QUICKLEXSORT: AN EFFICIENT ALGORITHM FOR LEXICOGRAPHICALLY SORTING NESTED RESTRICTIONS OF A DATABASE

DAVID HAWS
IBM T.J. WATSON RESEARCH CENTER, 1101 KITCHAWAN ROAD, YORKTOWN HEIGHTS, NY 10598, USA

ABSTRACT. Lexicographical sorting is a fundamental problem with applications to contingency tables, databases, Bayesian networks, and more. A standard method to lexicographically sort general data is to iteratively use a stable sort—a sort which preserves existing orders. Here we present a new method of lexicographical sorting called QuickLexSort. Whereas a stable sort based lexicographical sorting algorithm operates from the least important to most important features, in contrast, QuickLexSort sorts from the most important to least important features, refining the sort as it goes. QuickLexSort first requires a one-time modest pre-processing step where each feature of the data set is sorted independently. When lexicographically sorting a database, QuickLexSort (including pre-processing) has comparable running time to using a stable sort based approach. For a data base with \( m \) rows and \( n \) columns, and a sorting algorithm running in time \( O(m \log(m)) \), a stable sort based lexicographical sort and QuickLexSort will both take time \( O(mn \log(m)) \). However in many applications one has the need to lexicographically sort nested data, e.g. all possible sub-matrices up to a certain cardinality of columns. In such cases we show QuickLexSort gives a performance improvement of a log factor of the database length (rows in matrix) over using a standard stable sort based approach. E.g. to sort all sub-matrices up to cardinality \( k \), QuickLexSort has running time \( O(mn^k) \) whereas a stable sort based lexicographical sort will take time \( O(m \log(m) n^k) \). After the pre-processing step that is run only once for the entire matrix, QuickLexSort has a running time linear in the number of nested sub-matrices to sort. We conclude with an application to Bayesian network scoring to detect epistasis using SNP marker data.

1. Introduction

Lexicographical ordering is a method to sort a list of elements where each element has multiple features, such as a vector, provided one has an order for each feature. Lexicographic ordering is also known as dictionary or alphabetical ordering. Put simply, the lexicographical ordering places the elements in a sequence such that: elements are ordered according to the first feature, any ties are broken by the second feature, any ties are broken by the third feature, etc. Lexicographical sorting is a fundamental problem with applications to contingency tables, Bayesian networks, databases[18, 23], and more. A contingency table lists the frequency of each element present in the data. For example, given a matrix, one can form a contingency table which for each unique row, counts the number of equal rows in the data. A naive approach would loop through each row, then again loop through the rows and count the number of equal rows. A better approach would be to sort all the rows of the matrix lexicographically, then loop through the matrix one last time forming the counts for the contingency table. Contingency tables formed from a matrix are used in the learning of Bayesian networks, as well as other applications.

Traditionally, one sorts the rows of a matrix lexicographically by iteratively applying a stable sort—a sorting algorithm which preserves the original order of elements that are equal. The rows of the matrix \( D \) are stable sorted by the least important feature, the next to least important feature, etc. If \( D \in \mathbb{R}^{m \times n} \) and the stable sorting algorithm runs in time \( T(m) \), then the time to sort the matrix is \( O(n T(m)) \). If the stable sort is a comparison base sort then \( T(m) \) is bounded below by \( \Omega(m \log(m)) \)[6]. In many applications, such as learning Bayesian networks, one not only wants to sort the rows of a data matrix \( D \), but also sort the rows of \( D \) restricted to a sequence of columns.

October 8, 2013
dhaws@us.ibm.com, dchaws@gmail.com
We present *QuickLexSort* which can efficiently sort a database (rows of a matrix) restricted to any sequence of features (columns of a matrix). Moreover, QuickLexSort is designed to quickly sort a nested set of restrictions by features. For example, one may wish to lexicographically sort all possible sub-matrices given by all non-empty sets of columns up to a specific cardinality. QuickLexSort first requires a modest pre-processing step where each feature of the data set is sorted independently. When lexicographically sorting the rows of a single matrix, QuickLexSort (including pre-processing) has comparable running time to using a stable sort based approach. However, when sorting a set of nested sets of features we show QuickLexSort gives a performance improvement of a multiple of a log factor of the database length (rows in matrix) over using a standard stable sort based approach. That is, after the pre-processing step, QuickLexSort has a running time linear in the number of nested sub-matrices to sort. The pre-processing step need only be computed once for the entire matrix and not for each sub-matrix.

The article is organized as follows. In section 2 we give background details on lexicographical sorting and the stable sorting approach. In section 3 we present QuickLexSort and prove the validity and space and running time. In section 4 we show how QuickLexSort can be used to efficiently sort nested restrictions of databases. In section 5 we present a small experiment verifying computationally the advantage of QuickLexSort over a stable sort based approach to lexicographically sorting a nested set of data base restrictions. In section 6 we briefly compare QuickLexSort with AD-trees. In section 7 we give details on applications of QuickLexSort to scoring and learning Bayesian networks. Moreover, we perform experiments showing the validity to using QuickLexSort and a Bayesian scoring approach to detecting epistasis in biological SNP data. Finally, in the Appendix in section 8 we present an augmented version of QuickLexSort which provides a more natural encoding of the lexicographical ordering.

## 2. Background

Sorting is the method of rearranging a sequence of items such that they are placed with respect to some order. Here we consider total or linear orderings. A set follows a total ordering, given by the symbol ‘≤’, if the following three conditions hold:

1. If $a \leq b$ and $b \leq a$ then $a = b$, (antisymmetry)
2. If $a \leq b$ and $b \leq c$ then $a \leq c$, (transitivity)
3. $a \leq b$ or $b \leq a$ (totality).

Here we primarily focus on comparison sorting, a class of sorting algorithms which only uses a binary comparison operation. That is, the only information used to sort is the given comparison operation.

It is known that the optimal running time of comparison sort is bounded below by $\Omega(n \log n)$ [6].

Some non-comparison sorting algorithms, such as bucket sort, may run in linear or near linear time, depending on the data. The exposition here will focus on real valued data and the standard ordering. However, the results extend naturally to any data with some type of comparison operation.

Let $D \in \mathbb{R}^{m \times n}$ be a real matrix where we index rows and columns by \{0, ..., $m - 1$\} and \{0, ..., $n - 1$\} respectively.

**Definition 1 (Lexicographic Order).** We say $v \in \mathbb{R}^n$ is less than $w \in \mathbb{R}^n$ lexicographically, denoted $v <_{\text{lex}} w$, if

1. $v_0 < w_0$ or
2. $v_0 = w_0, \ldots, v_k = w_k$ and $v_{k+1} < w_{k+1}$ for some $0 \leq k < n - 1$.

