NISS

Preserving Confidentiality of High-dimensional Tabulated Data: Statistical and Computational Issues

Technical Report Number 130
November, 2002

Adrian Dobra, Alan F. Karr and Ashish P. Sanil

Preserving Confidentiality of High-dimensional Tabulated Data: Statistical and Computational Issues

Adrian Dobra[*], Alan F. Karr and Ashish P. Sanil
National Institute of Statistical Sciences
Research Triangle Park, NC 27709-4006, USA

November 13, 2002

Abstract

Dissemination of information derived from large contingency tables formed from confidential data is a major problem faced by statistical agencies. In this paper we present solutions to several computational and algorithmic issues that arise in the dissemination of cross-tabulations (marginal sub-tables) from a single underlying table. These include data structures that exploit sparsity and support efficient computation of marginals as well as algorithms such as iterative proportional fitting, and a generalized form of the shuttle algorithm that computes sharp bounds on (small, confidentiality threatening) cells in the full table from arbitrary sets of released marginals. We give examples illustrating the techniques.

Keywords: Branch and bound; Contingency tables; Disclosure limitation; Integer programming; Marginal bounds; Shuttle algorithm.

1 Introduction

Statistical agencies, such as the US Census Bureau and Statistics Canada, disseminate immense amounts of tabular information derived from confidential microdata. Protecting this confidentiality (and thereby the privacy of the data subjects) is mandated by law; doing so while releasing as much useful information as possible presents major challenges to statistical agencies.

For the past three years, the National Institute of Statistical Sciences (NISS) has been developing systems for disclosure-limited dissemination of tabular summaries of confidential microdata. Such summaries consist of marginal sub-tables of a large contingency table constructed by "summing out" one or more attributes. The full table of frequency counts of data records with the same (categorical) attribute values is assumed not to be releasable. More important, many sub-tables are also not releasable because either on their own or in conjunction with other sub-tables they provide too much information about the full table. Often, and in this paper, "too much information" means that small count (and therefore high risk [16]) cells can be bounded too accurately on the basis of the released sub-tables.

[*]Now at Duke University, Durham, NC 27708–0251.

Two classes of software systems have been developed [6, 9]. Table servers are "live," responding dynamically to incoming user queries for sub-tables of the full table, and assessing disclosure risk in light of previously answered queries. Table servers can be built at realistic scales, but defensible release rules and operating policies that the user community views as equitable are major impediments to their use in practice. Optimal tabular releases (OTRs), by contrast, are static releases of sets of sub-tables constructed by maximizing the amount of information released, as given by a measure of utility of that information, subject to a constraint on disclosure risk. Common underlying abstractions such as the query space and released and unreleasable sub-tables and frontiers are discussed in [6, 9].

In this paper we describe computational and algorithmic issues that must be confronted in order to build scalable implementations of table servers and OTRs. In the next section we introduce some basic notation that is needed to formally define the bounds problem. In §3 we describe the infrastructure—data structures and computational techniques—necessary to represent and operate on large (for example, 40–dimensional) contingency tables in the context of safe releases of sets of sub-tables. The common theme is sparsity: real, large tables are very sparse. §4 focuses on one method for computing sharp integer bounds based on any set of released sub-tables. The approach is based on the underlying hierarchical structure of the categorical data and includes a more general version of the shuttle algorithm [2], which we term the *generalized shuttle algorithm*. §5 contains a concluding discussion.

2 Terminology and Notation

Consider a k-way contingency table $\mathbf{n} = \{n(i)\}_{i\in\mathcal{I}}$ indexed by $\mathcal{I} = \mathcal{I}_1 \times \cdots \times \mathcal{I}_k$, where each $\mathcal{I}_j = \{1,\ldots,I_j\}$ is a finite set. With $A = \{i_1,\ldots,i_l\}$ an arbitrary subset of $K = \{1,\ldots,k\}$, let $\mathcal{I}_A = \mathcal{I}_{i_1} \times \cdots \times \mathcal{I}_{i_l}$. The A-marginal table of counts $\mathbf{n}_A = \{n_A(i_A)\}_{i_A\in\mathcal{I}_A}$ corresponding to A is given by

$$n_A(i_A) = \sum_{i_{K\setminus A}\in\mathcal{I}_{K\setminus A}} n(i_A, i_{K\setminus A}).$$

We refer to i (and i_A) as the *coordinate* of the table cell (and marginal table cell) it corresponds to. The corresponding $n(i)$ (and $n_A(i_A)$) are referred to as the cell *count*. The grand total of the table is $\mathbf{n}_\emptyset$. Also, let N_K be the number of cells in $\mathbf{n}$, and similarly, let N_A be the number of cells in a marginal subtable $\mathbf{n}_A$. Further, let N_K^+ and N_A^+ be the number of cells in $\mathbf{n}$ and $\mathbf{n}_A$ with positive counts (the *non-zero* cells).

Consider index sets $C_1,\ldots,C_p$. Suppose that for each C_j, we are given a table $\mathbf{n}^j$, and that these tables are consistent with each other: if $\mathbf{n}^{j_1}$ and $\mathbf{n}^{j_2}$ overlap ($C_{j_1} \cap C_{j_2} \neq \emptyset$), then $\mathbf{n}^{j_1}_{C_{j_1}\cap C_{j_2}} = \mathbf{n}^{j_2}_{C_{j_1}\cap C_{j_2}}$, while if $\mathbf{n}^{j_1}$ and $\mathbf{n}^{j_2}$ do not overlap, then $n^{j_1}_\emptyset = n^{j_2}_\emptyset$. Denote by $\mathbf{T}(\mathbf{n}^1,\ldots,\mathbf{n}^p)$ the set of all tables—termed *feasible*—whose $C_1,\ldots,C_p$–marginals are equal to $\mathbf{n}^1,\ldots,\mathbf{n}^p$.

