Extracting unique rows of a bounded degree array using tries

by DOUGLAS COMER
Purdue University
West Lafayette, Indiana

SUMMARY
An array with integer entries between 0 and \( d-1 \) has bounded degree \( d \). This paper considers several algorithms for extracting the set of unique rows from a bounded degree array. For each algorithm considered, it gives the time, space, and I/O requirements, and an assessment of the types of applications for which the method is well-suited. It begins with four methods based on well known techniques and data structures, and goes on to propose a new algorithm which uses a form of digitial search tree known as a trie. It shows that the trie-based scheme has advantages over the other methods. Finally, it discusses some applications including computing a projection in a relational database system, and finding classes of isomorphic rows.

INTRODUCTION
A 2-dimensional array \( A \) is of bounded degree \( d \) if each element of \( A \) is an integer, \( a \), such that \( 0 \leq a < d \). The problem of extracting the unique rows of an array with bounded degree arises in many applications. The UNIX operating system [7] provides the command "uniq" which extracts all unique lines from a text file. In [5], Housel develops an algorithm for scheduling processes in a data restructuring program. One step of the algorithm finds all unique rows in a bounded degree matrix.

Finally, in a relational database system [2], each 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 extracting unique 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 removing a subset of columns from a given row is usually small, the difficulty in computing such a projection is essentially that of extracting the unique rows from an array.

In order to quantify the cost of various solutions to the problem of extracting unique 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 \( A' \). \( k \) may not be known a priori.

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 [6] 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 \( A' \) insert it.
2) Comparative Sort: Read \( A \) into memory and sort it using a comparative sort (like quicksort), placing the rows in lexicographical order.
3) Radix Sort: Proceed as in #2 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, requires only space for \( A' \), and reads \( A \) in row order. It requires only \( nm/b \) disk fetches, where \( b \) is the blocking factor. To compare two rows takes \( m \) comparisons, so if \( A' \) is kept ordered and a binary search is used, the running time is \( O(mk^2 + nm \log k) \), 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 \( (k^2 + nm \log_2 k) \). 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 \( O(n^2 + nm \log n) \). We shall see that other methods have much better running time.

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 \( O(n \log n) \) and the space required is \( mn \) plus locations for \( n \) pointers (to eliminate moving rows of \( A \)). Since the array has bounded degree, an immediate improvement in running time to \( O(mn) \) can be obtained by using Method 3, a radix sort, 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. However, processing in the input in column order, as is done in a radix sort, requires \( nm \) disk fetches for reasonable size array \( A \), while processing the input by row requires only \( nm/b \) disk accesses. The number of entries per block, \( b \), would normally be high, making radix sort unreasonable. 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 procedure. Second, since the number of distinct rows of \( A \) is not known \textit{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 \textit{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, requiring only \( nm/b \) disk accesses.

A TRIE-BASED METHOD

In this section a solution for Problem 1 which meets the four criteria listed above is presented. Like the other methods discussed, 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 [3] and Fredkin [4] for the storage of character data. Sussenguth [9] 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:** Let \( A' \) be a \( k \times m \) array of bounded degree \( d \) such that row \( i \neq \text{row } j, i \neq j \). A trie for \( A' \) is a tree with \( k \) leaves, each of which lies at depth \( \text{row } i+1 \) for which \( i+1 \).

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 the property. To follow an edge with label \( p \), one follows the \( p \text{th} \) pointer. 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 \( d \) 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 [6] provides a detailed algorithm for insertion and shows that it will require only \( O(m) \) steps. Thus, to build a trie for a \( k \times m \) array will require \( O(km) \) steps.

We now address the solution of Problem 1 using a trie. The method is straightforward: for each row, \( r \), of \( A \), if \( r \) is not in the trie, insert 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 a 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. The constant overhead is very small, making the method good in practice. 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 as fast as possible, and uses only \( O(km) \) space.
Extracting Unique Rows of a Bounded Degree Array Using Tries

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 [8]) 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 needs only \( nm/b \) disk accesses while reading \( A \). Therefore, the method meets the criteria listed above.

Notice that the trie-based method does well in the application of computing projections because the extraneous columns do not actually have to be removed before the trie is built. Instead, one can read a row of the file representing the relation, and pick out exactly those columns that should remain in the relation as the trie is searched.

OTHER APPLICATIONS

One way to look at the problem of extracting the unique rows of an array is to think of placing all equal rows in the same class. In some applications, one would like to group together all those rows which are isomorphic (equal up to a renaming of values), and extract one representative of each such class. For a binary matrix, a trie can be used to find classes of isomorphic rows using a minor variation of the method outlined above. As each row is inserted into the trie, the first element is examined. If it is a 0, the remaining elements in the row are inserted into the trie as usual. If the first element is a 1, the complement 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 greater than 2, the process of finding classes of isomorphic rows becomes more complicated. The trick is to scan the row once, changing entries to a canonical form before the row is inserted into the trie. As the row is scanned, each entry is examined to determine whether that value had been seen before. If it had, then it is mapped into the same integer as it was before. Otherwise, a fresh integer is assigned as its code, the entry is changed into the new integer, and the integer is stored for use with later entries in the row with the same value.

An unsophisticated algorithm for mapping a row of \( m \) entries into canonical form follows. In the algorithm, \( r \) is the row vector, \( c \) a counter, and \( v \) a vector of length \( d \).

\[
c := -1;
\]
\[
\text{for } i := 1 \text{ to } d \text{ do }
\]
\[
v[i] := -1;
\]
\[
\text{for } i := 1 \text{ to } m \text{ do begin }
\]
\[
\text{if } v[r[i]] = -1 \text{ then begin }
\]
\[
c := c + 1;
\]
\[
v[r[i]] := c
\]
\[
\text{end; (* if *) }
\]
\[
r[i] := v[r[i]]
\]
\[
\text{end; (* for *) }
\]

This implementation, which requires \( O(d) \) overhead per row as well as space for a vector of length \( d \), is quite practical for small \( d \). In fact, the cost of building a trie will usually dominate the small overhead incurred in changing the rows into canonical form.

A more sophisticated approach makes use of the "constant-time array initialization" mentioned in [1]. This result shows that the vector \( v \) could be initialized in constant time provided some extra space is used. The total space used is roughly \( 3d \) which makes the method useful for even moderately large \( d \).

CONCLUSIONS

We have shown that a trie index can be an efficient method for extracting unique 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 of the trie-based method is that of computing the projection of a relation in a relational database system. Other applications include extracting classes of isomorphic rows in a bounded degree array.

REFERENCES