For example $[2, 1, 3, 0] <_{\text{lex}} [2, 1, 5, -1]$. If $n = 1$, lexicographical order is equivalent to the normal ordering. The task here is to sort the rows of sub-matrices of $D$ (obtained by taking subsets of the columns and all rows of $D$) using lexicographical order. Here we point out the distinction of a set and a sequence, both are groups of objects where in the former the ordering of the object is irrelevant and in the latter the ordering matters. That is $(1, 3, 2)$ and $(3, 2, 1)$ are distinguishable as sequences but not as sets. Typically we will use “\{\}” to denote sets and “()” to denote sequences.

By lexicographical sorting of $E \in \mathbb{R}^{m \times n}$ we mean the lexicographical sorting of the rows (vectors) of $E$ assuming the original ordering of columns of $E$. By lexicographical sorting of $E$ restricted to the set of columns $C$ we mean the lexicographical sorting of the sub-matrix of $E$ given by the set
of columns $C$ where the original order of the columns is preserved. By lexicographical sorting of $E$ restricted to the sequence of columns $S$ we mean the lexicographical sorting of the sub-matrix of $E$ given by the sequence of columns $S$ where the order of the columns is taken from $S$.

**Example 2.** Consider a matrix $E \in \mathbb{Z}^{5 \times 3}$, the lexicographical sorting ($E'')$ of $E$, the lexicographical sorting ($E''$) of the sub-matrix of $E$ given by the set of columns $\{1, 2\}$, and the lexicographical sorting ($E'''$) of the sub-matrix of $E$ given by the sequence of columns $\{2, 1\}$. In the following, we write the row and column indices of the original matrix $E$ on the left and top of the matrix respectively.

$$
E = \begin{pmatrix}
0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 \\
3 & 0 & 1 & 1 \\
4 & 0 & 0 & 1 \\
5 & 1 & 1 & 1 \\
\end{pmatrix}, \quad E' = \begin{pmatrix}
2 & 0 & 0 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
3 & 0 & 1 \\
4 & 0 & 1 \\
\end{pmatrix}, \quad E'' = \begin{pmatrix}
0 & 1 & 0 \\
4 & 0 & 1 \\
4 & 0 & 1 \\
1 & 1 & 0 \\
3 & 1 & 1 \\
\end{pmatrix}, \quad E''' = \begin{pmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
3 & 1 & 1 \\
\end{pmatrix}
$$

We note that all algorithms presented here do not modify the original data matrices, and simply represent the ordering via certain vectors which we define below. Note though, that in the examples we do reorder the rows to better illustrate certain concepts, although we are careful to preserve the original matrix row indices on the left.

**Definition 3 (Ranking Vector).** Let $D' \in \mathbb{R}^{m \times l}$ be a sub-matrix of $D \in \mathbb{R}^{m \times n}$. The unique ranking vector $L \in \mathbb{R}^m$ of some ordering $\tilde{O}$ of rows of $D'$ is a vector such that

1. $L \in \{0, \ldots, m-1\}^m$,
2. if row $i$ of $D'$ is equal to row $j$ in the ordering $\tilde{O}$, then $L_i = L_j$,
3. if row $i$ of $D'$ is less than row $j$ in the ordering $\tilde{O}$, then $L_i < L_j$,
4. $\sum_{i=0}^{m-1} L_i$ is minimal.

The last item guarantees the ranking vector $L$ is unique. We say ranking vector $L'$ is a refinement of ranking vector $L$ if $L_i \leq L_j$ implies $L'_i \leq L'_j$ for all $i, j$. Intuitively, the $m$ rows of $D$ are sorted lexicographically and thus form $l$ blocks, where $l \leq m$, every row in a block is lexicographically equal, and the $l$ blocks are in increasing lexicographic order. In this sense, $L_i$ is the block index in which row $i$ resides.

In what follows we consider a ranking vector sufficient information to describe an ordering. However, it may be desirable to have an alternative data structure to describe the ordering, such as an ordered list of row indices giving the smallest to largest row vectors. We describe such a case and the appropriate modifications to our algorithms in the Appendix in section 8. Our modified algorithm has the same time and space complexity.

**Example 4.**

| D       | D'     | D''    | D'''   |
|---------|--------|--------|--------|
| 0 1 2 3 4 | 0 1 2 3 4 | 0 3 4 | 0 3   |
| 0 1 1 1 2 1 | 0 0 0 3 0 | 2 3 0 | 2 3   |
| 1 1 1 1 2 0 | 1 0 1 1 0 | 1 1 0 | 1 1   |
| 2 0 0 0 3 0 | 4 1 0 1 1 1 | 4 1 1 | 4 1 1 |
| 3 1 1 1 2 0 | 7 1 1 0 2 1 | 5 1 1 | 5 1 1 |
| 4 1 0 1 1 1 | 5 1 1 1 1 1 | 9 1 1 | 9 1 1 |
| 5 1 1 1 1 1 | 9 1 1 1 1 1 | 1 1 2 | 0 1 2 |
| 6 1 1 1 1 3 0 | 1 1 1 1 2 0 | 3 1 2 | 0 1 2 |
| 7 1 1 1 0 2 1 | 3 1 1 1 2 0 | 1 2 1 | 3 1 2 |
| 8 1 0 1 1 0 | 0 1 1 1 2 1 | 7 1 2 | 7 1 2 |
| 9 1 1 1 1 1 | 6 1 1 1 3 0 | 6 1 3 | 6 1 3 |

A matrix $D$ with row indices $\{0, \ldots, 9\}$ and column indices $\{0, \ldots, 4\}$. The matrix $D'$ gives the rows of $D$ sorted lexicographically, $D''$ gives the rows of $D$ restricted to columns $\{0, 3, 4\}$ sorted lexicographically, and $D'''$ gives the rows of $D$ restricted to columns $\{0, 3\}$ sorted lexicographically. The ranking vectors of the lexicographic orderings shown in $D'$, $D''$, and $D'''$ are

$$
L' := [6, 5, 0, 5, 2, 4, 7, 3, 1, 4]^\top, \quad L'' := [4, 3, 0, 3, 2, 2, 5, 4, 1, 2]^\top, \quad L''' := [2, 2, 0, 2, 1, 1, 3, 2, 1, 1]^\top.
$$
Note the ranking vectors refer to the original row indices of the matrix \( D \). Note that \( L'' \) is a refinement of \( L''' \).

2.1. Stable Sort.

**Definition 5 (Stable Sort).** A sorting algorithm is *stable* if it maintains the relative order of items with equal value. That is, if \( a \) comes before \( b \) in the original input and \( a = b \), then a stable sorting algorithm orders \( a \) before \( b \).

