Conclusions

In the experimental results given above, the precision of 6 hexadecimal digits is approximately statistically equivalent to that of 22 binary digits. It must be emphasized, however, that the comparison of average accuracies is but one of several methods of comparison. In order to evaluate the relative merits of both systems, it is necessary to consider other aspects, such as the worst case accuracy and the ability to preserve algebraic identities.

Our tests do show the statistical superiority of R-mode arithmetic over C-mode in all cases except that of mixed sign sums, where the C-mode without guard characters has an advantage traceable to a slight bias in the R-mode. The unbiased R^*-mode with guard characters, however, is statistically more accurate than the C-mode for all of our tests.

Test results in the simpler cases have been verified analytically. In the remaining cases where we failed to substantiate the test results analytically, we nevertheless feel that the tests provide useful and well-defined information.

We finally note that machine implementations of hexadecimal or binary C-mode arithmetic systems using 0 or 1 guard digits are common. Implementations of a binary R-mode, or variations thereof, also exist. However, to our knowledge, there are no commercial implementations of the R^*-mode.

Acknowledgment. The authors wish to express their appreciation to W. Kahan, whose detailed comments on an early version of this work led to the pursuit of the analytic estimates presented here, as well as to other refinements in the presentation.

Received June 1971; revised April 1972

References

1. Hamming, R. *Numerical Methods for Scientists and Engineers*, McGraw-Hill, New York, 1962.
2. Knuth, D.E. *The Art of Computer Programming*, Vol. 2, Addison Wesley, Reading, Mass., 1969.
3. Urabe, M. Roundoff error distribution in fixed-point multiplication and a remark about the rounding rule. *SIAM J. Num. Anal. 5*, 2 (June 1968), 202–210.
4. Wilkinson, J.H. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1963.
5. Hull, T.E., and Swenson, J.R. Tests of probabilistic models for propagation of roundoff errors. *Comm. ACM 9*, 2 (Feb. 1966), 108–113.
6. Henrici, P. Test of probabilistic models for the propagation of roundoff errors (letter to the editor), *Comm. ACM 9*, 6 (June 1966), 409–410.
7. Brent, R.P. On the precision attainable with various floating-point number systems (to be published).
8. Kaneko, T., and Liu, B. On local roundoff errors in floating-point arithmetic (to be published in *J. ACM*).

Information Retrieval

P. Baxendale
Editor

Some Approaches to Best-Match File Searching

W.A. Burkhard
University of California, San Diego
and
R.M. Keller
Princeton University

The problem of searching the set of keys in a file to find a key which is closest to a given query key is discussed. After "closest," in terms of a metric on the the key space, is suitably defined, three file structures are presented together with their corresponding search algorithms, which are intended to reduce the number of comparisons required to achieve the desired result. These methods are derived using certain inequalities satisfied by metrics and by graph-theoretic concepts. Some empirical results are presented which compare the efficiency of the methods.

Key Words and Phrases: matching, file structuring, file searching, heuristics, best match

CR Categories: 3.73, 3.79, 4.9

This study was conducted while W. A. Burkhard was with Princeton University. Work reported here was sponsored by the National Science Foundation through grants GJ465 and GJ30126, and by Bell Telephone Laboratories, Murray Hill, N.J.

Authors' addresses: W. A. Burkhard, Department of Applied Physics and Information Science, University of California, San Diego, La Jolla, CA 92307; R. M. Keller, Department of Electrical Engineering, Princeton University, Princeton, NJ 08540.

Introduction

Suppose that X is a file, each member of which is indexed by a key, and that q is a query key possibly not in the file. The following types of searches through file X might be necessary:

Class 0 search. Is the query key in the file?

Class 1 search. Determine a key in the file which is closest to the query key.

Class 2 search. Determine all keys in the file which are closest to the query key.

