Information Processing Letters 19 (1984) 61–65
North-Holland

31 August 1984

PARTIAL MATCH RETRIEVAL IN IMPLICIT DATA STRUCTURES

Helmut ALT

Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, U.S.A.

Kurt MEHLHORN

Fachbereich Informatik, Universität des Saarlandes, 6600 Saarbrücken, Fed. Rep. Germany

J. Ian MUNRO

Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Communicated by J. Nievergelt
Received 10 December 1983
Revised 20 March 1984

Keywords: Data management, analysis of algorithms, combinatorial problems, data structures

1. Introduction

We consider a set of data which contains n records, each of them consisting of k keys.

A partial match query is one in which an arbitrary subset of the keys of a record is specified. Problems of this type have been studied by a number of researchers including Bentley [3], Ghosh and Abraham [6] and Rivest [9]. Our attention, unlike most, though not all, previous work, is restricted to structures which are *implicit* in that no information other than the data itself (and the number of records) is explicitly stored. That is, we are interested in storage schemes which for fast retrieval retain no pointers and represent the file of n k-key records as a simple n by k array T. Any structural information must be encoded in the order of the records.

The model of computation is the simple comparison model, as it is assumed that the keys are chosen from some arbitrarily large space. The measure of difficulty to perform a search is the maximum number of comparisons required when any j of k keys have been specified.

For example, in the case of a 2-key file, if the file were sorted according to the first key searches could be performed quickly if key 1 were given. However, one could do no better than a sequential search if only the second key were specified. Clearly some 'balance' between the ordering on the keys is required.

2. An upper bound for partial match retrieval

The following procedure (cf. Munro [7]) orders the records in a way such that the partial match retrieval can be done efficiently. Initially, the only segment is the entire file itself. Assume the keys are numbered $0, 1, \ldots, k-1$.

```
procedure Order;
begin i := 0
   while there remain segments of more than 1
                element do
   begin
      for each segment remaining from the pre-
                vious stage do
      begin
```

```
        Move the median element under key i
            of the segment to the middle
            position of the segment;
        Move all elements less than this median
            above it and all elements greater,
            below. This produces 2 new seg-
            ments;
    end
    i := (i + 1) mod k
  end
end
```

Let us consider, e.g., the case $k = 2$.

In the first step the records are separated with respect to the median element under key 1, which leaves 2 segments. They are separated with respect to their median elements under key 2, the resulting 4 segments are separated with respect to their medians under key 1, etc.

In order to do partial match retrieval in an implicit data structure thus constructed a generalized binary search is possible. The segments to be searched recursively are the same ones as the segments during the construction of the structure. So comparisons will always be made with the middle position of a segment and with respect to the key by which the segment is split. If that key is not specified in the partial match query, no information can be gained by a comparison and both subsegments of the current segment have to be searched recursively. If the key is specified in the input, a comparison can be made and like in binary search one of the subsegments can be excluded. Consider the moment when the search algorithm has rotated once through keys $1,\ldots,k$ and returns to comparisons with key 1. It has made some constant number c of comparisons. It also has split in half the size of segments in each phase and doubled the number of segments in each of the $k - j$ phases corresponding to a key not specified in the input.

So for the number $T(n)$ of comparisons, we get the recurrence equation

$$T(n) = 2^{k-j}T(n/2^k) + c$$

which, in the case $j < k$, has the solution

$$T(n) = O(n^{1-j/k})$$

and, in the case $j = k$,

$$T(n) = O(\log n).$$

So, altogether, we have the following theorem.

Theorem 1. *A table of* n *records, each consisting of* k *keys can be arranged as an* n *by* k *array so that any record specified by* j *of its* k *keys can be found in* $O(n^{1-j/k})$ *comparisons if* $j < k$ *and* $O(\log n)$ *comparisons if* $j = k$. *No additional information, other than the values* n *and* k *need be stored permanently with the structure.*

For example, a telephone directory with n entries can be stored in such a way that it is possible to search for a name as well as a phone number in $O(\sqrt{n})$ time. The next section will show that this upper bound is optimal.