**Example 6.** Suppose we performed a stable sorting of the rows of \( D \) where we use only the values in column 4 to perform the sort. We preserve the order of all rows which have the same value in column 4, and get \( D''' \) below.

\[
D = \begin{pmatrix}
0 & 1 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 1 & 1 & 2 & 0 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 1 & 2 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 1 & 3 & 0 \\
7 & 1 & 1 & 0 & 2 & 1 \\
8 & 1 & 0 & 1 & 1 & 0 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

\[
D''' = \begin{pmatrix}
0 & 1 & 1 & 2 & 3 & 4 \\
1 & 1 & 1 & 2 & 0 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 1 & 2 & 0 \\
6 & 1 & 1 & 1 & 3 & 0 \\
8 & 1 & 0 & 1 & 1 & 0 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

Although rows 1 and 2 have repeated values in column 4, a stable sorting algorithm places row 1 before row 2, preserving the original ordering.

A stable sorting algorithm can be used iteratively to perform lexicographical sorting. When a stable sorting algorithm is used to do lexicographical sorting we will refer to it as *StableLexSort*. See Algorithm 7 below.

**Algorithm 7 StableLexSort: Lexicographic sort using stable sort.**

**Require:** \( D \in \mathbb{R}^{m \times n}, (a_1, \ldots, a_p) \) where \( a_i \in \{0, \ldots, n-1\} \, \forall i \), \( S \) a stable sorting algorithm.

**Ensure:** \( D' \) a lexicographic sorting of rows of matrix \( D \).

1. Let \( D' := D \).
2. for \( j = p, \ldots, 1 \) do
3. Sort rows of \( D' \) using the stable sorting algorithm \( S \) and values in column \( a_j \).
4. end for
5. return \( D' \)

**Example 8.** We give an example of Algorithm 7 with input \( E \) and \((0, 1, 2, 3, 4)\). First stable sort the rows \( E \) by the values in column 4. Further stable sort the rows by values in column 3. Repeat.
stable sort of the rows by values in column 2, then 1, and finally 0.

\[
E := \begin{pmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 1 & 2 & 0 & 0 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 2 & 0 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 3 & 0 & 0 \\
7 & 1 & 0 & 1 & 1 & 0 \\
8 & 1 & 1 & 1 & 1 & 1 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix},
\begin{pmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 1 & 1 & 2 & 1 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 1 & 2 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 3 & 0 & 0 \\
7 & 1 & 0 & 1 & 1 & 0 \\
8 & 1 & 1 & 1 & 1 & 1 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix},
\begin{pmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 1 & 1 & 2 & 1 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 1 & 2 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 3 & 0 & 0 \\
7 & 1 & 0 & 1 & 1 & 0 \\
8 & 1 & 1 & 1 & 1 & 1 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix},
\begin{pmatrix}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 1 & 1 & 2 & 1 \\
1 & 1 & 1 & 1 & 2 & 1 \\
2 & 0 & 0 & 0 & 3 & 0 \\
3 & 1 & 1 & 1 & 2 & 0 \\
4 & 1 & 0 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 3 & 0 & 0 \\
7 & 1 & 0 & 1 & 1 & 0 \\
8 & 1 & 1 & 1 & 1 & 1 \\
9 & 1 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

**Proposition 9.** If \( D \in \mathbb{R}^{m \times n} \), and the running time of the stable sort algorithm \( S \) is \( T(m) \), then Algorithm 7 sorts in time \( O(nT(m)) \).

Thus if the stable sort algorithm \( S \) is a comparison sort, then the running time of Algorithm 7 is bounded below by \( \Omega(nm \log(m)) \). For example, if Merge sort \([16]\) was used, which has running time of \( O(m \log(m)) \), then the running time of StableLexSort on \( D \) would be \( O(nm \log(m)) \).

3. **QuickLexSort**

Here we present a new algorithm for lexicographical sorting called **QuickLexSort**. We will show that the running time of QuickLexSort is comparable to StableLexSort when sorting a single matrix. Moreover, we will demonstrate that QuickLexSort is considerably faster than StableLexSort when performing multiple lexicographic sorts of related sub-matrices.

The proposed algorithm QuickLexSort first requires each column of \( D \) to be independently sorted and stored. The results are stored in the matrix \( Q \in \mathbb{Z}_+^{m \times n} \) where the \( j \)th column \( Q_j \) of \( Q \) stores the row indices \( \{0, \ldots, m\} \) after sorting the \( j \)th column \( D_j \) of \( D \).
Example 10.

\[
D = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
1 & 1 & 1 & 2 & 1 \\
2 & 0 & 0 & 0 & 3 \\
3 & 1 & 1 & 1 & 2 \\
4 & 1 & 0 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 1 & 3 \\
7 & 1 & 0 & 1 & 1 \\
8 & 1 & 0 & 1 & 1 \\
9 & 1 & 1 & 1 & 1 \\
\end{bmatrix},
\quad
Q = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 \\
2 & 4 & 2 & 4 & 3 \\
4 & 2 & 7 & 5 & 8 \\
5 & 8 & 4 & 9 & 6 \\
6 & 5 & 6 & 8 & 2 \\
7 & 0 & 0 & 1 & 1 \\
9 & 7 & 1 & 7 & 4 \\
8 & 1 & 8 & 3 & 9 \\
3 & 3 & 3 & 2 & 0 \\
1 & 0 & 5 & 6 & 5 \\
\end{bmatrix}.
\]

A matrix \(D\) and the matrix \(Q\) storing the sort of the columns of \(D\) described above. E.g., reading down the 0th column of \(Q\), for column 0 of \(D\), the smallest entry is in row 2, followed by row 4, followed by row 5, etc.

The QuickLexSort algorithm sorts (conceptually) by iteratively appending columns to the current matrix and sorting, until the desired sequence of columns is reached. That is, Algorithm 11 refines the current sort with respect to the sequence of columns \((a_1, \ldots, a_j)\) to give a sort with respect to the sequence of columns \((a_1, \ldots, a_{j+1})\). In some sense this is opposite of StableLexSort. In StableLexSort one stable sorts from the least important column to the most important. In QuickLexSort one sorts from the most important column to the least important, refining the ranking vector as it goes.

Algorithm 11 (QuickLexSortRefine), is the core of the methods described here. Algorithm 11 takes as input the matrix \([D]\), the sorting of the columns of \([D]\) \([Q]\), the column to refine \([L]\) by \([i]\), and the current ranking vector \([L]\). It returns the refined ranking vector \([L']\). That is, if the input ranking vector \([L]\) represents the sorting of the rows of \([D]\) (restricted to some sequence of columns), the returned ranking vector \([L']\) represents the refined sorting where we consider appending the \(i\)th column of \([D]\). Again, if the input ranking vector \([L]\) represents the lexicographical sorting of a matrix

\[
D' = \begin{bmatrix}
j_0 & \cdots & j_p \\
0 & D_{0j_0} & \cdots & D_{0j_p} \\
1 & D_{1j_0} & \cdots & D_{1j_p} \\
\vdots & \vdots & \ddots & \vdots \\
n & D_{nj_0} & \cdots & D_{nj_p} \\
\end{bmatrix}
\]

the ranking vector \([L']\) output from Algorithm 11 represents the lexicographical sorting of the matrix

\[
D'' = \begin{bmatrix}
j_0 & \cdots & j_p & i \\
0 & D_{0j_0} & \cdots & D_{0j_p} & D_{0i} \\
1 & D_{1j_0} & \cdots & D_{1j_p} & D_{1i} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
n & D_{nj_0} & \cdots & D_{nj_p} & D_{ni} \\
\end{bmatrix}
\]

Intuitively, the task of Algorithm 11 is to 1) preserve the current ordering, i.e. if row \(j\) was lexicographically smaller than row \(k\), then this is true in the new order, 2) all previous rows that were lexicographically equal should be sorted given the newly appended column \(i\). The novelty of Algorithm 11 is that it performs the second item above in linear time using the pre-computed ordering of the newly appended column \(i\).

