of probes required to retrieve a key in a full table which is average-optimal is given in [5]. (Less than two probes per key are required.)

4. *Insertion Algorithms Which Maintain Optimality*

We now turn our attention to the problem of maintaining the optimality of an arrangement as new keys are inserted into a table. The main result of this section is that if the table is not too densely filled, then a new key can be inserted into the table

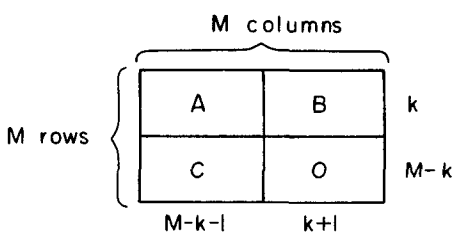

FIG 1

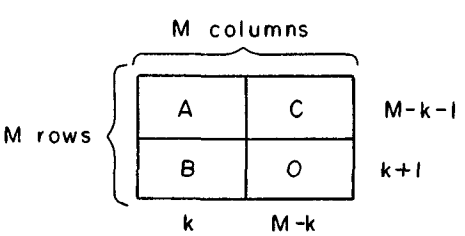

FIG 2

and the new optimal arrangement computed by solving a small (e.g 10×10) assignment problem. This result is obtained by a rather complicated analysis using generating functions.

We first examine an insertion algorithm due to Brent [1] and demonstrate that it does not maintain optimality. Of course, Brent only intended his algorithm to be a good heuristic, a means of inserting each new key in such a fashion that the increase in average retrieval cost is kept reasonably low

Brent's algorithm works as follows. Let K denote the new key being inserted, and suppose positions $h(K, 1), \ldots, h(K, s)$ are already occupied with keys $K_1, K_2, \ldots, K_s$, and that $T_{h(K,s+1)}$ is empty. Let r_i denote the number of probes required to retrieve K_i, so that $h(K_i, r_i) = h(K, i)$. Furthermore, let s_i denote $\min\{j \mid T_{h(K_i,j)} = \textit{empty}\}$, the number of probes required to retrieve K_i if we move it to position $h(K_i, s_i)$. Then $(i + (s_i - r_i))/(N + 1)$ is the increase in the average retrieval cost caused by moving K_i to position $h(K_i, s_i)$ and storing K in position $h(K, i)$. Brent chooses between storing K in position $h(K, s + 1)$ and moving that K which minimizes $i + (s_i - r_i)$ by comparing $(s + 1)$ to $\min_i\{i + s_i - r_i\}$.

In fact, the following example demonstrates that no algorithm which only moves keys forward in their probe sequence (that is, moves K from $h(K, i)$ to $h(K, i')$ for $i' > i$) can always arrive at the optimal arrangement. Consider the following arrangement (using the hash function of our previous examples), which is both average and worst-case optimal:

Location	1	2	3	4	5	6	7
Contents	1273456	1234567	3456712	4567123	5671234	6712345	empty

If the key 2345671 is now inserted, the only way to maintain optimality is to move 1273456 to location 7, move 1234567 (backward) to position 1, and then store 2345671 in position 2

Since Brent's algorithm is the only published algorithm which moves previously inserted keys when inserting a new key, we see that no existing insertion algorithm can maintain optimality for arbitrary hash functions. It is interesting to note, however, that for certain open-addressing collision-resolution schemes the standard insertion algorithm maintains average-optimality. We say that a hash function h exhibits *primary clustering* if $h(K_i, j) = h(K_{i'}, j')$ implies that $h(K_i, j + l) = h(K_{i'}, j' + l)$ for $0 \leq l \leq M - \min(j, j')$ for any $K_i, K_{i'}$. Linear probing ($h(K, i) \equiv h(K, 1) + (i - 1)$, mod M) is perhaps the best-known example of a collision-resolution scheme exhibiting primary clustering, and all primary clustering schemes are in fact isomorphic to linear probing in a natural manner