The class 0 search is one which is commonly dealt with in constructing symbol tables and in retrieving records from files. We mention it mainly for comparison purposes. Classes 1 and 2 appear to be less common. A few applications immediately suggest themselves, and we suspect that others will appear with increasing use of the computer as an information retrieval tool. One use concerns keys which are possible outcomes of tests in large switching networks, such as the Bell System No. 1 ESS [3]. Another is the classification of chemical compounds, such as in *Chemical Abstracts* [2]. The common feature of these problems is that it is desirable to retrieve information, the key of which is closely related to, but not necessarily identical to, a query key. The problem has been discussed in [3], but no solutions proposed. There are, of course, rather obvious techniques for handling all of these classes. It is sufficient to search through the file from one end to the other, examining all keys. In many instances, the file to be searched is extremely large, and the exhaustive search technique above requires an exorbitant amount of time. Moreover, when the file is extremely large, it will not be possible to store it entirely in the main memory of a computer. In this case an auxiliary memory must be used, and the longer time required to transfer keys from auxiliary to main memory lengthens the required search time. Thus we are interested in techniques which do not require that every key of the file be examined. When the file is stored in auxiliary memory, a few extra calculations done in the main memory may make it unnecessary to transfer a substantial number of segments of the file between memories. For emphasis we list:

Desideratum. We seek for search algorithms which do not examine every key of the file during the search process.

With this desideratum in mind, we will consider some representations of the file suitable for storage in computer memory and the corresponding search techniques for handling classes 0, 1, and 2. We denote as *preprocessing* the general endeavor of structuring the file, as well as initially storing it in memory. The actual search process will be referred to as *searching.*

Review of Conventional Techniques

We begin by introducing some notation which will be useful in our discussions. Let $X = \{x_1, x_2, \ldots, x_n\}$ be the set of keys in the file to be searched. Let X' be the space of all possible values which a key could assume.

Let us look at the record for file structuring strategies and search algorithms for class 0, namely, to determine whether a given query key q is in the file.

Exhaustive Search

Preprocessing. Store X in memory as a linear list.

Searching. Sequentially compare q with every key in the list.

When the file X is stored in this manner, any ordering of X is as likely as another. If q is not in the file, then all of the keys in the file must be examined to ascertain this. If q is in X, then on the average only $(n + 1)/2$ comparisons must be made to locate q. The maximum number of comparisons possible here is n.

Binary Search

Preprocessing. Order X into an ascending sequence and store it sequentially in memory.

Searching. Eliminate one half of the remaining file after comparing the query key with the key in the middle position.

Of course, to use this technique, it is necessary that an ordering exist for keys in X. The maximum number of comparisons required for class 0 is $\lceil \log_2(n + 1) \rceil$, where $\lceil x \rceil$ denotes the least integer greater than or equal to x.

Direct Lookup

Preprocessing. Associate each key of X' with a distinct location in auxiliary memory. Each location contains a flag which is set to "yes" or "no" depending on whether or not the associated key is in X.

Searching. Determine the value of the flag associated with key q by direct access.

This technique is the fastest (when it can be used) since only one examination of an item in memory suffices. If the set of keys is significantly smaller than the set of addresses spanned by them on a direct-access device, much space will be wasted. One means of avoiding this problem is to use "scatter storage" techniques [4]. In certain cases, the time-efficiency of direct access can be approached while wasted space can be reduced, if these techniques are used.

There are nontrivial cases in which class 1 or 2 searches cannot be handled by these methods. Before expounding on this, let us state specifically what we mean by "closest" in these situations, and then point out the difficulty. The "distance" between two keys, x and y, is given by the value of a metric, $d(x, y)$. Specifically, a metric is any function d: $X' \times X' \rightarrow$ nonnegative integers (we say integers for simplicity) such that

(1) for every x, y in $X', d(x, y) = 0 \Leftrightarrow x = y$,

(2) for every x, y in $X', d(x, y) = d(y, x)$,
(3) for every x, y, z in $X', d(x, z) \leq d(x, y) + d(y, z)$.