Example 12. Consider matrix \(D\) in Example 4. Suppose \(L\) is the ranking vector of the lexicographical ordering of the sub-matrix given by columns \(\{0, 3\}\) of \(D\) and we then perform Algorithm 11 with
Algorithm 11 QuickLexSortRefine

Require: $D \in \mathbb{R}^{m \times n}, Q \in \mathbb{Z}^{m \times n}, i \in \{0, \ldots, n-1\}, L \in \mathbb{Z}^m$.

Ensure: $L' \in \mathbb{Z}^m$.

1: $L' := 0 \in \mathbb{Z}^m$. # Records most recent value in $D$ w.r.t. ID.
2: IDval := 0 \in \mathbb{Z}^m.
3: IDvalInit := 0 \in \mathbb{Z}^m. # Records subID of $L$.
4: newCount := 0 \in \mathbb{Z}^m. # Records count of refinements of each input ID of $L$.
5: for $j = 0, \ldots, m-1$ do
6: \hspace{1em} if IDvalInit[$L[Q[j,i]]] == 0 then
7: \hspace{2em} IDvalInit[$L[Q[j,i]]] := 1.
8: \hspace{1em} IDval[$L[Q[j,i]]] := D[Q[j,i], i].
9: \hspace{1em} else
10: \hspace{2em} if IDval[$L[Q[j,i]]] != D[Q[j,i], i] then
11: \hspace{3em} IDval[$L[Q[j,i]]] := D[Q[j,i], i].
12: \hspace{3em} newCount[$L[Q[j,i]]] := newCount[$L[Q[j,i]]] + 1.
13: \hspace{2em} end if
14: \hspace{1em} end if
15: \hspace{1em} end if
16: subID[$Q[j,i]] := newCount[$L[Q[j,i]]].
17: end for
18: numNewID := 0 \in \mathbb{Z}^m.
19: numNewID[m-1] := \sum_{j=0}^{m-2} newCount[j].
20: for $j = m-2, \ldots, 1$ do
21: \hspace{1em} numNewID[j] := numNewID[j+1] - newCount[j].
22: end for
23: for $j = 0, \ldots, m-1$ do
24: \hspace{1em} $L'[j] := L[j] + numNewID[L[j]] + subID[j].$
25: end for
26: return $(L')$.

$i = 4$. Thus, part of the input would be

\[
\begin{pmatrix}
0 & 2 \\
1 & 2 \\
2 & 0 \\
3 & 2 \\
4 & 1 \\
5 & 1 \\
6 & 3 \\
7 & 2 \\
8 & 1 \\
9 & 1
\end{pmatrix}
\quad
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
2 & 0 \\
3 & 0 \\
4 & 1 \\
5 & 1 \\
6 & 0 \\
7 & 1 \\
8 & 0 \\
9 & 1
\end{pmatrix}
\quad
\begin{pmatrix}
0 & 3 \\
1 & 8 \\
2 & 6 \\
3 & 2 \\
4 & 1 \\
5 & 4 \\
6 & 7 \\
7 & 9 \\
8 & 0 \\
9 & 5
\end{pmatrix}
\]
The progression of the vectors IDval, subID, and newCount are shown from left to right as the for loop on line 6 goes from \( j = 0 \) to \( j = m - 1 \). Note, “·” signifies unassigned values.

\[
\begin{align*}
  j &= 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \\
  0 &= \begin{pmatrix}
  \cdot & \cdot & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 &= \begin{pmatrix}
  \cdot & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
  2 &= \begin{pmatrix}
  0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
  3 &= \begin{pmatrix}
  \cdot & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\

  \end{pmatrix}
  \end{pmatrix}
  \end{pmatrix}
  \end{pmatrix}
\]

Next we present Algorithm 13 (QuickLexSort) which, we will prove, lexicographically sorts a sub-matrix restricted to a sequence of columns of \( D \), requiring \( Q \) an initial sorting of the columns of \( D \).

\begin{algorithm}[h]
\caption{QuickLexSort}
\begin{algorithmic}[1]
\Require \( D \in \mathbb{R}^{m \times n} \), \( Q \in \mathbb{Z}^{m \times n} \) where the \( j \)th column of \( Q \) stores the sorting of the \( j \)th column of \( D \), \( L \) is the ranking vector of the lexicographical sorting of the sub-matrix of \( D \) determined the sequence of columns \( (a_1, \ldots, a_p) \), and \( i \in \{0, \ldots, n-1\} \).
\Ensure \( L' \in \mathbb{Z}^m \).
\State \( L' := 0 \in \mathbb{R}^n \).
\ForAll \( i \in \{a_1, \ldots, a_p\} \)
  \State \( L' := \text{QuickLexSortRefine}(D, Q, i, L') \).
\EndFor
\State \( \text{return } L' \).
\end{algorithmic}
\end{algorithm}

We now prove the validity and running times of Algorithm 11 and Algorithm 13.

**Lemma 14.** If \( D \in \mathbb{R}^{m \times n} \), \( Q \in \mathbb{R}^{m \times n} \) where the \( j \)th column of \( Q \) stores the sorting of the \( j \)th column of \( D \), \( L \) is the ranking vector of the lexicographical sorting of the sub-matrix of \( D \) determined the sequence of columns \( (a_1, \ldots, a_p) \), and \( i \in \{0, \ldots, n-1\} \), then Algorithm 11 returns the ranking vector \( L' \) of the lexicographical sorting of the sub-matrix of \( D \) determined by the sequence of columns \( (a_1, \ldots, a_p, i) \).

**Proof.** We need to prove

1. if \( L[j] < L[k] \) then \( L'[j] < L'[k] \), and
(2) when \( L[j] = L[k] \) we have \( D_{ji} < D_{ki} \) if and only if \( L'[j] < L'[k] \).

First, we note the for loop on line 6 visits the elements of \( D_i \) in increasing order by using the data structure \( Q \). Thus, for all ranks \( r \) in \( L \), all elements of \( D_i \) of rank \( r \) will be visited in increasing order. This is the crux of the validity of Algorithm 11 and is worthwhile to repeat. Given the current lexicographical order given by \( L \), every row of \( D \), restricted to the sequence of columns \((a_1, \ldots, a_p)\) has some rank \( r \). Because Algorithm 11 uses \( Q \), the algorithm will visit all the rows of current rank \( r \) in the order given by the new column \( i \) of interest.