3. Lower bounds

In [7] it was shown that the upper bounds of Theorem 1 are optimal if the keys in each column of the array T are inserted in some partial order. We now prove that in the case $j = 1$ the lower bounds hold even if we do not have this restriction.

Throughout this section we assume that the different keys of a record are *independent*, i.e., any k-tuple of keys can appear as a record.

Now the idea of the lower bound proof is the following.

Every method for partial match retrieval consists of rules, how to order the records in the table T and of the retrieval algorithm. If arbitrary sets of records are stored in T, it cannot be expected that in each of the k columns of T the elements of that column always appear sorted, for example, in ascending order. For example, in a telephone directory the column of names appears in ascending order and can therefore be searched efficiently. The column of numbers, however, may appear in any arbitrary permutation of the ascending order and a linear search has to be done in order to find a given number in a telephone directory.

We will show that, in general, no matter by which rules the records are stored in the table T,

there must be some of the columns of T, which can appear in 'many' different permutations of the ascending order. This fact makes it hard to search if one of those keys is specified.

So, by just looking at single columns of T, we get a reduction to partial match retrieval from the following one-dimensional problem ('searching semisorted tables' (SST), cf. [1]):

Assume that n elements of some linearly ordered universe are stored in an array A. They need not be stored in ascending order but according to some permutation π of the ascending order, which means that the *jth* smallest element is stored in $A[\pi(j)]$, $1 \leqslant j \leqslant n$. Assume that p such permutations are allowed ($1 \leqslant p \leqslant n!$). The problem is to search in A, without knowing which one of the p permutations is used.

We want to find a lower bound on the number of comparisons necessary to solve SST as a function of n and p. For both extreme values of p, SST has obvious optimal solutions:

If $p = 1$, a modified binary search can be done and SST can be solved in O(log n) steps.

If $p = n!$, i.e., all permutations are allowed, linear search, i.e., $\Omega(n)$ steps are necessary.

The following theorem gives a lower bound for arbitrary values of p.

Theorem 2. *If* n *elements can be inserted in an array* A *in* p *different permutations of the ascending order, then searching requires* $\Omega(p^{1/n})$ *comparisons.*

(A similar theorem was proven concurrently and independently by S. Cook.)

Proof. Let Π be the set of p permutations and consider a search algorithm for Π making at most s comparisons for any search. So it can be represented by a comparison tree of height s. Its nodes are labelled with array positions $\ell \in \{1,\ldots,n\}$, the outcomes of a comparison may be $<$, $>$ or $=$ in which case the algorithm stops. The leaves of the tree denote unsuccessful searches. Furthermore, denote by

$$\Pi_{i_1,\ldots,i_k} = \{\pi \in \Pi \mid \pi(j) = i_j \quad \text{for } 1 \leqslant j \leqslant k\}$$

the set of permutations in Π having the initial segment $i_1,\ldots,i_k$. (We allow $k = 0$, i.e., the empty initial segment ϵ, and define $\Pi_\epsilon = \Pi$.) It can be shown that any initial segment of length k of permutations in Π can be extended in at most s ways to an initial segment of length $k + 1$.

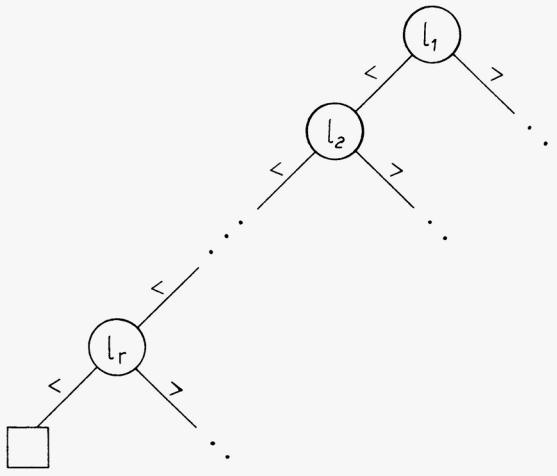

Fig. 1.