The third condition is usually referred to as the *triangle inequality*. Since X is finite, we may assume d takes on values from the set $\{0, 1, 2, \ldots, m\}$ for some m.

A common example of a metric occurs when the keys are binary numbers. In this case the "Hamming distance" between two keys, defined to be the number of bit positions in which the keys differ, is easily shown to be a metric. If the bits of a key represent those "attributes" which a member of a file possesses, then a class 1 search corresponds to finding a member of the file which most closely matches a given list of attributes, while a class 2 search corresponds to finding all such members. Minsky and Papert [3] refer to this as the "best match" problem and comment on its difficulty.

Observation

The reason that conventional techniques for handling class 0 do not generally suffice for the other classes is that the linear order of keys is not significant. Because many distinct keys may be at a given distance from a query, the key space may assume aspects of a multidimensional space in either class 1 or class 2. Conventional techniques generally are only capable of exploiting a one-dimensional key space.

There are some metrics which are trivial exceptions. One is the metric

$$\begin{aligned} d(x, y) &= 1, \quad x \neq y \\ &= 0, \quad \text{otherwise.} \end{aligned}$$

Another is the case in which the keys are integer valued and $d(x, y) = |x - y|$. The first case degenerates to class 0, while binary search may be applied in the second. In the more general case, the only "obvious" way appears to revert to exhaustive search, which fails to meet our desideratum, since in general the entire file must be searched.

The file structures presented here may be viewed as what are generally called "clustering techniques." Clustering techniques have also been used in conjunction with "term classification." A goal of term classification is to identify groups of attributes of the keys for which most records in the file assign similar values. The resulting set of groups of attributes may be used to redefine the key space. Such approaches are discussed in [5, 6, 7]. They are, in a sense, orthogonal to the approach presented here, which assumes that the key space is fixed and given a priori. Our attention is directed at identifying groups of records whose keys possess certain distance relationships.

Search Techniques for Classes 1 and 2

Suppose that during the application of a search algorithm, b is the "best" key found thus far. That is, b is the key which is known to minimize $d(q, x)$ over all keys previously considered. Let $\xi = d(q, b)$. We update b with key x by the following procedure:

Update (b, x). If $d(q,x) < \xi$, then replace b with x, and ξ with $d(q, x)$; otherwise do nothing.

Every update necessitates a comparison. To meet our desideratum, we next give some conditions which imply that an update is not necessary. The discussion will be limited to class 1 for the time being; extension to class 2 will follow. We need not update with a key x in X if it is known a priori that

$$d(q, x) \geq \xi. \tag{1}$$

The cutoff criteria presented next are designed to supply sufficient conditions for inequality (1) to be satisfied.

Let x° be an element of X, and suppose that Y is a subset of X with the property that for some $k = k(Y, x^\circ)$, we have

$$\text{for every } x \in Y,\ d(x, x^\circ) \leq k. \tag{2}$$

Let us derive a sufficient condition for (1), given (2). From the triangle inequality,

$$\text{for every } x \in Y,\ d(q, x) \geq d(q, x^\circ) - d(x, x^\circ). \tag{3}$$

From (2) and (3) we have

$$\text{for every } x \in Y,\ d(q, x) \geq d(q, x^\circ) - k. \tag{4}$$

Hence the following is a sufficient condition for (1):

$$d(q, x^\circ) - k \geq \xi. \tag{5}$$

We call this the *first cutoff criterion*, since if the condition is satisfied, the entire subset Y may be eliminated from consideration, i.e. "cut off." Note that the application of this criterion depends on ξ, and is more likely to provide a cutoff if ξ is small.