Note that $\mathbf{T}(\mathbf{n}^1,\ldots,\mathbf{n}^p)$ might be empty even if $\mathbf{n}^1,\ldots,\mathbf{n}^p$ are consistent with each other. "Good" algorithms (for example, to compute bounds) should first check whether there exists at least one integer table consistent with the constraints, and generate upper and lower bounds only if a feasible table exists.

Let

$$U(i;\mathbf{T}(\mathbf{n}^1,\ldots,\mathbf{n}^p)) = \max\left\{\mathbf{x}(i) : \mathbf{x} \in \mathbf{T}(\mathbf{n}^1,\ldots,\mathbf{n}^p)\right\}$$

and

$$L(i; \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)) = \min \left\{ \mathbf{x}(i) : \mathbf{x} \in \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p) \right\}.$$

be the maximum and minimum values of cell i over all tables in $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$

We denote by $\mathcal{P}(\mathcal{I}_r)$ the set of all partitions of $\mathcal{I}_r$ and by $\mathcal{RD}$ the set of marginal tables that can be obtained by aggregating $\mathbf{n}$ not only across variables, but also across categories within variables. A table $\mathbf{n}' \in \mathcal{RD}$ is uniquely determined from $\mathbf{n}$ by choosing $\mathcal{I}'_1 \in \mathcal{P}(\mathcal{I}_1), \ldots, \mathcal{I}'_k \in \mathcal{P}(\mathcal{I}_k)$:

$$\mathbf{n}' = \left\{ t_{J_1 \ldots J_k} : (J_1, \ldots, J_k) \in \mathcal{I}'_1 \times \cdots \times \mathcal{I}'_k \right\},$$

where

$$t_{J_1 \ldots J_k} = \sum_{i_1 \in J_1} \cdots \sum_{i_k \in J_k} n_K(i_1, \ldots, i_k).$$

Let $\mathbf{T}$ denote the set of cells of all tables in $\mathcal{RD}$. If the set of cell entries in $\mathbf{n}$ that define a "super-cell" $t_1 \in \mathbf{T}$ is included in the set of cells defining another "super-cell" $t_2 \in \mathbf{T}$, we write $t_1 \prec t_2$. Define a partial ordering "$\prec$" on the cells in $\mathbf{T}$ by

$$t_{J_1^1 \ldots J_k^1} \prec t_{J_1^2 \ldots J_k^2} \Leftrightarrow J_1^1 \subseteq J_1^2, \ldots, J_k^1 \subseteq J_k^2.$$

With this partial ordering, $(\mathbf{T}, \prec)$ has a maximal element, namely the grand total of $\mathbf{n}$, and several minimal elements—the cell entries in the initial table $\mathbf{n}$.

If $t_1 = t_{J_1^1 \ldots J_k^1}$ and $t_2 = t_{J_1^2 \ldots J_k^2}$ are such that $t_1 \prec t_2$ with $J_r^1 = J_r^2$, for $r = 1, \ldots, r_0 - 1, r_0 + 1, \ldots, k$ and $J_{r_0}^1 \neq J_{r_0}^2$, we define the *complement* of the cell t_1 with respect to t_2 to be the cell $t_3 = t_{J_1^3 \ldots J_k^3}$, where for $r = 1, \ldots, k$,

$$J_r^3 = \begin{cases} J_r^1, & \text{if } r \neq r_0, \\ J_r^2 \setminus J_r^1, & \text{if } r = r_0. \end{cases}$$

In this case we write $t_1 \oplus t_3 = t_2$. The operator "$\oplus$" is equivalent to joining two blocks of cells in $\mathbf{T}$ to form a third block. The blocks to be joined have to be composed from the same categories in $(k-1)$ dimensions and cannot share any categories in the remaining dimension.

3 Data Structures and Algorithms for Large, Sparse Tables

The simplest computer representation of a contingency table is as a multi-dimensional array of non-negative integers. (Even though it is unwieldy to use multi-dimensional arrays in most programming languages, this structure can readily be implemented as a one–dimensional array by representing the multi-dimensional coordinates as mixed-radix integers [10].) However, the number of cells in a table increases exponentially with the table dimension, and so even moderate sized tables (10–20 dimensional) can have unmanageably large numbers of cells. For instance, a 14-dimensional table derived from the Current Public Survey (CPS) data from 1993, which we have used as a test case, has 4.5 billion cells! The time and space requirements of processing such tables using a naive representation are clearly prohibitive. Fortunately, tables of real data are extremely sparse—the 14-dimensional table mentioned above with 4.5 billion cells has merely 76,000 cells with non-zero counts.

In this section we describe useful data structures and algorithms that exploit the extreme sparsity of the tables to enable us to perform certain operations on the tables. We first present a hash-table based structure for tables and outline algorithms for building the table from microdata and for generating marginal subtables (§3.1) followed by an illustration (§3.2) of how the basic structures and minor extensions can be used to perform a variant of Iterative Proportional Fitting (IPF) [1]. We note that AD-Trees [13] and certain Online Analytical Processing (OLAP) technologies [8] provide complementary methods for handling tables.