We first claim that \( newCount[r] \) is equal to the number of unique elements of \( D_i \) of rank \( r \), with respect to \( L \). The data structure \( IDval[r] \) records the most recently observed value of \( D_i \) of rank \( r \). The data structure \( IDvalInit[r] \) simply denotes if nothing has been observed yet. Thus, when we observe an element of \( D_i \) of rank \( r \) that differs from \( IDval[r] \), we update \( IDval[r] \) (line 9 and line 12) and increase \( newCount[r] \) by one (line 13).

Second, we claim that \( subID \) restricted to all elements of rank \( r \) is a ranking vector over the elements of \( D_i \) restricted to elements of rank \( r \). More specifically, \( subID[j] \) is equal to the number of unique entries in \( D_i \) of rank \( L[j] \) strictly less than \( D_{ji} \). The variable \( subID[j] \) is initialized to be zero and is set to the current value of \( newCount[L[j]] \) (line 16). That is, \( subID[j] \) is set to the current number of unique elements of \( D_i \) of rank \( L[j] \).

The vector \( numNewID \) is simply a partial sum (offset by one index) of the vector \( newCount \). We can now prove the two important properties required to complete the proof. Suppose \( L[j] < L[k] \) and consider

\[
L'[k] - L'[j] = L[k] + numNewID[L[k]] + subID[k] - L[j] - numNewID[L[j]] - subID[j] \\
= L[k] + \sum_{l=0}^{L[k]-1} newCount[l] + subID[k] - L[j] - \sum_{l=0}^{L[j]-1} newCount[l] - subID[j]
\]

and note \( subID[j] \leq newCount[L[j]] \). Thus we have

\[
= L[k] + \sum_{l=L[j]}^{L[k]-1} newCount[l] + subID[k] - L[j] - \sum_{l=0}^{L[j]-1} newCount[l] - subID[j] \\
= L[k] + \sum_{l=L[j]+1}^{L[k]-1} newCount[l] + subID[k] - L[j] - subID[j] \\
\geq L[k] + \sum_{l=L[j]}^{L[k]-1} newCount[l] + subID[k] - L[j] \geq 0,
\]

and therefore \( L'[j] < L'[k] \).

Lastly, if \( L[j] = L[k] \) then considering the definition of \( L[j] \) and \( L[k] \) (line 24) we see the only variable is \( subID \). We have already shown that \( subID \) is a ranking vector of items of the same rank. Thus \( D_{ji} < D_{ki} \) if and only if \( subID[j] < subID[k] \) and the claim is proved.

\[\square\]

**Lemma 15.** If \( D \in \mathbb{R}^{m \times n} \), \( Q \in \mathbb{R}^{m \times n} \) where the \( j \)th column of \( Q \) stores the sorting of the \( j \)th column of \( D \), \((a_1, \ldots, a_p)\) where \( a_i \in \{0, \ldots, n-1\} \) \( \forall i \), then Algorithm 13 returns the ranking vector \( L' \) of the lexicographical sorting of the sub-matrix of \( D \) determined by the sequence of columns \((a_1, \ldots, a_p)\).

**Proof.** Since Algorithm 11 refines the ranking vector for each newly appended column, the result follows. \(\square\)

**Lemma 16.** Algorithm 11 runs in time \( O(m) \) and space \( O(m) \).

**Proof.** There are only three loops in Algorithm 11, each of them repeated \( m \) times. Each inner operation is constant time. The only space requirements are determined by column vectors of the \( m \times n \) input matrices and the vectors of length \( m \). \(\square\)
Lemma 17. Algorithm 13 runs in time $O(mp)$ and space $O(mn)$.

Proof. There are $p$ calls made to Algorithm 11 which by Lemma 16 imply the total running time is $O(mp)$. The only space requirements are determined by the $m \times n$ input matrices and the vectors of length $m$. □

Recall that both Algorithm 11 and Algorithm 13 require the columns of $D$ to be sorted and recorded in the input $Q$. Thus to lexicographically sort a matrix $D \in \mathbb{R}^{m \times n}$ using Algorithm 13 requires $O(nm \log(m) + nm) = O(nm \log(m))$, where we use an $O(m \log(m))$ comparison sort to find $Q$.

4. Sorting Sub-Matrices

Consider the problem of sorting all sub-matrices of $D$ given by every possible sequence of columns.

Problem 18 (Sort All Sub-Matrices Given By Column Sequences).
Let $D \in \mathbb{R}^{m \times n}$.

- For every sequence of columns $(a_1, \ldots, a_p)$ where, $1 \leq p \leq n$, $a_i \neq a_j \forall i, j$, $a_i \in \{0, \ldots, n-1\}$ \forall $i$:
  - Lexicographically sort the sub-matrix of $D$ determined by the sequence of columns $(a_1, \ldots, a_p)$.

Also consider the sub-problem of sorting all sub-matrices of $D$ given by every possible subset of columns.

Problem 19 (Sort All Sub-Matrices Given By Column Sets).
Let $D \in \mathbb{R}^{m \times n}$.

- For every non-empty subset of columns $\{a_1, \ldots, a_p\} \subseteq \{0, \ldots, n-1\}$:
  - Lexicographically sort the sub-matrix of $D$ determined by the set of columns $\{a_1, \ldots, a_p\}$.

In Problem 18 there are $\sum_{i=1}^{n} \frac{n!}{(n-i)!}$ non-empty sub-matrices to consider. In Problem 19 there are $2^n - 1$ non-empty sub-matrices to consider. Both StableLexSort (Algorithm 7) and QuickLexSort (Algorithm 13) can be used to solve Problem 18 and Problem 19. One simply enumerates the set of sub-matrices and applies either algorithm.

We now present how the core of the QuickLexSort Algorithm (Algorithm 11) lends itself ideally to Problem 18 and Problem 19. That is, we can use Algorithm 11 to efficiently sort all the nested sub-matrices. The new Algorithm 20 for Problem 18 enumerates all sequences of columns in a depth-first-search (DFS) manner. It then exploits the fact that Algorithm 11 will take a current ranking vector and refine it by considering appending an additional column. In this way we save the current ranking vector and refine it based on all possible ways to append a column to the current sub-matrix.

Algorithm 20 QuickLexSortAllSeq

Require: $D \in \mathbb{R}^{m \times n}$, $Q \in \mathbb{Z}^{m \times n}$, $(a_1, \ldots, a_p)$ where $a_i \in \{0, \ldots, n-1\}$ \forall $i$.
1: for $i \in \{0, \ldots, n\} \setminus \{a_1, \ldots, a_p\}$ do
2: $L' :=$QuickLexSortRefine($D, Q, i, L$).
3: Print $L'$.
4: QuickLexSortAllSeq($D, Q, (a_1, \ldots, a_p, i), L'$)
5: end for

Algorithm 20 is initially called with QuickLexSortAllSeq($D, Q, (), 0 \in \mathbb{R}^m$).