Suppose that instead of (2), there is a $k = k(Y, x^\circ)$ such that

$$\text{for every } x \in Y,\ d(x, x^\circ) \geq k. \tag{2'}$$

Then we may proceed analogously to obtain

$$\text{for every } x \in Y,\ d(q, x) \geq d(x, x^\circ) - d(q, x^\circ), \tag{3'}$$
$$\text{for every } x \in Y,\ d(q, x) \geq k - d(q, x^\circ). \tag{4'}$$

In this case, the following is a sufficient condition for (1):

$$k - d(q, x^\circ) \geq \xi.$$

We call this the *second cutoff criterion*. Again this is more likely to bear fruit if ξ is small. We may combine both of these criteria if Y satisfies

$$\text{for every } x \in Y,\ d(x^\circ, x) = k. \tag{6}$$

In this case, a sufficient condition for Y to be eliminated is

$$|k - d(q, x^\circ)| \geq \xi. \tag{7}$$

We call this the *joint cutoff criterion*.

We next give a method which makes use of the joint criterion.

File Structure 1. Pick an arbitrary element x° of X. Then divide $X - \{x^\circ\}$ into subsets $X^0, X^1, X^2, \ldots, X^m$ where for all $k = 0, 1, \ldots, m$ and all $x \in X^k$, $d(x, x^\circ) = k$. Each of these subsets is then stored so that one may be distinguished from another. Clearly, by the way in which the subsets are constructed, the joint cutoff criterion can be applied to each. The search algorithm for this structure is given below.

Search Algorithm 1. Using file structure 1, to find the best match for query q:

```
Step 1.
    Let $b = x^\circ$, $\xi = d(q, x^\circ)$, and $j = d(q, x^\circ)$.
Step 2.
    Do Step 3 for $k = j, j+1, j-1, j+2, j-2, \ldots$
Step 3.
    (Apply joint cutoff.) If $| k - j | < \xi$, do Step 4.
Step 4.
    Search $X^k$ by some algorithm, possibly updating $b$,
    $\xi$ in the process.
Step 5.
    Stop. The best match is $b$.
```

For reasons to be explained presently, Step 4 is purposely left vague. To make it concrete, let us say that the search of X^k is done by exhaustive comparison.

While the order of varying k above is not necessary, it does seem to be the most natural, since once the test in Step 3 gives a negative answer because $k - j \geq \xi$ and a negative answer because $j - k \geq \xi$, the search may be terminated; for it is then known that no other value of k can give a positive answer.

Note that the element x° was chosen arbitrarily. We may make use of this fact to structure each set X^k by picking an element $x^{k,\circ}$ from X^k, then forming sets $X^{k,0}, X^{k,1}, X^{k,2}, \ldots$, such that $d(x^{k,r}, x) = r$ for all x in $X^{k,r}$. We are then equipped to reapply Algorithm 1 recursively to X^k in Step 4, instead of simply using exhaustive search. In fact this recursion may be done to any depth desired.

We next present a different technique in which X is again partioned into $X^1, X^2, \ldots$, in which we allow an initial comparison with one representative of each X^k to provide cutoff information.

File Structure 2. Divide X into a number of sets $X^1, X^2, X^3, \ldots, X^s$ such that for each $i = 1, 2, \ldots, s$ there is a key x^i and a number k^i such that for each $x \in X^i$, $d\,(x^i, x) \leq k^i$. For simplicity, we will assume here that $k = k^i$ for all i.

Search Algorithm 2. Using File Structure 2, to find the best match for query q:

```
Step 1.
    Let $b$ be an arbitrary element of $X$; let $\xi = d(b, q)$.
Step 2.
    Compute $h_i = d(x^i, q)$, and Update $(b, x^i)$ for
    $i = 1, 2, \ldots, s$.
Step 3.
    For $i$ varying over $1, 2, \ldots, s$ in order of increasing $h_i$, do Step 4.
Step 4.
    (Apply first cutoff criterion.) If $h_i - k < \xi$, then
    search $X^i$, possibly updating $b, \xi$.
Step 5.
    Stop. The best match is $b$.
```

As in Algorithm 1, the order of searching the X^i is not important. However in the particular ordering given, once the test in Step 4 gives a negative answer, the search may be stopped. This algorithm may also be applied recursively, provided that k is decreased as the level increases.