3.1 Building Tables and Marginal Tables

It is clear that any viable representation for large, sparse tables ($N_K \gg N_K^+$) must exploit the table's sparseness. The most natural strategy is to store the location (coordinate) and content (count) of only the non-zero cells. This leads to an enormous saving of space, since we only have to store N_K^+ items instead of N_K items. However, the naive storage of a list of (coordinate, count) pairs suffers from the disadvantage that retrieving the count of an arbitrary cell entails a search through the list of cells. Hence, this query, which could be processed in $O(1)$ time with the multidimensional array representation, now takes $O(N_K^+)$ time (or $O(\log N_K^+)$ if the list is sorted). Fortunately, we can achieve the space-saving of a list with the fast access time of an array by using a *hash table*.

Hash tables are well-known data structures that support efficient storage and retrieval for sets of (*key*, *value*) pairs—(*coordinate*, *count*) pairs in our case. In a hash table, the data are stored in an array. The essential component is a *hash function* that maps every key (coordinate) to a location in the array and allows us almost instantly to access data corresponding to a given key. There are many design issues pertaining to construction of a good hash function, selection of a good array size, and several implementation tricks that could be employed. We do not cover the details here since this is a widely-used and well-studied data structure [3, 10], and has implementations incorporated as standards in computer languages like Java (part of the "Java collections framework" [15]) and C++ (part of the standard library [14]).

Three essential aspects are relevant to us. First, tables and marginal subtables are stored in hash tables where the *key* is a K-variate generalized coordinate, $(x_i, \ldots, x_K)$ with $x_j \in \mathcal{I}_j \cup \{0\}$ and the *value* is the corresponding cell count. (We call it a generalized coordinate since it can represent cells in both the full table and any marginal subtable, with the key for a cell in a marginal table $\mathbf{n}_A$ being a K-variate coordinate with $x_j = 0$ if $j \notin A$). Second, the hash function is defined over the set of all generalized coordinates.

Third, the following operations are supported efficiently:

1. ADDCOUNT(`Tab`, `coord`): Increments by one the count for the cell with coordinate `coord` in table or marginal table `Tab`.

2. GETCOUNT(`Tab`, `coord`): Retreives the cell count for the cell with coordinate `coord` in table or marginal table `Tab`.

3. GETMARGINALCOORD(`M`, `coord`): Returns the generalized coordinate in the marginal subtable `M` corresponding to `coord` in the full table (if `M` $\equiv \mathbf{n}_A$, then this can be computed by setting $x_j = 0$ for $j \notin A$).

We assume the the full table fits into the computer's main memory. This is a reasonable assumption for modern computers in cases such as survey data, where the number of subjects, $\mathbf{n}_\emptyset$, is seldom more than an order of 10^5 and since $N_K^+ < \mathbf{n}_\emptyset$ and, usually, $N_K^+ \ll \mathbf{n}_\emptyset$. It is clear that the hash table representation of the contingency table can be easily constructed by reading through the microdata records sequentially and incrementally updating the table using ADDCOUNT.

Given the hash table representation, computation of an arbitrary marginal sub-table is easy:

Pseudocode 3.1 COMPUTEMARGINAL(Table,Marginal)

```
for each coord in Table do
    marginalCoord = GETMARGINALCOORD(Marginal,coord)
    ADDCOUNT(Marginal, marginalCoord)
end for
```

3.2 Example: IPF

We illustrate here how the hash table representation can be used to develop efficient algorithms to process large, sparse tables, using Iterative Proportional Fitting (IPF) as an example. Intuitively, IPF finds the "best" reconstruction of the full table consistent with a set of marginal tables of the original full table. More precisely, IPF calculates the maximum likelihood estimate for a log-linear model defined on variables on the full table whose minimal sufficient statistics are the given by the set of marginal tables [1].

The algorithm involves iteratively refining estimated cell values for the table. Each iteration consists of stepping through the list of marginal tables and scaling the current cell estimates to make the current table estimate consistent with the marginal table. (To illustrate, for a two-way table, entries are alternately re-scaled row-wise to make the row sums "correct" and then column-wise to make the column sums "correct.") Specifically, consider cell i with $\hat{n}(i)$ as its current estimate. Let $n(+)$ be the cell count in the marginal table currently under consideration to which i contributes to, and let $\hat{n}(+)$ be the sum of all current estimates of cells like i that contribute to the marginal total. Then $\hat{n}(i)$ is adjusted as $\hat{n}(i) \leftarrow [n(+)/\hat{n}(+)]\hat{n}(i)$. After adjusting all cells for a given marginal, the current table estimate will be consistent with the marginal under consideration [1].

The IPF variant we present here only considers non-zero cells—this is equivalent to the original IPF with structural or known zeroes. Implementing it involves a minor extension the basic data structure. The (*key*, *value*) pair in the hash table now consists of *key* = coordinate as before and *value* = (count,fit) = $(n, \hat{n})$, where $n \equiv n(i)$, $\hat{n} \equiv \hat{n}(i)$ for the table, and $n \equiv n(+)$, $\hat{n} \equiv \hat{n}(+)$ for the marginal table. We define procedures ADDFIT(Tab, coord) and GETFIT analogous to ADDCOUNT and GETCOUNT in §3.1. We also define a procedure UPDATEFIT(Tab, coord, fit) that sets the "fit" component to the value of fit.