PROPOSITION 4. *If h exhibits primary clustering, then the usual insertion algorithm maintains average-optimality*

PROOF. This theorem is due to Peterson [9]; the proof is also given in Knuth [6, p. 531]. Knuth also remarks that if the keys have associated retrieval probabilities, then the average-optimal arrangement can be achieved by using the standard insertion routine to insert the keys one by one into the table, in order of decreasing request probabilities. □

In spite of the fact that for linear probing the standard insertion algorithm maintains average-optimality, other hashing schemes are to be preferred, since the expected retrieval cost in the average-optimal scheme for a primary-clustering hashing function generally exceeds the expected cost for other schemes, even if average-optimality is not maintained.

We now turn our attention to the task of finding an insertion algorithm that will maintain the optimality of an arrangement. In essence, we need an algorithm to solve the assignment problem "incrementally."

One approach is to observe that if N/M is small enough (how small this is we shall determine), then the number of keys already in the table which we need to consider moving might be reasonably small. Brent considers moving only those keys on the

probe sequence of the new key K; if we also consider moving all of the keys on their probe sequences, and so on, we can determine the maximum set $\mathcal{S}$ of keys that might need to be moved. Similarly we let $\mathcal{T}$ denote the set of locations that $\mathcal{S}$ might occupy in the optimized table; it suffices then to solve the assignment problem for placing $\mathcal{S}$ into $\mathcal{T}$, rather than $\mathcal{K} \cup \{K\}$ into T.

Define, for a given arrangement α, the functions:

$$\pi(K) = \min\{j \mid h(K, j) = \text{empty}\},$$
$$\sigma(K) = \{K_i \mid \alpha(K_i) = h(K, j) \text{ for some } j < \pi(K)\},$$
$$\tau(K) = \{i \mid h(K, j) = i \text{ for some } j \leq \pi(K)\}.$$

Then

$$\mathcal{S}(K) = \{K\} \cup \{\mathcal{S}(K_i) \mid K_i \in \sigma(K)\},$$
$$\mathcal{T}(K) = \tau(K) \cup \{\mathcal{T}(K_i) \mid K_i \in \sigma(K)\}$$

define by means of their minimal solutions the sets $\mathcal{S}$ and $\mathcal{T}$ of keys and positions relevant to the insertion of K into an arrangement α

Let $\beta = N/M$ denote the "loading factor" of the existing arrangement α. In order to estimate the expected size $\mathcal{S}(K)$, we assume that the hashing function is uniform in the sense that every permutation of $\{1, \ldots, M\}$ is equally likely to be a probe sequence of some key K. We can then use the approximation $\text{Prob}(\pi(K) = i) = (1 - \beta)\beta^{i-1}$

Let s_i denote the probability that $|\mathcal{S}(K)| = i$, and let

$$S(z) = \sum_{i=1}^{\infty} s_i z^i$$

denote the corresponding generating function. We shall develop an equation for $S(z)$ which depends on the generating function:

$$P(z) = \sum_{i=1}^{\infty} p_i z^i$$

(where p_i is the probability that, for a key K' already stored in T, $\alpha(K') = h(K', i)$). However, determining $P(z)$ for optimized hash tables remains an open problem, so we shall approximate $S(z)$ after we develop the correct defining equation.

Let $C(z) = \sum_{i=1}^{\infty} c_i z^i$ be the generating function with coefficients c_i equal to the probability that the "contribution" of a key K' on the probe sequence of the new key K to $S(K)$ is i keys. Therefore

$$S(z) = \sum_{i=0}^{\infty} (1 - \beta)\beta^i [C(z)]^i \cdot z,$$

since there is a probability of $(1 - \beta)\beta^i$ that $\pi(K) = i + 1$ (that is, there are i keys on the probe sequence for the new key K). The final z is for the key K itself.