Lemma 21. Algorithm 20 has running time $O\left(\sum_{i=1}^{n} m \frac{n!}{(n-i)!}\right)$ and space requirements $O(mn)$.

Proof. Exactly $\frac{n!}{(n-i)!}$ calls are made to Algorithm 11, which itself has running time and space $O(nm)$. □
As it stands, StableLexSort could be used inside Algorithm 20 but would not achieve the same running time. If we replaced QuickLexSort (Algorithm 20) with StableLexSort (Algorithm 7) on line 2 of Algorithm 20 then the running time would increase to \( O \left( m \log(m) \sum_{i=1}^{m} \frac{n!}{(n-i)!} \right) \).

This highlights the distinct advantage of QuickLexSort: It is linear time to refine the lexicographical sorting when appending a column, provided the columns of the data matrix have been pre-sorted.

Naively one may think to use a stable sort algorithm and append the columns in the opposite order (since it has to work from least to most important columns), and proceed in a DFS manner to explore all possible sorting. However, the stable sort can not take advantage of the information contained in \( Q \) and would still need to do a comparison sort on each new column.

With minor alteration of Algorithm 20 we can handle Problem 19.

Algorithm 22

| QuickLexSortAllSubsets |
|------------------------|
| **Require:** \( D \in \mathbb{R}^{m \times n}, Q \in \mathbb{Z}^{m \times n}, \{a_1, \ldots, a_p\} \) where \( a_i \in \{0, \ldots, n-1\} \ \forall i. \) |
| **1:** for \( i \) such that \( n > i > \max(\{a_1, \ldots, a_p\}) \) do |
| **2:** \( L' := \text{QuickLexSortRefine}(D,Q,i,L). \) |
| **3:** Print \( L'. \) |
| **4:** QuickLexSortAllSubsets(\( D,Q,\{a_1, \ldots, a_p, i\},L' \)) |
| **5:** end for |

Algorithm 22 is initially called with QuickLexSortAllSubsets(\( D,Q,(.)\),0 \( \in \mathbb{R}^{m} \)).

**Lemma 23.** Algorithm 22 has running time \( O(m2^n) \) and space requirements \( O(mn) \).

**Proof.** Exactly \( 2^n - 1 \) calls are made to Algorithm 11, which itself has running time and space \( O(mn) \). □

Again, attempting to use StableLexSort on line 2 of Algorithm 22 would increase the running time to \( O(m \log(m)2^n) \). In both cases, this gain may seem modest given the dominating terms involving \( n \). However, we note that in many applications one may not in fact enumerate all sub-matrices but will instead enumerate all nested sub-matrices up to a certain cardinality. For example if one wishes to enumerate all sub-matrices with up to two columns then the running time of using QuickLexSort is \( O(mn^2) \) compared to \( O(m \log(m)n^2) \) for StableLexSort.

In general consider a set of nested sub-matrices indexed by their sequence of columns \( A \), and let \( |A| \) denote the size of \( A \). Nested in the sense that if \( A \in A \) then either \( A \) is a singleton or there exist \( B \in A \) such that \( A \) and \( B \) differ by one element. Then if one can efficiently (linear in \( |A| \)) enumerate the sub-matrices given by \( A \) then the running time to sort all \( |A| \) sub-matrices using QuickLexSort is \( O(m|A|) \). Extending the previous example, if one wishes to sort all sub-matrices with up to \( k \) columns, then the running time of QuickLexSort is \( O(mn^k) \).

5. Experiments

As a verification of the running times of Algorithm 11 and Algorithm 13 claimed in Lemma 14 and Lemma 15, we performed a short experiment using the Poker Hand data set from the University of California, Irvine’s Machine Learning Repository [1]. The data set consists of a matrix with 25,010 rows and 7 columns with discrete numerical values. Ten data sets were created for the experiments, consisting of the first 10\%, 20\%, ..., 100\% rows. For each of the ten data sets, QuickLexSort (using merge sort for the preliminary sorting of data columns) and StableLexSort (using merge sort) were run to sort all possible non-empty subsets of 7 columns. Note, the running times for QuickLexSort includes the pre-sorting step. Figure 1 shows the time to lexicographically sort using both methods. It is fairly easy to see the linear growth in running time of QuickLexSort compared to the linear times log factor running time of StableLexSort with respect to the number of rows.
QuickLexSort vs. StableLexSort

Figure 1. Running times of QuickLexSort and StableLexSort to sort all possible subsets of seven columns on ten truncations of the Poker Hand data set from the UCI Machine Learning data base.

6. Comparison to AD-trees

A popular method which specifically computes contingency tables (and sorts) is ADtrees [21]. Although retrieving a contingency table (or sorting) can be very fast – faster than QuickLexSort – the time and space requirements to compute and store the required data structures can be enormous. Assuming binary features (features only take two values), the cost to build an AD tree is bounded above by

$$\sum_{k=0}^{\left\lfloor \log_2(m) \right\rfloor} \frac{m}{2^k} \binom{n}{k},$$

(1)

where $D \in \mathbb{R}^{m \times n}$. If all possible combinations of the binary features appear in $D$, the space requirement would be $2^n$. Even with a reasonable number of rows the space requirement would be bound above by

$$\sum_{k=0}^{\left\lfloor \log_2(m) \right\rfloor} \binom{n}{k},$$

(2)

The time and space requirements can become practically prohibitive as $n$ and $m$ grow. For example, constructing and storing the ADtree for a dataset $D \in \mathbb{R}^{1000 \times 50000}$ would be infeasible. By contrast, QuickLexSort only requires linear space and time for each sort.
Bayesian networks (BN) are graphical models that have applications in a plethora of areas including machine learning, statistical inference, finance, biology, artificial intelligence, etc [17, 28]. Bayesian networks represent the conditional independences in some given data and are modeled through directed acyclic graphs (DAGs). In a naive sense, the task of learning the BN structure is to explore all possible DAGs and choose the DAG which best fits the data. Note that learning the BN structure is NP-hard [3, 5]. The fit of a proposed DAG to the data is evaluated by a scoring function such as Bayesian information criteria (BIC) or Bayesian Dirichlet equivalence (BDE) The BIC and BDE graph scoring functions evaluate a graph $G$ by looking at each node $i$ and its parents $Pa(i)$ (the nodes which have edges directed to $i$ in $G$). For every pair $(i, Pa(i))$, BIC and BDE compute local scores (a score depending only on $(i, Pa(i))$). In the end, the score of the proposed graph $G$ is a sum over all local scores $(i, Pa(i))$. At a low level, the BIC or BDE local score of $(i, Pa(i))$ simply requires two contingency tables: 1) a contingency table of the input data matrix restricted to columns indexed by $P(i)$, 2) a contingency table of the input data matrix restricted to columns indexed by $i \cup Pa(i)$.