The IPF employs two subroutines MAKEMARGINALSUMS(Table,Marginal) (Pseudocode 3.2) and ADJUSTTABLE(Table,Marginal) (Pseudocode 3.3) that compute the $\hat{n}(+)$s and the $\hat{n}(i)$s respectively.

The IPF algorithm can be implemented as outlined in Pseudocode 3.4. This version trades off time efficiency for memory space savings by recomputing marginal tables every time they are required. Since this strategy maintains only two tables in memory at any given time, it allows us to handle arbitrarily large

Pseudocode 3.2 MakeMarginalSums(Table,Marginal)

```
for each coord in Table do
  marginalCoord = GetMarginalCoord(MarginalTable,coord)
  AddFit(Marginal, marginalCoord)
end for
```

Pseudocode 3.3 AdjustTable(Table,Marginal)

```
for each coord in Table do
  marginalCoord = GetMarginalCoord(Marginal,coord)
  marginalFit = GetFit(Marginal,marginalCoord)
  tableFit = GetFit(Table,coord)
  marginalCount = GetCount(Marginal,marginalCoord)
  tableFit ← tableFit * (marginalCount/marginalFit)
  UpdateFit(Table,coord)
end for
```

sets of marginals. It is also possible to precompute the marginal tables and avoid the ComputeMarginal in the inner loop if both the set of marginal tables and the full table can fit into main memory.

Pseudocode 3.4 IPF(Table,ListOfMarginalNames)

```
repeat
  for each Marginal in ListOfMarginalNames do
    ComputeMarginal(Table,Marginal)
    MakeMarginalSums(Table,Marginal)
    AdjustTable(Table,Marginal)
  end for
until Estimates converge
```

4 The Generalized Shuttle Algorithm

The fundamental idea behind the "shuttle" algorithm is that the upper and lower bounds for the cells in a table $\mathbf{T}$, based on knowledge of an arbitrary set of marginal sub-tables, are interlinked. Our method builds on [2], which treats the problem of a 3-way table given the three 2-way marginals, and sequentially improves the bounds for cells of interest until no further adjustment can be made.

Denote by $L(t)$ and $U(t)$ the current lower and upper bounds for the "super-cell" $t \in \mathbf{T}$. (See §2) for details of the notation.) Let $L(\mathbf{T}) = \{L(t) : t \in \mathbf{T}\}$ and $U(\mathbf{T}) = \{U(t) : t \in \mathbf{T}\}$. If $\mathcal{N} \subset \mathbf{T}$ is the set of cells in $\mathbf{x} \in \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$, then $L(\mathcal{N})$ and $U(\mathcal{N})$ are the bounds arrays to be determined. Every $t \in \mathbf{T}$ has a value $V(t)$ assigned to it. If t corresponds to an entry in a fixed marginal, we "know" the value $V(t)$ of that entry, so we set both current bounds of t to $V(t)$.

Let $\mathbf{T}_0$ be the set of cells in $\mathbf{T}$ for which the lower bound is currently equal to the upper bound. If we fix all the cells in $\mathcal{N}$ at a certain value, then all the remaining cells in $\mathbf{T}$ will also be fixed: $\mathcal{N} \subset \mathbf{T}_0$ implies $\mathbf{T} = \mathbf{T}_0$. Consider $\mathbf{M} \subset \mathbf{T}$ to be the set of cells in the fixed marginals $\mathbf{n}^1, \ldots, \mathbf{n}^p$. When the iterative procedure described below starts, $\mathbf{T}_0$ will contain only the cells in the fixed marginals, i.e., $\mathbf{T}_0 = \mathbf{M}$. For the remaining cells in $\mathbf{T}$, we set $L(t) = 0$ and $U(t) = \mathbf{n}_\emptyset$. Denote by $L_0(\mathbf{T})$ and $U_0(\mathbf{T})$ this initial set of upper and lower bounds induced by $\mathbf{n}^1, \ldots, \mathbf{n}^p$.

As the algorithm progresses, the current bounds $L(\mathbf{T})$ and $U(\mathbf{T})$ are improved (lower bounds increase and upper bounds decrease), and more and more cells are added to $\mathbf{T}_0$. When the bounds associated with t become equal, t is added to $\mathbf{T}_0$, and is assigned value $V(t) = L(t) = V(t)$. We state the bounds problem in a new equivalent form: *Find sharp integer bounds for the cells in* $\mathbf{T}$ *if the values of some cells* $\mathbf{T}_0 \subset \mathbf{T}$ *are fixed.*

Let $\mathcal{Q} = \mathcal{Q}(\mathbf{T})$ denote the triplets of cells $\mathcal{Q}(\mathbf{T}) = \{(t_1, t_2, t_3) \in \mathbf{T} \times \mathbf{T} \times \mathbf{T} : t_1 \oplus t_3 = t_2\}$ that represent the cell dependencies to be satisfied. Let $S[L_0(\mathbf{T}), U_0(\mathbf{T})]$ be the set of integer tables consistent with $L_0(\mathbf{T})$ and $U_0(\mathbf{T})$. It is easy to see that

$$\{V(\mathcal{N}) : V(\mathbf{T}) \in S[L_0(\mathbf{T}), U_0(\mathbf{T})]\} = \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p).$$

