1977

Removing Duplicate Rows of a Bounded Degree Array Using Tries

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
77-240

Removing Duplicate Rows of a Bounded Degree Array Using Tries

Douglas Comer

Department of Computer Science

Purdue University

West Lafayette, IN 47907

July 1977

CSD-TR 240

Removing Duplicate Rows of a Bounded Degree Array Using Tries

Keywords and Phrases: Trie, Trie index, digital searching
CR categories: 3.73, 3.74

1. Introduction:

A 2-dimensional array A is of bounded degree d if each element of A is an integer, a, such that $0 \leq a \leq d$. The problem of removing duplicate rows of an array with bounded degree arises in many applications. For example, in a relational database system [1], a relation may be thought of as a 2-dimensional array in which each column is a domain. The process of projecting a relation over a subset, P, of its domains consists of eliminating all domains (columns) not in P and removing duplicate rows from the resulting subarray. For a relation, R, with domains "employee name" and "county of residence", a projection of R over "county of residence" would be a list of all those counties in which employees reside. Since the cost of extracting a subset of columns from a given row is usually small, the difficulty in computing such a projection is essentially that of removing duplicate rows from an array.

In order to quantify the cost of various solutions to the problem of removing duplicate rows, we state it as follows:

Problem 1: (row compression) Let A be an $n \times m$ array of bounded degree d, and let k be the number of distinct rows of A. Find a $k \times m$ array, A', such that each row of A appears in A'.

Since we are interested in the space required to generate A' as well as the time required, we will assume that A is stored on secondary storage by row and need not be kept in main memory. We further assume that A is "unordered" in the sense that the rows are not arranged lexicographically (the ordering of A' is discussed below).

Several solutions for Problem 1 are reviewed below which are based on well known algorithms. Knuth [4] is a good reference for both the detail and analysis of the sorting and hashing algorithms mentioned.

Solutions to Problem 1:

1) Insertion: For each row, r, of A, if r is not in A' then insert it.

2) Comparative Sort: Read A into memory and sort it using a comparative sort (like quicksort), placing the rows in lexicographic order. Then compare adjacent rows of A and eliminate duplicates.

3) Radix Sort: Proceed as in #3 using a radix sort.

4) Hashing: Hash the rows of A into a table, skipping duplicates and adding the rest to A'.

Each of these solutions may have advantages over the others depending on n, m, and k. Method 1, insertion, is easy to program and requires only space for A'. To compare two rows takes m comparisons. Thus, if A' is kept ordered and a binary search is used, the running time is $O(mk^2 + nm \log_2 k)$, where the term mk^2 accounts for inserting a row in order. By using k extra locations for pointers and not actually moving the rows of A', the time can be reduced to $O(k^2 + nm \log_2 k)$. For small k, the running time is

reasonable even if n is large. But if k is as large as n, the running time is $O(n^2 + nm \log_2 n)$. We shall see that other methods do much better.

Method 2, a comparative sort, is practical when $k = n$ (i.e. there are only a few duplicate rows in the array), and n is large. The running time is only $O(nm \log_2 n)$ and the space required is nm plus locations for n pointers (to eliminate moving rows of A). Since the array has bounded degree, an immediate improvement in running time to $O(nm)$ can be obtained by using a radix sort, Method 3, with radix d. Both of these sorting methods also have the advantage that the rows of A' can be generated in either sorted order or in the original order.

Since radix sorting requires time proportional to the size of the input, no faster method can be found in the general case. In a paged memory environment, however, processing the array in row order is far less expensive than processing by column as is done in a radix sort. In such cases Method 4, hashing,would be desirable since it processes A by row in one pass while still using only $O(nm)$ steps. But there are drawbacks to hashing as well. First, the new array, A', can no longer be output in sorted order without a separate sorting procedure. And secondly, since the number of distinct rows of A is not known a priori, the hash table may be allocated (and initialized) much larger than necessary.

We seek a solution to Problem 1 which meets the following criteria:

1. No more than $O(nm)$ steps are required,
2. No more than $O(km)$ space is taken (where k is not known a priori),
3. The new array, A', can be generated in either sorted order or in the original order, and
4. The array A is processed by row in a single pass.

2. A Trie Based Method:

In this section a solution for Problem 1 which meets the four criteria above is discussed. Like the other methods presented, this one is based on a well known idea, that of a trie index. The definition of a trie will be given first, and then its use in solving Problem 1 will be discussed.

Tries were introduced by de la Briandais [2] and Fredkin [3] for the storage of character data. Sussenguth [6] proposes an alternative implementation which requires more time to access but saves space. In this paper we will give a slightly modified definition of a trie and relate a trie to an array of bounded degree.

