A Little Knowledge Goes a Long Way: Faster Detection of Compromised Data in 2-D Tables

Dan Gusfield *
Division of Computer Science, University of California, Davis

Abstract

In this paper we reexamine a problem of protecting sensitive data in an n by n table of integer statistics, when the non-sensitive data is made public along with the row and column sums for the table. We consider the problem of computing the *tightest upper bounds* on the values of sensitive (undisclosed) cells. These bounds, together with tightest lower bounds (which can be efficiently computed [G]), define precisely the *smallest* intervals that an adversary can deduce for the missing sensitive cell values. Small intervals compromise the security of the undisclosed data, and in some cases violate laws on public data disclosure. We showed previously [G] that each (upper and lower) cell bound can be computed in $O(n^3)$ time by a single network flow computation, but that the bounds are not independent so that only $O(n \log n)$ upper bounds need to be computed in this (relatively expensive) way. That is, after $O(n \log n)$ initial bounds have been computed, each of the remaining (possibly $\Theta(n^2)$) bounds can be computed in *constant* time. In this note we observe that the number of initial needed bounds can be reduced from $O(n \log n)$ to $2n - 1$, by exploiting a recent result of Cheng and Hu [CH].

1 Introduction

In this paper we study the problem of protecting sensitive data in a two dimensional table of statistics when the non-sensitive data is made public. Work in this area was begun by Fellegi, Cox and Sande [FEL], [COX75], [COX77], [COX78], [COX80], [SAN], and partly reported in Denning [DEN page 360-364]. The general problem is motivated by concerns for privacy and security, and is a problem of practical importance and active interest for the U.S. Census Bureau [USDC], Statistics Sweden [DEL] and Statistics Canada [BCS]. For a more complete discussion of background and motivation see [DEN].

2 Problem Statements, Definitions and Main Results

The basic setting for the paper is that one party (the Data Collector), has a two-dimensional table, D, of cross tabulated integer statistics; each entry D(i,j) is a non-negative integer in cell (i,j) of D; R(i) is the sum of the cell values in the i'th row of D, and C(j) is the sum of the cell values in the j'th column of D. All the row and column sums are to be made public (disclosed) along with *some* of the cell values, but the remaining cell values, called *sensitive* values, are to be *suppressed* (not disclosed). Unless care is taken, however, the disclosure of the non-sensitive values might allow an adversary to deduce a small interval into which the original value of a suppressed cell must lie. Such deductions are undesirable and, in fact, the Data Collector is often required by law to guarantee that small intervals cannot be deduced from the publicly disclosed data [DEL] [BCS].

The problem for the Data Collector is to determine for each suppressed cell what interval an adversary can deduce. Then if the interval is too small for a suppressed cell, additional cell suppressions can be made before the table is released to the public. The problem for the adversary is to efficiently compute the tightest interval for each suppressed cell in the released table.

The Data Collector's problem must often be solved repeatedly in the inner loop of algorithms for more complex security problems. For example, if the upper and lower bounds are too close for some cells, then additional cells will be suppressed, and then the bounds must be recomputed to see the affect of the new suppressions. No good way is known to pick additional suppressions, so many iterations testing out different sets may be needed. In the case of government statistical agencies the volume of data is massive, so the computational burden of checking (and rechecking after additional suppressions) each table before its release is very great. Hence methods that deduce the intervals more efficiently are of great value. In this paper we show how to compute the tightest upper bounds on the suppressed cells with much fewer computations than any previous method.

To make the problem setting precise, let X be the set of all suppressed cells and for cell (i,j) $\in$ X, let x(i,j) be a *variable* denoting the value of the cell. For each row i and column j, let R*(i) and C*(j) be respectively the sum of the undisclosed values in row i of D, and the sum of the undisclosed values in column j of D. We may assume that $|X| \geq 2n$ since the exact value of any suppressed cell is immediately determined if the cell is the only suppressed cell in a row or column.

Definition: For i $\in$ R, let X(i) be the set of indices j $\in$ C such that (i,j) is a suppressed cell. Similarly, for j $\in$ C, let X(j) be the set of indices i $\in$ R such that (i,j) is a suppressed cell.

Definition: We define a *legal solution* x of D as a non-negative *integer* solution to the following system of linear equalities:

For each fixed i $\in$ R, $\sum_{j \in X(i)} x(i,j) = R^*(i)$

and

*Research partially supported by grants MCS-81/05894 and CCR-8803740 from the National Science Foundation, and by grant JSA 86-9 from the U.S. Census Bureau.

for each fixed j ∈ C, $\sum_{i \in X(j)}$ x(i,j) = C*(j).

Definition: For a suppressed cell (i,j), $u(i,j)$ denotes the tightest upper bound on the value of cell (i,j), i.e., $u(i,j)$ is the largest value that $x(i,j)$ can be given in any legal solution of D. Hence, $x(i,j) \leq u(i,j)$ for any legal solution x of D, and $x(i,j) = u(i,j)$ for at least one legal solution x of D.

The tightest lower bound on (i,j), $l(i,j)$, is the smallest value that $x(i,j)$ can be given in any legal solution of D.

The most direct way to compute the tightest upper (or lower) bound on a cell (i,j) is to solve a linear program maximizing (or minimizing) $x(i,j)$ subject to the linear inequalities stated above. This is quite inefficient both because each linear program is expensive to execute, and because for $|X|$ suppressed cells, $|X|$ such linear programs must be solved. Note that $|X|$ can be $\Theta(n^2)$. That is, as a function of n, $|X|$ can grow quadraticly.

A somewhat more efficient approach was discussed in [COX80], showing that each bound can be computed by one *minimum cost* network flow computation.