We now mention a refinement of Algorithm 2 which applies when additional structure is added to the file. Suppose that Y is a subset of X such that for some k

$$x, x' \in Y \text{ implies } d(x, x') \leq k. \tag{8}$$

(If we consider the file as an undirected graph with keys as nodes, such that two nodes x, x' are connected only when $d(x, x') \leq k$, then such a set Y is commonly called *complete*. A complete set which is maximal, in the sense that no additional nodes can be added while retaining the complete property, is referred to as a *clique* of the graph. Equivalently, a set Y is a clique if the converse of (8) also holds. Any subsequent references to these terms will pertain to this correspondence.)

If x° is *any* element of Y, we may, by the first cutoff criterion, eliminate $Y - \{x^\circ\}$ if

$$d(q, x^\circ) - k \geq \xi. \tag{9}$$

Suppose that we are searching within a set Y satisfying (8). Inequality (9) suggests that as well as updating with each key the value ξ as in previous algorithms, we also record the maximum value δ of $d(q, x) - k$ as we search. Then if at some time $\delta \geq \xi$ is satisfied (recalling that ξ generally decreases as the algorithm proceeds), we may immediately cease the search of Y, knowing that no further improvement is possible. We call this the *clique criterion.* Note that it may be applied in Algorithm 2 when the X^i have the clique property. We emphasize the distinction that this criterion is applied *within* the search of X^i, whereas the first cutoff criterion is applied *before* the search of X^i. We now elaborate on the methods which employ the clique criterion. The reader may observe that the methods also work when "clique" is replaced by "complete set." That is, the sets need not be maximal.

File Structure 3. Find a set $C = \{X^1, X^2, \ldots\}$ of cliques such that every key in X is in at least one element of C. An arbitrary element x^i of X^i will be designated *clique representative.* The file is stored so that cliques in C are distinguishable from each other. It is noted that the keys of particular records may appear in several cliques. Search Algorithm 2 may be employed using this file structure; the clique criterion may be utilized within each clique.

It is noted that as the number of cliques in C decreases, the total memory for pointers, etc., required by this structuring of X decreases. Moreover, it is possible to strengthen the notion of a clique representative thus decreasing the average number of updates required in searching. The idea is to select as representatives those keys which are in the greatest number of cliques. The following algorithm determines such a set of clique representatives.

```
Step 1.
    Determine how many elements of $C$ each key of $X$ is in.
Step 2.
    Do Step 3 for $i = 1, 2, \ldots$ until all elements of $C$ have been considered.
Step 3.
    Select any key $x^i$ in $X^i$ as representative for clique $X^i$ provided $x^i$ is in no fewer elements of $C$ than any other key in $X^i$.
Step 4.
    Stop. The set of representatives is $\{x^1, x^2, \ldots\}$.
```

To utilize this notion of clique representative, the file X is stored as blocks $\{B_1, B_2, \ldots\}$. Each block B_j contains all cliques in C with the same clique representative x^j. With common clique representatives, the first cutoff criterion can often eliminate several cliques at a time in the search process with only one comparison. The following algorithm utilizes File Structure 3 with the generalized notion of clique representatives.

Search Algorithm 3. Using File Structure 3, this algorithm finds a best match b in X for query key q.

```
Step 1.
    Let $b$ be an arbitrary element in $X$; let $\xi = d(q, b)$.
Step 2.
    Do Step 3 for $j = 1, 2, \ldots$ until all blocks of $X$ have been considered.
Step 3.
    Update $(b, x^j)$; (apply first cutoff criterion) if $d(q, x^j) - k < \xi$, then do Step 4.
Step 4.
    Do Step 5 for each clique $c$ in block $B_j$.
Step 5.
    Set $\delta = 0$. Do Step 6 for each $x$ in $c$ while $\delta < \xi$ (the clique cutoff criterion).
Step 6.
    Update $(b, x)$. Set $\delta = \max(\delta, d(q, x) - k)$.
Step 7.
    Stop. The best match is $b$.
```

All of our simulations involving clique structuring use the generalized notion of clique representative.