Similarly we can define

$$C(z) = \left[\sum_{i=1}^{\infty} p_i (C(z))^{i-1}\right] \cdot \left[\sum_{i=0}^{\infty} (1 - \beta)\beta^i (C(z))^i\right] \cdot z$$

(or equivalently,

$$(1 - \beta C(z)) \cdot (C(z))^2 = (1 - \beta) P(C(z)) z).$$

The first term accumulates the contributions of those keys K'' on the probe sequences of a key K' on the probe sequence for K, such that K'' occurs before K' in the probe sequence for K' The second term adjusts for those keys K'' occurring after K' in the probe sequence for K'. Finally, the third term z is for the key K' itself.

The expected size of $\mathcal{S}(K)$ is $S'(1)$; and

$$S'(z) = \frac{d}{dz}\left(\frac{(1 - \beta)z}{(1 - \beta C(z))}\right) = \frac{(1 - \beta C(z))(1 - \beta) + (1 - \beta) z \beta C'(z)}{(1 - \beta C(z))^2}$$

so that

$$S'(1) = 1 + \frac{\beta C'(1)}{(1 - \beta)}.$$

Now

$$(1 - \beta C(z))2C(z)C'(z) - \beta C'(z)(C(z))^2 = (1 - \beta)[P'(z)C'(z)z + P(C(z))]$$

so we obtain

$$C'(1) = (1 - \beta)/(2 - 3\beta - (1 - \beta)P'(1))$$

and thus

$$S'(1) = 1 + \beta/(2 - 3\beta - (1 - \beta)P'(1)).$$

Unfortunately, $P(z)$ is unknown. We observe, however, that $S'(1)$ can be expected to remain finite as long as $P'(1) \leq (2 - 3\beta)/(1 - \beta)$. Since $P'(1)$ is the expected number of probes required to retrieve a key from an optimized table, it is bounded above by the expected number of probes required to retrieve a key from a table organized with any open-addressing hashing method. For uniform probing (all probes sequences equally likely) we have [6]

$$P'(1) \cong \beta^{-1}\log(1/(1 - \beta))$$

approximately. Substituting this into the final equation for $S'(1)$ yields Figure 3; we see that the size of the relevant assignment problem is reasonably small (say 10 keys or less) as long as $\beta \leq 0.4$ roughly The function $S'(1)$ has a pole $\beta = 0.41466541$; for loading densities less than this we can expect the number of relevant keys to be finite. In practice we should expect to be able to handle even higher loading densities without much trouble, since our formulas for S, C, and P explicitly ignore the probability of overlapping probe sequences Furthermore, replacing $P(z)$ by its correct definition (rather than the one for uniform probing) should yield a definite improvement.

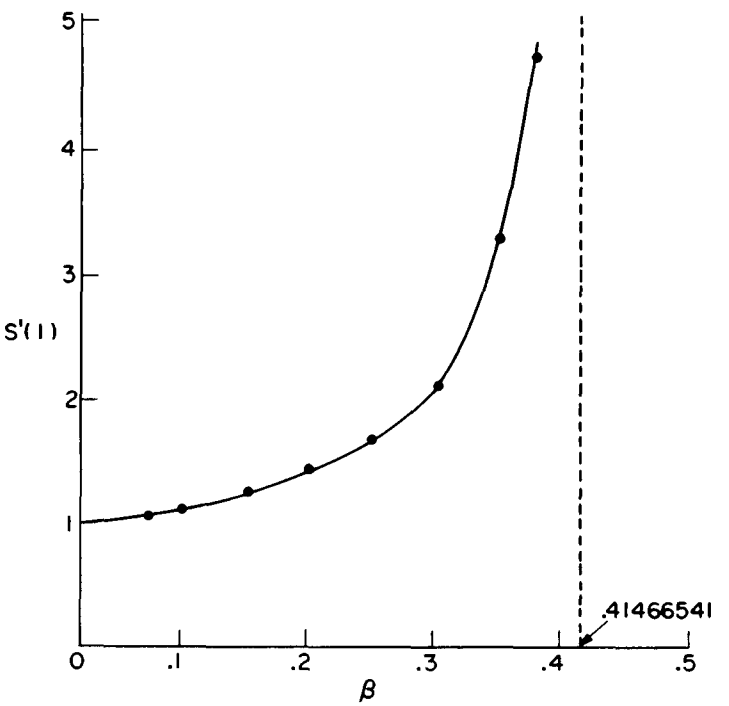

FIG 3

The result of this rather complicated analysis is that if the loading density of the file is less than roughly 0.4 we can hope to insert a new key K into the table by solving a small assignment problem. For higher densities the problem is inherently a global one apparently; we must consider for relocation a considerable number of keys.