To improve the current bounds, we go sequentially through all dependencies in $\mathcal{Q}$ and update upper and lower bounds in the following way. Consider a triplet $(t_1, t_2, t_3) \in \mathcal{Q}$ with $t_1 \prec t_2$ and $t_3 \prec t_2$. If $t_1, t_2, t_3 \in \mathbf{T}_0$, we check whether we came across an inconsistency. The procedure stops if $V(t_1) + V(t_3) \neq V(t_2)$. Assume that $t_1, t_3 \in \mathbf{T}_0$, and $t_2 \notin \mathbf{T}_0$. Then t_2 can only take one value, namely $V(t_1) + V(t_3)$. If $V(t_1) + V(t_3) \notin [L(t_2), U(t_2)]$, we have encountered an inconsistency and exit the procedure. Otherwise we set $V(t_2) = L(t_2) = U(t_2) = V(t_1) + V(t_3)$, and include t_2 in the set $\mathbf{T}_0$ of cells having a fixed value. Similarly, if $t_1, t_2 \in \mathbf{T}_0$ and $t_3 \notin \mathbf{T}_0$, t_3 can only be equal to $V(t_2) - V(t_1)$. If $V(t_2) - V(t_1) \notin [L(t_3), U(t_3)]$, we again discovered an inconsistency. Otherwise, we set $V(t_3) = L(t_3) = U(t_3) = V(t_2) - V(t_1)$ and $\mathbf{T}_0 = \mathbf{T}_0 \cup \{t_3\}$. In the case that $t_2, t_3 \in \mathbf{T}_0$ and $t_1 \notin \mathbf{T}_0$ is handled analogously.

Now we examine the situation when at least two of the cells t_1, t_2, t_3 do not have a fixed value. For each of the three cells not having a fixed value, we update its upper and lower bounds so that the new bounds satisfy the dependency $t_1 \oplus t_3 = t_2$. Suppose that $t_1 \notin \mathbf{T}_0$. If $U(t_2) - L(t_3) < L(t_1)$ or if $L(t_2) - U(t_3) > U(t_1)$, an inconsistency is detected and the procedure stops. Otherwise, the updated bounds for t_1 will be $U(t_1) = \min\{U(t_1), U(t_2) - L(t_3)\}$ and $L(t_1) = \max\{L(t_1), L(t_2) - U(t_3)\}$. If $t_3 \notin \mathbf{T}_0$, we update $L(t_3)$ and $U(t_3)$ in the same way. Finally, assume that $t_2 \notin \mathbf{T}_0$. If $U(t_1) + U(t_3) < L(t_2)$ or if $L(t_1) + L(t_3) > U(t_2)$, we stop the algorithm. Otherwise, we set $U(t_2) = \min\{U(t_2), U(t_1) + U(t_3)\}$ and $L(t_2) = \max\{L(t_2), L(t_1) + L(t_3)\}$. After updating the bounds of $t \in \mathbf{T}$, we check whether the new upper bound is equal to the new lower bound. If so, we add t to $\mathbf{T}_0$ and set $V(t) = L(t) = U(t)$.

We continue cycling through dependencies in $\mathcal{Q}$ until the upper bounds no longer decrease, the lower bounds no longer increase and no new cells are added to $\mathbf{T}_0$. The procedure terminates in a finite number of steps: either an inconsistency is detected or bounds cannot be improved.

If an inconsistency is detected, $S[L_0(\mathbf{T}), U_0(\mathbf{T})]$ is empty and no bounds are generated. However, $S[L_0(\mathbf{T}), U_0(\mathbf{T})]$ could still be empty even if the shuttle procedure did not come across any inconsistencies and has converged to bounds $L_s(\mathbf{T})$ and $U_s(\mathbf{T})$. These two bounds arrays define the same feasible set of tables as the arrays $L_0(\mathbf{T})$ and $U_0(\mathbf{T})$ we started with, namely $S[L_s(\mathbf{T}), U_s(\mathbf{T})] = S[L_0(\mathbf{T}), U_0(\mathbf{T})]$. The fact that, for any $t \in \mathbf{T}$, we have $L_0(t) \leq L_s(t) \leq U_s(t) \leq U_0(t)$, could make one think that $S[L_s(\mathbf{T}), U_s(\mathbf{T})]$ might be strictly included in $S[L_0(\mathbf{T}), U_0(\mathbf{T})]$. Nevertheless, the dependencies in $\mathcal{Q}(\mathbf{T})$ imply the equality of the two sets of feasible integer tables [4].

4.1 Convergence Properties

The bounds produced by the generalized shuttle algorithm (provided it did not stop due to an inconsistency) are *valid* in the sense that the sharp bounds for each cell $t_0 \in \mathbf{T}$ lie inside the interval defined by the shuttle bounds for t_0. However, there are two cases when the shuttle bounds are sharp: dichotomous k-dimensional tables with all $(k-1)$-dimensional marginals fixed, and when fixed the marginals are the minimal sufficient statistics of a decomposable log-linear model. In both instances, explicit formulas for the bounds exist, and employing the generalized shuttle algorithm is equivalent to using these formulas. Computing the bounds directly is more efficient, but it is insightful to examine how the algorithm works in these two cases.