Lemma 2.1

$$\left|\{\pi(k+1) \mid \pi \in \Pi_{i_1,\ldots,i_k}\}\right| \leqslant s$$

for ***all*** $i_1,\ldots,i_k \in \{1,\ldots,n\}$.

Proof. Suppose $a_1,\ldots,a_n$ are the elements of the array in ascending order and suppose that they are stored according to any $\pi \in \Pi_{i_1,\ldots,i_k}$.

For reasons of clarity let us first consider the case $k = 0$, i.e., count the number of possibilities for $\pi(1)$, the array location, where the smallest element can be stored. Assume, that a search is performed with an argument x, which is smaller than any of the elements in the array. The search will traverse the comparison tree T on its leftmost path, since all outcomes of comparisons will be $<$. Assume that $\ell_1,\ldots,\ell_r$, $r \leqslant s$, are the labels on that path, i.e., the array locations x is compared with (cf. Fig. 1).

Claim 2.1.1. *The smallest element* a_1 *of the array must be stored in one of the locations* $\ell_1,\ldots,\ell_r$.

Proof. Assume otherwise. Then, traversing the tree

with a_1 instead of x would result in the same path, since all outcomes of comparisons would be $<$, too. Thus, the algorithm would give the same answer for a_1 as for x, namely that it is not in the set, a contradiction. □

Proof of Lemma 2.1 (*continued*). So there are only $r \leqslant s$ possibilities to store the smallest element, i.e., the lemma is proven for $k = 0$.

The proof can be generalized to any arbitrary $k \leqslant n$: Consider a search with an argument x such that $a_k < x < a_{k+1}$. Since the elements $a_1, \ldots, a_k$ are stored in $A[i_1], \ldots, A[i_k]$, a comparison $x \,?\, A[\ell]$ has outcome "$>$" if $\ell \in \{i_1, \ldots, i_k\}$ and outcome "$<$" otherwise. This identifies a unique path in the comparison tree and after at most s comparisons the algorithm will answer that x is not in the set. Let $\ell_1, \ldots, \ell_t$ ($t \leqslant s$) be the labels of the nodes of the above path, i.e., the positions x is compared with. We claim that

$$\pi(k+1) \in \{\ell_1, \ldots, \ell_t\}.$$

Assume otherwise. Then the comparisons to $\ell_1, \ldots, \ell_t$ would have the same outcome if we traverse T with a_{k+1} instead of x so the answer "not in the set" would be given for a_{k+1} also, a contradiction to the correctness of our algorithm.

So there are only $t \leqslant s$ possibilities to store a_{k+1} which proves Lemma 2.1. □

Proof of Theorem 2 (*continued*). Clearly there is only one initial segment of length 0. Applying Lemma 2.1 repeatedly yields that there are at most s^k initial segments of length k, $0 \leqslant k \leqslant n$. Since Π is the set of all initial segments of length n we have

$$p = |\Pi| \leqslant s^n,$$

which proves Theorem 2. □

Let us return to partial match retrieval. Different sets of records stored in the table T can be in different 'relative positions'. A relative position can be characterized, e.g., by the sequence of permutations $(\pi_2, \ldots, \pi_k)$ in which columns $2, \ldots, k$ would appear if the records were sorted by the first key. Because of the independence of the keys there are $(n!)^{k-1}$ different relative positions possible if we consider all possible sets of n records.

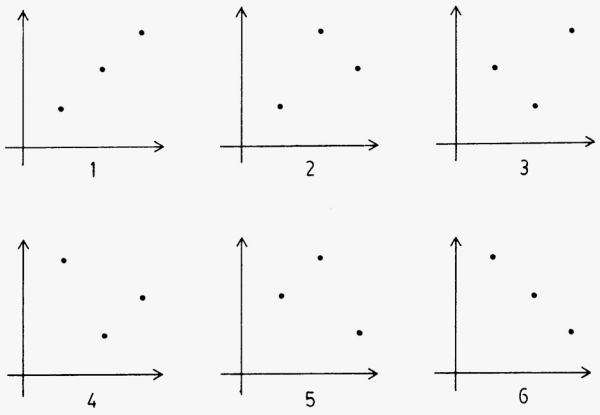

Fig. 2.