In many current methods to learn a BN, the task is roughly broken into two steps. In step one, all local scores are precomputed. Often this is prohibitive and in practice only the local scores up to a certain cardinality are computed, or steps are taken to theoretically exclude certain local scores [8, 9]. In step two, the structure is learned by some intelligent method (Integer programming, Dynamic Programming, Heuristically, and more) [2, 4, 7–10, 14, 26, 27]. Research has focused mainly on the second step. However, the first step of local scoring merits exploration. For example the condition that parent sets are limited in cardinality can be quite artificial. But, if $m$ and $n$ become large, it may be prohibitive to compute all local scores using an approach such as StableLexSort. However, QuickLexSort is fast and requires small space. Moreover, approaches can be taken in which the local scores are done on-the-fly, in which case QuickLexSort can be of use.

One approach to learning BN is to perform a heuristic search of the solution space by iteratively changing the current graph structure[11, 19, 20, 22]. Again, in many cases it may be infeasible to store all contingency tables and it would be better to score each new graph. The proposed new graph can be chosen such that previous contingency tables can be updated efficiently by QuickLexSort. For example if the graphical moves are restricted to simply adding or removing a single edge of the current graph. As a new approach, we are currently developing a method which explores the solution space using characteristic imsets [13, 29–31] – a more natural encoding of unique probability models forming BNs – and QuickLexSort in order to efficiently move through the solution space.

A biological example of an application of Bayesian networks is the modeling of epistasis – the interaction of multiple genes to produce a phenotype. Using Bayesian networks (and related measures dependent on contingency tables) has proven useful in detecting epistasis [15, 25, 32]. In this case, one does not need to consider the full class of DAGs, and the problem reduces to simply scoring. Suppose we are given the genotypes for 1,000 individuals each with 50,000 single nucleotide polymorphisms (SNPs) and some phenotype (disease/no-disease). Then in this case we have a matrix $D \in \mathbb{R}^{1000 \times 50000}$. The task of detecting k-way epistasis using Bayesian networks reduces to computing contingency tables (sorting) all possible choices of $k$ subsets columns of $D$. Considering Equation 1 and Equation 2, it would be impractical to use ADtrees. For most $k > 2$, it would be infeasible to store all contingency tables for all $\binom{50,000}{k}$ SNP $k$-tuples. However, QuickLexSort requires linear time and space, to check each choice of $k$ columns of $D$.

As a preliminary experiment, QuickLexSort was used for Bayesian network scoring and the detection of epistasis on Maize genotype and phenotype data [24] used for the European CornFed program. The data used consisted of 261 inbred dent maize plant lines (rows) and 30,027 SNPs (columns). The phenotype used was male flowering time. The Bayesian Dirichlet equivalent (BDE) [12] score was used to detect up to two-way epistasis. In general, to compute the BDE score of any $k$ SNPs with respect to the phenotype, one needs two contingency tables: the contingency table given by the sub-matrix over those $k$ SNPs as well as the contingency table given by the sub-matrix over the $k$ SNPs with the additional column of phenotypes. QuickLexSort lends itself naturally to this process since we iterate through the sets of SNPs in depth-first-search manner.
Moreover, to compute the latter contingency table we simply append the phenotype column and do one call to Algorithm 11. For the experiment, the BDE score was computed for all singleton and pairs of SNPs. The QuickLexSort based approach took approximately 4.99 hours to perform \((30027^2) - 1 + 30027 = 450,825,379\) BDE scores on the \(261 \times 30,027\) data matrix, which required twice as many contingency table computations.

References

[1] K. Bache and M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.
[2] Mark Barlett and James Cussens. Advances in Bayesian network learning using integer programming.
[3] David Maxwell Chickering. Learning Bayesian networks is np-complete. In Learning from data, pages 121–130. Springer, 1996.
[4] David Maxwell Chickering. Optimal structure identification with greedy search. The Journal of Machine Learning Research, 3:507–554, 2003.
[5] David Maxwell Chickering, David Heckerman, and Christopher Meek. Large-sample learning of Bayesian networks is np-hard. The Journal of Machine Learning Research, 5:1287–1330, 2004.
[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. MIT Press, 2001.
[7] James Cussens. Bayesian network learning with cutting planes. arXiv preprint arXiv:1202.3713, 2012.
[8] Cassio P De Campos, Zhi Zeng, and Qiang Ji. Structure learning of Bayesian networks using constraints. In Proceedings of the 26th Annual International Conference on Machine Learning, pages 113–120. ACM, 2009.
[9] Cassio Polpo de Campos and Qiang Ji. Efficient structure learning of Bayesian networks using constraints. Journal of Machine Learning Research, 12(3):663–689, 2011.
[10] Luis M De Campos and Juan F Huete. A new approach for learning belief networks using independence criteria. International Journal of Approximate Reasoning, 24(1):11–37, 2000.
[11] Paolo Giudici and P.J. Green. Decomposable graphical gaussian model determination. Biometrika, 86(4):785–801, 1999.
[12] David Heckman, Dan Geiger, and David M Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. Machine learning, 20(3):197–243, 1995.
[13] Raymond Hemnecke, Silvia Lindner, and Milan Studeny. Characteristic insets for learning Bayesian network structure. International Journal of Approximate Reasoning, 53(9):1336–1349, 2012.
[14] Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using lp relaxations. In International Conference on Artificial Intelligence and Statistics, pages 358–365, 2010.
[15] Xia Jiang, Richard Neapolitan, M Michael Barnada, and Shyam Visweswaran. Learning genetic epistasis using Bayesian network scoring criteria. BMC bioinformatics, 12(1):89, 2011.
[16] Donald Knuth. The Art of Computer Programming, volume 3, pages 158–168. Addison-Wesley, 2nd edition, 1998.
[17] Daphne Koller and Nir Friedman. Probabilistic graphical models: principles and techniques. The MIT Press, 2009.
[18] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. Data & Knowledge Engineering, 69(1):3–28, 2010.
[19] David Madigan and Adrian E Raftery. Model selection and accounting for model uncertainty in graphical models using ocelli’s window. Journal of the American Statistical Association, 89 (428):1535–1546, 1994.
[20] David Madigan, Jeremy York, and Denis Allard. Bayesian graphical models for discrete data. International Statistical Review/Revue Internationale de Statistique, pages 215–232, 1995.
[21] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. Journal of Artificial Intelligence Research, 8:67–91, 1998.
and partitioning vectors are $L$

Example 26. Consider matrix $D$ and the sub-matrix $D''$ in Example 4. The ranking, ordering, and partitioning vectors are $L'':=[4,3,3,0,3,2,2,5,4,1,2]$, $O'':=[2,8,4,5,9,1,3,0,7,6]$, and $P'':=[0,1,2,5,7,9]$. Immediately, we can read off from $L''$ that there are six rank blocks, if we want to traverse the rows of the matrix $D''$ in the order we simply use $O''$, and $P''$ tells us how many rows of each rank are present.
We now give Algorithm 27 which takes the same input as Algorithm 11 as well as the ordering and partitioning vectors. It outputs the new ranking, ordering, and partitioning vector with respect to the lexicographical ordering one gets by appending column \( i \).