4.2 Dichotomous k-way Tables with Fixed $(k-1)$-way Marginals

Consider a k-way table $\mathbf{n} = \{n(i)\}_{i \in \mathcal{I}}$ for which $\mathcal{I}_1 = \cdots = \mathcal{I}_k = \{1, 2\}$. Collapsing $\mathbf{n}$ across categories is equivalent to collapsing $\mathbf{n}$ across variables, thus the set $\mathbf{T}$ associated with the dichotomous table $\mathbf{n}$ is the set of cells in every marginal of $\mathbf{n}$. Assume that $(k-1)$-dimensional marginals of $\mathbf{n}$ are fixed. This implies that every lower-dimensional marginal of $\mathbf{n}$ will also be known. The only cells in $\mathbf{T}$ that are unknown are those in the original table.

The $(k-1)$-dimensional marginals of $\mathbf{n}$ are the minimal sufficient statistics of the log-linear model of no $(k-1)$-order interaction. This log-linear model has only one degree of freedom because $\mathbf{n}$ is dichotomous [7]. Consequently, we can uniquely express the count in any cell as a function of one single fixed cell alone—only one more quantity is needed in order to determine the entries for the full table.

Let n^* be the unknown count in the $(1, \ldots, 1)$ cell. In Proposition 1 [4] we give an explicit formula for computing the count in an arbitrary cell based on n^* and on the set of fixed marginals.

Proposition 1. *Consider an index $i^0 \in \mathcal{I}$. Let $\{q_1, \ldots, q_l\} \subset K$ be such that, for $r \in K$,*

$$i^0_r = \begin{cases} 1, & \text{if} \quad r \in K \setminus \{q_1, \ldots, q_l\}, \\ 2, & \text{if} \quad r \in \{q_1, \ldots, q_l\}. \end{cases} \tag{1}$$

For $s = 1, \ldots, l$, let $C_s = K \setminus \{q_s\}$. Then

$$n(i^0) = (-1)^l \cdot n^* - \sum_{s=0}^{l-1} (-1)^{l+s} \cdot n_{C_{(l-s)}}(1, \ldots, 1, i^0_{q_{(l-s)}+1}, \ldots, i^0_k). \tag{2}$$

The upper and lower bounds can therefore be obtained by imposing the non-negativity constraints $n(i^0) \geq 0$, $i^0 \in \mathcal{I}$, in these relations. For example, the sharp lower bound for the cell $(1, \ldots, 1)$ is

$$\max \left\{ \sum_{s=0}^{l-1} (-1)^s \cdot n_{C_{(l-s)}}(1, \ldots, 1, i^0_{q_{(l-s)}+1}, \ldots, i^0_k) : l \text{ even} \right\}, \tag{3}$$

where i^0 is as in (1), and the corresponding upper bound is given by

$$\min \left\{ \sum_{s=0}^{l-1} (-1)^s \cdot n_{C_{(l-s)}}(1, \ldots, 1, i^0_{q_{(l-s)}+1}, \ldots, i^0_k) : l \text{ odd} \right\}. \tag{4}$$

The generalized shuttle algorithm converges to the bounds in (3) and (4) [4]. Moreover, one can obtain *all* feasible tables consistent with the $(k-1)$-dimensional marginals of $\mathbf{n}$ by replacing every possible value n^* that the cell $(1, 1, \ldots, 1)$ can take in (2) applied for all cells i^0. In particular, all the cells in $\mathbf{n}$ can take the same number of values—the difference between the upper and lower bounds is constant for all cells, and is equal with the number of feasible integer tables consistent with the $(k-1)$-dimensional marginals of $\mathbf{n}$.

4.3 The Decomposable Case

Log-linear models are a common way of representing and studying contingency tables with fixed marginals. In particular, a *graphical* log-linear model corresponds to conditional independence relationships that can be summarized by means of an independence graph [12]. Decomposable log-linear models [11] are a sub-class of graphical models with closed form structure and special properties that lead to explicit formulas for computing bounds [5]. The complete proof of the next theorem appears in [4].

Theorem 1. *Let $\mathbf{n}^1 = \mathbf{n}^1_{C_1}, \ldots, \mathbf{n}^p = \mathbf{n}^1_{C_p}$ be a set of tables that are consistent with each other. Assume that $C_1, \ldots, C_p$ are the minimal sufficient statistics of a decomposable log-linear model. Let $S_2, \ldots, S_p$ be the set of separators associated with $C_1, \ldots, C_p$ in the corresponding independence graph. Then:*

(i) $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$ contains at least one table.

(ii) The sharp upper bound for $i \in \mathcal{I}$ given $\mathbf{n}^1, \ldots, \mathbf{n}^p$ is

$$U\left(i; \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)\right) = \min \left\{ n^j \left(i_{C_j}\right) : j = 1, \ldots, p \right\}.$$

(iii) The sharp lower bound for $i \in \mathcal{I}$ given $\mathbf{n}^1, \ldots, \mathbf{n}^p$ is

$$L\left(i; \mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)\right) = \max \left\{ \sum_{j=1}^{p} n^j \left(i_{C_j}\right) - \sum_{j=2}^{p} n^j_{S_j} \left(i_{S_j}\right), 0 \right\}.$$

Therefore, when the fixed marginals define a decomposable graphical model, pairwise consistency of marginals implies the existence of a feasible table. In this case, the generalized shuttle algorithm converges to the bounds in Theorem 1 [4].

4.4 Finding a Feasible Integer Table

As noted above, the generalized shuttle algorithm can converge to bounds $L_s(\mathbf{T})$ and $U_s(\mathbf{T})$ even if there does not exist an integer table consistent with the fixed marginals $\mathbf{n}^1, \ldots, \mathbf{n}^p$. Since bounds make no sense when no table exists, we augment our procedure by proposing a method that will actually determine a feasible table in $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$, provided one exists.