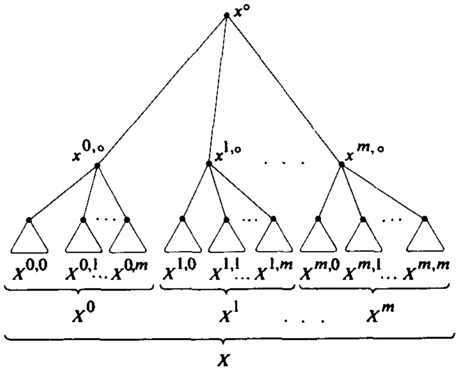
Preprocessing Phase

As mentioned before, all file structuring is done initially in a preprocessing phase. We discuss briefly the amount of work required in this phase for two of the file structures presented.

File Structure 1. First choose as arbitrary element x°. The sets $X^1, X^2, \ldots$ are then created by what amounts to a sort on the value of $d(x^\circ, x)$ where x represents a typical key. The sets $X^1, X^2, \ldots$ may be delineated either by storing a special set of pointers for the beginning location of each set, or by storing the set of indices with the sets themselves and performing a sequential search for the proper group during the search algorithm.

If multiple-level structuring is desired, then this process is simply repeated on the individual sets to form subsets, and so on. The overall structure may be envisioned as a tree as shown in Figure 1.

Fig. 1. File Structure 1 viewed as a tree.



File Structure 2. (Since File Structure 3 is a more structured form of File Structure 2, we omit consideration of the latter here.)

File Structure 3. The file is stored in terms of cliques. The blocks may be stored sequentially and processed one after the other, in which case the active work space (memory needed to perform the required computations) by the search may be fixed and independent of the size of the file.

In light of previous comments, the set C is determined with as few cliques as possible. A *minimal cover* for X is a set C of cliques such that: (1) every key in X is in at least one element of C; and (2) for no smaller set C' does property 1 hold. In terms of economy of space, we would like to determine a minimal cover for X. A minimal cover for X may be computed by first determining the set C of all cliques of the graph (of the file) and then extracting a minimal set cover. An algorithm known as the Bierstone algorithm for computing the set of all cliques of an undirected graph is given in [8, 9].

The following algorithm yields an approximate minimal cover. Let C be the set of cliques determined by the Bierstone algorithm. Let M be empty and all keys be untagged.

```
Step 1.
    Do Step 2 while some key is untagged.
Step 2.
    Determine a clique $c$ in $C$ such that $c$ contains no fewer untagged keys than any other clique of $C$. Place this clique in set $M$ and tag all keys of $c$.
```

Step 3.

Stop. The set M contains an approximate minimal cover.

We use the adjective "approximate" here since the algorithm described above does not in general yield a minimal cover; this approximation is used in all of our simulations since the computation of the minimal cover is an extremely time-consuming process.

File Maintenance

So far the discussion has centered on fast search algorithms for the given file structures. Another problem may occur if records must be added or deleted after the initial structuring has occurred. Of course the file could be restructured *ab initio*, but this is useful only if additions and deletions occur infrequently. We wish to make a few remarks here concerning the manner in which existing files may be modified without repeating the initial structuring.

In the case of File Structures 1 and 2, the addition of a new key simply involves locating the corresponding set X^k and adding the key to this set. If multiple levels of structuring are used then this process is iterated for each level. If there is no corresponding set X^k then one must be created with the new key as its only element. Of course, once the set X^k is found, the insertion may or may not be trivial, depending on the details of implementation. The point is that the imposed structuring does not cause the insertion process to be appreciably more complicated than it would be for, say, an indexed-sequential-access file.