Definition 1: Let A' be a $k \times m$ array of bounded degree d such that row $i \neq$ row j, $i \neq j$. A trie for A' is a tree with k leaves, each of which lies at depth[1] m such that:

1. For each row of A' there is a path in the trie from the root to a leaf with the sequence of labels on edges in the path equal to the sequence of elements of the row, and
2. Each such path in the trie has a sequence of labels on the edges equal to a row of A'. □

Figure 1 shows an array and the trie for it.

To search for a row in the trie, one begins at the root and follows those edges with labels which are the same as the elements of the row in question. An important property of tries is that the decision about which edge to follow at a given node can be made in constant time. Fredkin's implementation uses an array of pointers at each node to achieve this property. To follow an edge with label p, one follows the p^{th} pointer.

---

[1] the root of a trie lies at depth 0, the sons of a node at depth i lie at depth $i+1$.

Of course, the range of label values determines the storage necessary at each node. For a trie corresponding to an array with bounded degree d, each node would have d possible pointers.

Since the decision about which edge to follow takes constant time, searching for a row of length m requires O(m) steps. Adding a row to an array corresponds to adding a leaf to the trie and establishing a path from the root to the leaf. Knuth [4] provides a detailed algorithm for insertion and shows that it will require only O(m) steps. Thus, to build a trie for a k x m array will require O(km) steps.

We now address the solution of Problem 1 using a trie. The method is quite simple: for each row, r, of A, if r is not in the trie then add it. Since the search and insertion algorithms are nearly identical, they can be merged. A search continues until a null pointer is found, at which time the addition of the new path begins. Therefore, any row can be processed in O(m) steps, so A can be processed in O(nm) steps, the minimum possible. Furthermore, if storage is allocated on demand, only O(km) space will be used for the trie (even though k is not known before the trie is begun).

Consider the four criteria for a row compression algorithm outlined in the previous section. We have already shown that a trie is fast as possible and uses only O(km) space. To see that a trie can be used to order A', observe that a preorder traversal of the trie (details of a traversal can be found in [5]) will generate A' in sorted order. To obtain A' in the original order, the trie can be constructed while A' is being generated: new rows are output as they are inserted into the trie while duplicate rows are ignored. Since A is processed by row, the trie based method meets the criteria listed.

3. Other Applications:

The problem discussed so far deals with finding rows in an array which are equal. In some applications, however, we want to group rows together which are isomorphic (equal up to a renaming of values). For a binary matrix, a trie can be used to find classes of isomorphic rows using almost the same method outlined above. As each row is inserted into the trie, the first element is examined. If it is a 0, then the remaining elements in the row are inserted as usual. If the first element is a 1, then the compliment of the row is inserted. Each leaf in the trie is the head of a list of the row numbers of all rows which terminate there and represents a class of rows which are isomorphic. By chaining the leaves together as they are added to the trie, a list of the classes of isomorphic rows can be obtained in $O(nm)$ time for an array of n rows and m columns.

For arrays with degree higher than 2, the process of finding classes of isomorphic rows becomes more costly. There are $d!$ different permutations of the values used in the array, and therefore, $d!$ different namings possible. With the technique used in the binary case, that of examining the first element of a row, $1/d$ of the namings can be eliminated. By inserting $(d-1)!$ paths in the trie for each row all the renamings can be accounted for. For fixed d, the process still requires only $O(nm)$ steps but is practical only for small values of d.

4. Conclusions:

We have shown that a trie index can be an efficient method for eliminating duplicate rows from an array of bounded degree. The method is fast, uses storage only as necessary, processes the input array by row, and can be used to generate the output in sorted order. One application is that of computing the projection of a relation in a relational database system.

References

[1] Date, C.J., An Introduction To Database Systems, Addison Wesley, 1975.

[2] de la Briandais, R., "File Searching Using Variable Length Keys," Proc. Western Joint Computer Conference, 1959, 295-298.

[3] Fredkin, E., "Trie Memory," CACM 3:9 (Sept 1960) 490-499.

[4] Knuth, D., The Art of Computer Programming, vol 3, Sorting and Searching, Addison Wesley, 1973.

[5] Reingold, E., Nievergelt, J., and Deo, N., Combinatorial Algorithms Theory and Practice, Prentice Hall, 1977.

[6] Sussenguth, E., "Use of Tree Structures for Processing Files," CACM 6:5 (May 1963) 272-279.

1	3	1
3	3	1
1	1	1
3	3	2
2	1	3
3	2	2
1	3	3
3	2	1

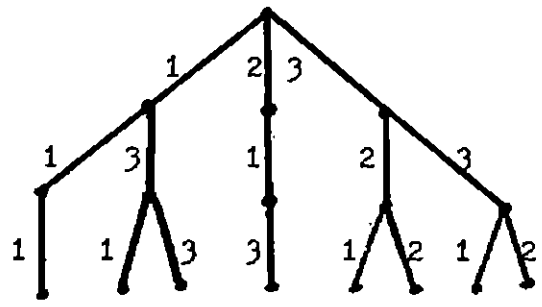

Figure 1. An array of degree 3 and the trie for it.