The 6 possible relative positions in the case, where $n = 3$, $k = 2$ and the universe is the set of real numbers, are illustrated in Fig. 2.

Now let us assume that a method allows to store key i ($i = 1, \ldots, k$) in the table T in p_i different permutations of ascending order. In order to be able to store records of all possible relative positions, it must be

$$p_1 \cdot p_2 \cdot \ldots \cdot p_k \geqslant (n!)^{k-1}. \tag{1}$$

This gives us a lower bound for the 1-key partial match retrieval. There must be one p_i which is $\geqslant (n!)^{(k-1)/k}$. By Theorem 2, searching for that key requires $\Omega[((n!)^{(k-1)/k})^{1/n}] = \Omega(n^{1-1/k})$ comparisons. So we have the following theorem.

Theorem 3. *If* n *records each consisting of* k *keys are arranged in an* n *by* k *table and no other information is retained* (*other than* n *and* k), *then* $\Omega(n^{1-1/k})$ *comparisons are necessary, in the worst case, to find a record specified by one of its keys.*

4. Conclusion, open problems

The problem of Theorem 2 ('searching semisorted tables') has been investigated in a separate paper [1]. It could be shown that the same lower bound holds for nondeterministic algorithms and for the average number of comparisons. Because of those results, the same holds true for Theorem 3.

The results of this paper could be generalized in different directions:

The authors belive that the lower bound in Theorem 3 not only holds if one of the keys is specified, but for any arbitrary number $j < k$ of keys. They gave a proof idea in [2] but some essential steps are still missing.

There also is the interesting problem of lower and upper bounds in a dynamic environment, i.e., if insertions and deletions are allowed (dictionary problem). What is the complexity of these operations themselves? Recent papers by Frederickson [4,5] give algorithms whose upper bounds nearly match the lower bounds for the static case we prove here:

There is an implicit data structure such that retrieval, if j keys are specified is possible in time $O(n^{1-j/k})$ for most of them (just in one case the time required is $O(n^{1-j/k} \log n)$). Insertion and deletion can be done in time $O(n^{1-1/k+\epsilon})$ for any $\epsilon > 0$.

Another way to generalize our results could be to use a more general model of computation. A first step would be to allow comparisons of two table elements instead of comparing only the key we are searching for with table elements.

An even more general model of computation was introduced by Yao [10]: The computational process consists of probes into the table T. The location of the *ith* probe depends on the search key x and the contents of the locations of the first $i - 1$ probes. This model includes, e.g., hashing.

For the problem of searching semisorted tables (cf. Theorem 2) a reduction of the general model to a more general comparison tree model is possible, but no lower bounds for the problem in the latter model are known, either.

References

[1] H. Alt and K. Mehlhorn, Searching semisorted tables, Rept. CS-82-82, Computer Science Dept., Pennsylvania State Univ., 1982.

[2] H. Alt, K. Mehlhorn and J.I. Munro, Partial match retrieval in implicit data structures, Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 118 (Springer, Berlin, 1981) pp. 156–161.

[3] J.L. Bentley, Multidimensional binary search trees used for associative searching, Comm. ACM 18 (9) (1975) 509–516.

[4] G.N. Frederickson, Implicit data structures with fast update, IEEE FOCS (1980) pp. 255–259.

[5] G.N. Frederickson, Implicit data structures for the dictionary problem, J. ACM 30 (1) (1983) 80–94.

[6] S.P. Ghosh and C.T. Abraham, Application of finite geometry in file organization for records with multiple-valued attributes, IBM J. Res. Develop. 12 (1968) 180–187.

[7] J.I. Munro, A multikey search problem, Proc. 17th Ann. Allerton Conf., 1979.

[8] J.I. Munto and H. Suwanda, Implicit data structures, J. Comput. System Sci. 21 (1980) 236–250.

[9] R.L. Rivest, Partial match retrieval algorithms, SIAM J. Comput. 5 (1) (1976) 19–50.

[10] A.C. Yao, Should tables be sorted?, J. ACM 28 (3) (1981) 615–628.