The deletion of a key in Structures 1 and 2 follows a process similar to that above, with special consideration for the case in which the key to be deleted is the representative x^0. In this case, the key cannot simply be deleted, as it is essential for the structure information. Instead an extra bit must be used for each key which denotes whether the key actually corresponds to a record or not. The search algorithm is modified correspondingly to ignore keys which do not correspond to records. This involves testing the extra bit in the Update procedure.

File Structure 3 may be maintained in the manner described below. In the discussion, we assume that a covering set of cliques is used as previously described. The extensions to other versions will be evident.

Notice that the addition of a new key to the file cannot decrease the number of cliques needed in a minimal cover of the file. However, adding a new key may necessitate increasing the number of cliques in a minimal cover. Consider adding a key $\hat{x}$ to the file. We may use the existing file structure to locate a key closest to $\hat{x}$. If this key is no closer than $k + 1$, then $\hat{x}$ is added to the file structure as a clique containing only itself. The set of clique representatives is then enlarged to include $\hat{x}$. If the key closest to $\hat{x}$ is closer than $k + 1$, a complete set may be determined consisting only of keys contained in the clique containing $\hat{x}$. This set may be inserted into the cover. The set determined in this manner may not be a clique; however, as remarked earlier, as long as the newly created set is complete, the search algorithm will suffice.

Obviously, deleting a key $\hat{x}$ from the file can never increase the number of cliques in a cover of the file. We may use repeated applications of the existing search technique to delete copies of $\hat{x}$ in the file. If $\hat{x}$ is a clique representative, it can be retained as structure information and simply be tagged as not being in the file, as in the case of the other file structures.

For either insertion or deletion, it may be expedient, in terms of subsequent performance of the search algorithm, to redefine the set of clique representatives. Of course, this redefinition may necessitate reorganizing the block structure of the existing file implementation.

While the three file structures were conceived with time-invariant files in mind, it is seen that they are useful even in an environment necessitating changing file membership. The retrieval performance of perturbed files maintained by these techniques has not been studied; however, it seems plausible that for small perturbations the performance will not be degraded significantly.

Experimental Results

The techniques presented here were tried experimentally—a total of 1000 keys of 30 bits, each of which was obtained using a linear shift register generator [10]. The characteristic polynomial of the generator is $1 + X^3 + X^{31}$. The Hamming distance was used as a metric, and a number of class 1 experiments were run, each using 80 trial queries, none of which exactly matched any key in the file.

Tables I and II compare File Structures and Algorithms 1 and 3 respectively. It should be emphasized that the numerical quantities presented are sensitive to arbitrary choices made in the preprocessing phase. Another set of experiments was run in which the keys were divided in half to produce a file of 2000 keys of 15 bits each. The results in this case were substantially better and are given in Table III. We also show in Table III a comparison between class 1 and class 2 searches.

We believe that the difference in the results between the two key sizes is due to the relative values of the size of the file and the number of possible values in the key space. For an n-bit key, there are 2^n such values. The trees with 30-bit keys were run with 200 or 500 elements in the file, whereas those with 15-bit keys were run with 2000 elements in the file. Thus, in the latter case, the probability of a closer match is higher, and the probability of achieving a cutoff is correspondingly greater.

Table I. Results for File Structure 1, with three levels of structuring, and Search Algorithm 1 on 30-bit keys (80 trials were used on each file).

		Number of Comparisons/Query				Final ξ Value	
File Composition Key Numbers	Number of Keys	Mean (%)	Standard Deviation (%)	Max. (%)	Min. (%)	Max.	Min.
401–600	200	76.9	19.6	100	1.0	9	5
601–800	200	76.8	28.1	100	1.0	10	5
801–1000	200	75.7	25.3	100	1.0	9	4
500–1000	500	69.4	21.9	99.2	0.4	9	4

Table II. Results for File Structure 3 and Search Algorithm 3 on 30-bit keys (80 trials were used on each file).