5. *Discussion and Conclusions*

In this paper we have shown how to arrange a set of keys in a hash table so as to minimize the expected (or worst-case) number of probes required to retrieve a key. Our analysis demonstrates that the worst-case cost can be reduced to $O(\log_2(M))$ in almost all cases. (In practice it should be possible to achieve $O(\log_2(M))$ in all cases with very little work, since a set of keys which has an optimized cost that is too large can, by choosing another hash function randomly, be expected to yield an $O(\log_2(M))$ cost.)

Our analysis assumes that uniform hashing is used, however; an open problem is to confirm this result for the more common techniques such as double hashing.

We have also examined briefly a technique for inserting a new key into an optimized table so as to maintain optimality of the arrangement. Our result here is that as long as the loading factor is less than 0.41 (approximately), we can usually insert a new key and maintain optimality by solving a small (approximately 10-element) assignment problem. For tables of higher density one must apparently solve an assignment problem which involves most of the keys previously stored. (By saving the primal and dual variables of the previous solution, one can significantly speed up the solution of the new problem, but the extra storage required might better be used to store the keys themselves, thereby reducing the overall density.)

The reader is encouraged to consult the excellent article by Gonnet and Munro [5], which gives explicit listings of algorithms for optimizing the arrangement of keys in a hash table and tight results on the expected number of probes required to retrieve a key from an average-optimal table.

The techniques described here should be most useful when the hash table is relatively static, with the number of retrievals considerably exceeding the number of insertions Large databases are often of exactly this nature, and frequently utilize hashing techniques

ACKNOWLEDGMENT. I would like to thank Professor Donald Knuth for suggesting directions in which to extend a previous draft of this paper.

REFERENCES

1 BRENT, R P Reducing the retrieval time of scatter storage techniques *Comm ACM 16*, 2 (Feb 1973), 105–109
2 ERDOS, P , AND RENYI, A On random matrices *Magyar Tud Akad Mat Kutató Int. Kozl 8* (1964), 455–461 Reprinted in Erdos, P *The Art of Counting*, J Spencer, Ed , M I T Press, Cambridge, Mass (1973), pp 625–631
3 FROBENIUS, G Uber zerlegbare Determinaten *Sitzungsberichte der Berliner Akademie* (1917), 274–277
4 GONNET, G H Interpolation and interpolation hash searching Res Rep 76-02, Comptr Sci Dept , U of Waterloo, Waterloo, Ont , 1976
5 GONNET, G , AND MUNRO, I The analysis of an improved hashing technique Proc Ninth Annual ACM Symp on Theory of Comptng , Boulder, Colo , 1977, pp 113–121
6 KNUTH, D E *The Art of Computer Programming, Vol 3 Sorting and Searching* Addison-Wesley, Reading, Mass , 1973
7 KONIG, D Graphok és matrixok *Matematikai és Fizikai Lapok 38* (1931), 116–119.
8 KUHN, H W The Hungarian method for the assignment problem *Naval Res Log Quart 2* (1955), 83–97
9 PETERSON, W W Addressing for random-access storage *IBM J Res and Develop 1* (1957), 130–146
10 RYSER, H J Combinatorial Mathematics Carus Math Mono #14, Math Assoc Amer , 1963
11 TRIPP, R *International Thesaurus of Quotations* Thomas Y Crowell, New York, 1970

RECEIVED JUNE 1976, REVISED JUNE 1977