This is done by sequentially choosing possible values for cells in $\mathcal{N}$. Once a new cell has been fixed, bounds for all the cells in $\mathbf{T}$ are updated using the shuttle algorithm. If the shuttle procedure did not stop because of an inconsistency, we pick a value for another cell. Otherwise, we choose a new value for the cell fixed at the current step. If all the cells in $\mathcal{N}$ have been fixed and the shuttle procedure completed successfully, which indicates that the dependencies in $\mathcal{Q}(\mathbf{T})$ are satisfied, we have determined a table in $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$. We denote by $T_0^{(0)} = T_0$ the set of cells fixed by the shuttle procedure when computing the bounds $L_s(\mathbf{T})$ and $U_s(\mathbf{T})$. Also, initialize the bounds arrays $L^{(0)}(\mathbf{T}) = L_s(\mathbf{T})$ and $U^{(0)}(\mathbf{T}) = U_s(\mathbf{T})$. Here is the procedure in pseudo-code:

```
Step 1. Set $l = 1$.
Step 2. Check whether $\mathcal{N}^l = \mathcal{N} \cup (\mathbf{T} \setminus \mathbf{T}_0^{(l-1)})$ is empty. If so, a feasible table has been determined; stop
    the algorithm.
Step 3. Select a cell $t^l \in \mathcal{N}^l$.
Step 4. FOR every integer $v_l \in [L^{(l-1)}(t_l), U^{(l-1)}(t_l)]$ DO
        • Initialize new bound arrays $L^{(l)}(\mathbf{T}) = L^{(l-1)}(\mathbf{T})$ and $U^{(l)}(\mathbf{T}) = U^{(l-1)}(\mathbf{T})$.
        • Set $V(t_l) = L^{(l)}(t_l) = U^{(l)}(t_l) = v_l$ and put $\mathbf{T}^{(l)} = \mathbf{T}^{(l-1)} \cup \{t_l\}$.
        • Run the generalized shuttle algorithm to update $L^{(l)}(\mathbf{T})$, $U^{(l)}(\mathbf{T})$ and $\mathbf{T}^{(l)}$.
        • If the generalized shuttle algorithm did not stop because of an inconsistency, set $l = l + 1$ and
          go to Step 2.
    END FOR.
Step 5. The algorithm stops; there does not exist a feasible integer table in $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$.
```

Instead of "blindly" searching the entire space of all possible combinations of values for the cells in $\mathcal{N}$ defined by the bounds arrays $L_s(\mathcal{N})$ and $U_s(\mathcal{N})$, the procedure reduces the search space continuously as the algorithm progresses because the bounds are updated at each step. Several heuristics exist for substantially increasing the speed of this search procedure [4]. Moreover, if we do not stop the algorithm at Step 2 once the first feasible table is generated, then *all* the integer tables in $\mathbf{T}(\mathbf{n}^1, \ldots, \mathbf{n}^p)$ will be identified.

4.5 Calculating Sharp Bounds

We can now obtain sharp bounds for any cell $t_0 \in \mathcal{T}$. Let $L_s(\mathbf{T})$ and $U_s(\mathbf{T})$ be the bounds produced by the generalized shuttle algorithm. Then

$$L_s(t_0) \leq L_{t_0} \leq U_{t_0} \leq U_s(t_0).$$

We need to "adjust" $L_s(t_0)$ and $U_s(t_0)$ to the sharp bounds L_{t_0} and U_{t_0} by making use of the simple fact that, for any integer $v \in [L_s(t_0), L_{t_0}) \cup (U_{t_0}, U_s(t_0)]$, there does not exist an integer array $V(\mathbf{T}) \in S[L_s(\mathbf{T}), U_s(\mathbf{T})]$ such that $V(t_0) = v$. To determine L_{t_0}, we start with $v = L_s(t_0)$ and attempt to determine an array $V(\mathbf{T}) \in S[L_s(\mathbf{T}), U_s(\mathbf{T})]$ with $V(t_0) = v$. If such an array exists, $L_{t_0} = v$. Otherwise, we do the same thing for $v = L_s(t_0) + 1$, and so on. Here is the complete algorithm in pseudo-code. (Again, $\mathbf{T}_0$ represents the cells t with $L_s(t) = U_s(t)$.)

```
FOR every integer value $v_l \in [L_s(t_0), L_s(t_0)+1, \ldots, U_s(t_0)]$ DO
    • Initialize new bound arrays $L^v(\mathbf{T}) = L_s(\mathbf{T})$ and $U^v(\mathbf{T}) = U_s(\mathbf{T})$.
    • Set $V(t_0) = L^v(t_0) = U^v(t_0) = v$ and set $\mathbf{T}^v = \mathbf{T}_0 \cup \{t_0\}$.
    • Run the generalized shuttle algorithm to update $L^v(t_0)$, $U^v(t_0)$ and $\mathbf{T}^v$.
    • Using the procedure described in the previous section, find out whether $S[L^v(t_0), U^v(t_0)]$ is empty.
    • If $S[L^v(t_0), U^v(t_0)]$ is not empty, set $L_{t_0} = v$ and stop the algorithm.
END FOR
```

Determination of U_{t_0} is similar: start with $v = U_s(t_0)$ and attempt to determine an array $V(\mathbf{T}) \in S[L_s(\mathbf{T}), U_s(\mathbf{T})]$ with $V(t_0) = v$. If such an array exists, U_{t_0} is equal to v; otherwise, we set $v = U_s(t_0) - 1$ and continue. In this way, we obtain sharp bounds not only for the cells in $\mathcal{N}$ but also for any cell in a cross-classification obtained by collapsing $\mathbf{n}$ across categories.