	Number of Comparisons/Query			
File Composition Key Numbers	Mean (%)	Standard deviation (%)	Max. (%)	Min. (%)
401–600	82	9	97	50
601–800	82	10	99	51
801–1000	78	11	99	33
501–1000	67	11	87	39

Table III. Results for File Structure 1, with 3 levels of structuring, and Search Algorithm 1 on 2000 keys of 15 bits each (80 trials were used in each case). Final ξ values varied between 0 and 2.

	Number of Comparisons/ Query		
	Mean (%)	Max. (%)	Min. (%)
Class 1 Search	13.4	39.2	0.1
Class 2 Search	26.7	61.5	0.1

Conclusions

Our initial goal—to determine some heuristics suitable for file structuring in particular situations—has been met, and our results are contained here. The empirical results suggest that these heuristics do offer techniques which allow fast file searches. A question naturally arises—how good are these techniques? A detailed analysis of the techniques could shed some light on questions such as: (1) How much of an average file can be eliminated using cutoff criteria (of some type)? (2) Is it possible to take into account additional knowledge of the contents of the file and thereby "do better" than in the average case? and (3) Are there other clustering techniques which do as well as or better than the two mentioned here? The list of unanswered questions can be continued.

File Structure 3 was not empirically investigated as thoroughly as File Structure 1 since the computation of all cliques of an undirected graph using the Bierstone algorithm is an extremely lengthy process. Another clique finding algorithm is now known which will shorten the process, and the empirical study of File Structure 3 can be completed in a more thorough manner [11].

Summary

The study described here is concerned with the general problem of efficiently searching files in the following situations: (1) find a key in the file closest to a given query key, and (2) find all keys in the file closest to a given query key. By "closest" we mean according to some suitable measure of distance, specifically a "metric" in the mathematical sense. These situations may arise in information retrieval applications in which members of a file are keyed to a number of numerically-represented attributes. Our search algorithms have the following common basis: let *b* be the "best" key found at a certain point in the application of the algorithm. Based on certain "cutoff criteria" which depend on *b*, subsets of the file remaining to be examined are eliminated without explicitly comparing the keys in these subsets with the query key. Our experimental results (on randomly-generated files) indicate that large segments of the file may be eliminated from consideration using these cutoff criteria.

Acknowledgment. The authors wish to thank Dennis Leung for assistance in programming some of the algorithms presented here.

Received March 1972; revised June 1972

References

1. Chang, H.Y., and Thomas, W. Methods of interpreting diagnostic data for locating faults in digital machines. *Bell Syst. Tech. J.* (1967), 289–317.
2. Wipke, W.T. Private communication. Dep. of Chem., Princeton U., Princeton, N.J.
3. Minsky, M., and Papert, *S. Perceptrons: An Introduction to Computational Geometry* M.I.T. Press, Cambridge, Mass., 1969.
4. Morris, R. Scatter storage techniques. *Comm. ACM, 11,* (1 Jan. 1968), 38–44.
5. Jackson, D.M. Classification, relevance, and information retrieval. *Advances in Computers* Vol. 11, Academic Press, N.Y., 1971.
6. Litofsky, B., and Prywes, N.S. All-automatic processing for a large library. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 323–331.
7. Salton, G. Experiments in automatic thesaurus construction for information retrieval. Proc. IFIP Congress 1971, North Holland Pub. Co., Amsterdam.
8. Augustson, J.G., and Minker, J. An analysis of some graph theoretical cluster techniques. *J. ACM 17,* 4 (Oct. 1970), 571–588.
9. Mulligan, C.D., and Corneil, D.G. Corrections to Bierstone's algorithm for generating cliques. *J. ACM 19,* 2 (Apr. 1972), 244–247.
10. Golomb, S.W. *Shift Register Sequences.* Holden-Day, Inc., San Francisco, 1967.
11. Bron, C., and Kerbosh, J.A.G.M. Finding all cliques of an undirected graph. (To be published.)