**Algorithm 27 QuickLexSortRefine**: Handles order vector.

**Require**: \( D \in \mathbb{R}^{m \times n}, Q \in \mathbb{R}^{m \times n}, i \in \{0, \ldots, n-1\}, L \in \mathbb{Z}^m, P \in \mathbb{Z}^m \).

**Ensure**: \( L' \in \mathbb{Z}^m, O' \in \mathbb{Z}^m, P' \in \mathbb{Z}^m \).

1: \( L' := 0 \in \mathbb{Z}^m \).
2: \( O' := 0 \in \mathbb{Z}^m \).
3: \( P' := P \in \mathbb{Z}^m \).
4: \( \text{IDval} := 0 \in \mathbb{Z}^m \). # Records most recent value in \( D \) w.r.t. ID.
5: \( \text{IDvalInit} := 0 \in \mathbb{Z}^m \).
6: \( \text{subID} := 0 \in \mathbb{Z}^m \). # Records subID of \( L \).
7: \( \text{newCount} := 0 \in \mathbb{Z}^m \). # Records count of refinements of each input ID of \( L \).
8: \textbf{for} \( j = 0, \ldots, m-1 \) \textbf{do}
9: \( O'[P'[L'[Q[j], i]]] := Q[j, i] \).
10: \( P'[L'[Q[j], i]] := P'[L'[Q[j], i]] + 1 \).
11: \textbf{if} \( \text{IDvalInit}[L'[Q[j], i]] \neq 0 \) \textbf{then}
12: \( \text{IDvalInit}[L'[Q[j], i]] := 1 \).
13: \( \text{IDval}[L'[Q[j], i]] := D[Q[j], i, i] \).
14: \textbf{else}
15: \( \text{IDval}[L'[Q[j], i]] ! = D[Q[j], i, i] \) \textbf{then}
16: \( \text{IDval}[L'[Q[j], i]] := D[Q[j], i, i] \).
17: \( \text{newCount}[L'[Q[j], i]] := \text{newCount}[L'[Q[j], i]] + 1 \).
18: \textbf{end if}
19: \textbf{end if}
20: \( \text{subID}[Q[j, i]] := \text{newCount}[L'[Q[j, i]]] \).
21: \textbf{end for}
22: \( \text{numNewID} := 0 \in \mathbb{Z}^m \).
23: \( \text{numNewID}[m-1] := \sum_{j=0}^{m-2} \text{newCount}[j] \).
24: \textbf{for} \( j = m-2, \ldots, 1 \) \textbf{do}
25: \( \text{numNewID}[j] := \text{numNewID}[j + 1] - \text{newCount}[j] \).
26: \textbf{end for}
27: \( \text{prevRank} := -1 \).
28: \textbf{for} \( j = 0, \ldots, m-1 \) \textbf{do}
29: \( L'[O'[j]] := L'[O'[j]] + \text{numNewID}[L'[O'[j]]] + \text{subID}[O'[j]] \).
30: \textbf{if} \( \text{prevRank} < > L'[O'[j]] \) \textbf{then}
31: \( \text{prevRank} := L'[O'[j]] \).
32: \( P'[L'[O'[j]]] := j \).
33: \textbf{end if}
34: \textbf{end for}
35: \textbf{return} \( (L', O', P') \).

**Lemma 28.** If \( D \in \mathbb{R}^{m \times n}, Q \in \mathbb{R}^{m \times n} \) where the \( j \)th column of \( Q \) stores the sorting of the \( j \)th column of \( D \), \( L \) is the ranking vector, \( O \) is the ordering vector, and \( P \) is the partitioning vector of the lexicographical sorting of the sub-matrix of \( D \) determined the sequence of columns \((a_1, \ldots, a_p)\), and \( i \in \{0, \ldots, n-1\} \), then Algorithm 27 returns the ranking vector \( L' \), the ordering vector \( O' \), and the partitioning vector \( P' \) of the lexicographical sorting of the sub-matrix of \( D \) determined by the sequence of columns \((a_1, \ldots, a_p, i)\).

**Proof.** The data structures \( \text{IDval}, \text{IDvalInit}, \text{newCount}, \) and \( \text{subID} \) all are initialized and updated the same in Algorithm 27 as they were in Algorithm 11. The new order vector \( O' \) is initialized to be zeros and the new partitioning vector \( P' \) is initialized to equal the input partitioning vector \( P \). Recall the \( P_i \) is the index into the ordering vector \( O \) where the rows of rank \( i \) begin. The goal of
lines 9 – 10 are to create the new ordering vector $O'$. To do this we must reorder all rows that have the same previous ranking according to $L$. Thus, the vector $P'$ is temporarily used to point to the next available index into $O'$ were the rows all have the same rank according to $L$. In line 9 we fill in the entries of $O'$ as we traverse the new column $i$ according to the pre-computed sorting given in $Q_i$. Specifically we look at the current rank of $Q[j, i]$ which is given in $L[Q[j, i]]$. Then $P'[L[Q[j, i]]]$ points to the next available position in $O'$ with rank equal to $L[Q[j, i]]$. Since we have filled this position $O'$ we increment $P'[L[Q[j, i]]]$ in line 10. In the end, $O'$ will be the ordering vector with respect to the new order given by the sequence of columns $(a_1, \ldots, a_p, a_i)$.

In line 27 we initialize the data structure $prevRank$ which will store the previously observed rank in the following for loop. In Algorithm 11 and lines 23 – 24 we filled in entries of $L'$ by traversing $j = 0, \ldots, m - 1$. We note that we could have traversed $j$ in any particular order. Thus, in Algorithm 27 we traverse $j$ in the order given by the new ordering vector $O'$. Therefore, by the arguments in the proof Lemma 14, $L'$ is the unique ranking vector given by the sequence of columns $(a_1, \ldots, a_p, a_i)$.

Lastly in lines 30 – 32 whenever we observe a row with a new rank, we set $P'$ to point to the appropriate index into $O'$. Therefore, $P'$ is the partitioning vector given by the sequence of columns $(a_1, \ldots, a_p, a_i)$.

Lemma 29. Algorithm 27 runs in time $O(m)$ and space $O(m)$.

Proof. There are only three loops in Algorithm 27, each of them repeated $m$ times. Each inner operation is constant time. The only space requirements are determined by column vectors of the $m \times n$ input matrices and the vectors of length $m$.

E-mail address: dhaws@us.ibm.com, dchaws@gmail.com