Example 1. *The left-hand panel of Table 1 contains a* $2\times2\times2\times2$ *contingency table. The generalized shuttle algorithm was used to compute the upper and lower bounds induced by fixing the* 6 *two-way marginals of this table. There is only one table consistent with this set of two-way marginals, so for all the cells in this table, the sharp integer upper and lower bounds are equal to the actual cell entries.*

The right panel in Table 1 contains comparable bounds compute using linear programming (specifically, the simplex algorithm). For all the cells, one of the bounds is different from the corresponding sharp integer bound. In some cases, the simplex algorithm found fractional bounds. In other cases, although simplex converged to integer bounds, these bounds are not sharp. Especially, cell $(1,1,2,1)$*, marked with a box in Table 1, contains a count of 0, while the actual upper bound is 1.67. Therefore the distance between the integer and the real bounds can be strictly greater than* 1.

Adjusting the bounds can be done *in parallel* for all the cells in the table [4]. Other methods for reducing the computational effort required by the generalized shuttle algorithm in the case of large sparse multi-way tables, as well as an example of calculating bounds in parallel for a 2^{16}, table are presented in [4].

		C	no		yes		C	no		yes	
A	B	D	no	yes	no	yes	D	no	yes	no	yes
no	no		1	0	$\boxed{0}$	1		[0, 1]	[0, 0.67]	$\boxed{[0, 1.67]}$	[0, 1]
	yes		0	0	1	0		[0, 0.67]	[0, 0.67]	[0, 1]	[0, 0.67]
yes	no		0	0	1	0		[0, 0.67]	[0, 0.67]	[0, 1]	[0, 0.67]
	yes		0	1	0	0		[0, 0.67]	[0, 1]	[0, 0.67]	[0, 0.67]

Table 1: A $2 \times 2 \times 2 \times 2$ table (left-hand panel) and the real bounds computed using linear programming (right-hand panel).

5 Conclusions

In this paper we have outlined how to deal with the challenging issues associated with storing and manipulating large, sparse tables in the context of statistical disclosure limitation. Dissemination systems and software implementing these techniques are described in [6, 9]. To the extent that they exploit sparsity, these techniques exhibit strong scalability properties. For instance, the fundamental data structures and methods for calculating marginal sub-tables seem able to handle almost any table that is likely to arise in practice. By contrast, the generalized shuttle algorithm of Section 4, in its current form, entails computations for every cell in the underlying table, and consequently does not scale well. Deriving a scalable version is just one of many research challenges that remain.

Acknowledgments

Support for the research was provided by National Science Foundation grant EIA–9876619 to NISS. The authors thank Stephen E. Fienberg and Stephen F. Roehrig for suggestions, comments and useful discussions. The example in Table 1 was suggested by Bernd Sturmfels.

References

[1] Y. M. M. Bishop, S. E. Fienberg, and P. W. Holland. *Discrete Multivariate Analyses: Theory and Practice*. MIT Press, Cambridge, MA, 1975.

[2] L. Buzzigoli and A. Giusti. An algorithm to calculate the lower and upper bounds of the elements of an array given its marginals. In *Statistical Data Protection (SDP'98) Proceedings*, pages 131–147, Luxembourg, 1999. Eurostat.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[4] A. Dobra. *Statistical Tools for Disclosure Limitation in Multi-way Contingency Tables*. PhD thesis, Department of Statistics, Carnegie Mellon University, 2002.

[5] A. Dobra and S. E. Fienberg. Bounds for cell entries in contingency tables given marginal totals and decomposable graphs. *Proc. Nat. Acad. Sciences*, 97:11885–11892, 2000.

[6] A. Dobra, A. F. Karr, S. E. Fienberg, and A. P. Sanil. Software systems for tabular data releases. *Int. J. Uncertainty, Fuzziness and Knowledge Based Systems*, 2002.

[7] S. E. Fienberg. Fréchet and bonferroni bounds for multi-way tables of counts with applications to disclosure limitation. In *Statistical Data Protection (SDP'98) Proceedings*, pages 115–129, Luxembourg, 1999. Eurostat.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of theACM SIGMOD International Conference on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, pages 205–216, New York, June 4–6 1996. ACM Press.

[9] A. F. Karr, A. Dobra, and A. P. Sanil. Table servers: Protecting confidentiality in tabular data releases. *Comm. ACM*, 2003. To appear.

[10] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[11] S. L. Lauritzen. *Graphical Models*. Clarendon Press, Oxford, 1996.

[12] D. Madigan and J. York. Bayesian graphical models for discrete data. *Internat. Statis. Rev.*, 63:215–232, 1995.

[13] Andrew W. Moore and Mary S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *J. Artificial Intell. Res.*, 8:67–91, 1998.

[14] Bjarne Stroustrup. *The C++ Programming Language (3rd Edition)*. Addison–Wesley, Reading, MA, 1997.

[15] Sun Microsystems. Java programming language. *http://java.sun.com*, 2002.

[16] L. C. R. J. Willenborg and T. de Waal. *Elements of Statistical Disclosure Control*. Springer–Verlag, New York, 2001.