In contrast to the linear programming and minimum cost flow approaches, we showed previously [G] how to compute each bound of a suppressed cell with a single $O(n^3)$ time network flow (non-cost) computation, a large improvement over the linear programming and min-cost flow approachs. In an n by n table with $|X|$ sensitive cells, this approach yields an $O(|X|n^3) = O(n^5)$ time algorithm to compute all the bounds. Still, each network flow is relatively expensive, and so we would like to reduce the number of network flow computations needed to well below $|X|$.

In [G] we noted that the cell bounds are not independent, and surprisingly there can never be more than $2n-1$ distinct tightest upper (or lower) bound values in an n by n table, no matter how many cells are suppressed (we will prove this in section 3.1). This fact suggests that perhaps only $2n-1$ upper bound values need to be computed independently, and that after those bounds are known all the other bounds could be obtained much more quickly. In [G] we showed how to achieve this for the tightest lower bounds. We also showed that after a certain set of only $O(n \log n)$ upper bounds have been computed (by any method), all the other upper bounds can be quickly inferred from these initial upper bounds.

In this note we continue this idea, exploiting a recent result by Cheng and Hu [CH] on computing ancestor cut trees to show that only $2n-1$ initial upper bounds need to be computed. These can be computed in $O(n^4)$ time. After that, *all* the remaining $|X|-2n+1$ (perhaps $\Theta(n^2)$) upper bounds can be inferred in only $O(|X|)$ or $O(n^2)$ total time. This yields an extremely efficient approach to finding compromised cells. In order to explain the new improvement, we first review some material from [G].

3 Computing a single bound

We first consider the problem from the perspective of the Data Collector (who knows the complete D) and then from the perspective of the adversary.

Definition: A *graph* is defined by a set of *nodes* and *edges*, where each edge extends between two nodes. In an undirected graph an edge (i,j) extends between nodes i and j, and in a directed graph an edge $< i,j >$ extends from i to j. The capacity of an edge $< i,j >$, denoted $c < i,j >$, is a fixed non-negative number assigned to the edge. A *bipartite* graph is a graph where the nodes can be partitioned into two sets such that every edge in the graph touches one node of each set. Figure 2c shows a directed bipartite graph with edge capacities.

The following graph G(D) is the most important construct in this paper and its use transforms the upper bound problem from a problem in linear algebra to a problem in graph theory.

Definition: Given the complete table D, construct the bipartite graph G(D) as follows: the nodes of G(D) consist of two sets R and C; in R there is one node for each row of D, and in C there is one node for each column of D. The edges of G(D) are constructed as follows: for every suppressed cell (i,j) in D there are two directed edges between row node i and column node j; one edge is from i to j and the other from j to i. The capacity of edge <i,j> is set to M, a *finite* number larger than the largest R*(i); the capacity of <j,i> is set to D(i,j).

Figure 2c gives the graph $G(D)$ for the table in Figure 2a.

Definition Given a directed graph with edge capacities and two distinguished vertices, a *source* s and a *sink* t, an s,t *flow* is an assignment, f, of nonnegative real numbers to the edges such the following two properties hold:

Capacity Constraint. For any directed edge $< v,w >$, the flow $f < v,w >$ must be between 0 and $c < v,w >$.

Flow Conservation. For any vertex $v \neq s,t$, $f_{in}(v) = f_{out}(v)$ where $f_{in}(v)$= the total flow on arcs directed in to v, $f_{out}(v)$= the total flow on arcs directed out of v.

Definition The *value* of an s,t flow f is $\sum_v f < s,v > = \sum_v f < v,t >$, i.e., the total flow out of s and into t. A *maximum* s,t flow is an s,t flow whose value is as large as any s,t flow.

Definition A *cut* of a graph G is a partition of the nodes of G into two sets $(K, \bar{K})$. In an undirected graph, the *capacity* of the cut, denoted $c(K,\bar{K})$, is the sum of the capacities of the edges that extend between K and $\bar{K}$. In a directed graph, the capacity of the cut is the summation of the capacities of the edges that extend *from* K *to* $\bar{K}$. A cut $(K,\bar{K})$ is called an s,t cut if $s \in K$ and $t \in \bar{K}$).

The following Theorem is the most important theorem in network flow and is crucial in this paper.

Max-Flow Min-Cut Theorem [FF]: For any s,t pair, the value of a maximum flow from s to t is equal to the minimum capacity of any s,t cut.

Below we will consider a version of the network flow problem where the graph remains the same, but the choices of source and sink nodes vary.

Definition: Let FG(i,j) be the value of the maximum flow from node i to node j in G(D), i.e., i is the source node and j is the sink node. Note that FG(i,j) is not just the flow $f < i,j >$ assigned to the single edge <i,j>.

With this background, we can state the fundamental theorem from [G] which shows how to compute any single upper bound.

Theorem 1 [G]: If (i*,j*) is a suppressed cell in D, with i* ∈ R and j* ∈ C, then FG(j*,i*) is the tightest upper bound on the value of cell (i*,j*) in D, and Max[0, D(i*,j*) - FG(i*,j*) + M] is the tightest lower bound on the value of cell (i*,j*) in D. Hence for any cell (i,j), $u(i,j)$ can be computed with only a single (non-cost) network flow computation on a graph with $2n$ nodes.

Note that since u(i*,j*) = FG(j*,i*), we only need the *value* FG(j*,i*), and not the full associated flow pattern. By the Max-Flow Min-Cut Theorem, FG(j*,i*) is also the value of the minimum directed cut separating node j* from i*, so the tightest upper bounds can be obtained from algorithms which only find minimum cuts, rather than flows. This will be important below.

The adversaries problem

In the above method, and in its improvements below, we use the original values in D to compute the tightest upper bounds. The Data Collector has these values, but the adversary does not. In [G] we showed that the tightest bounds can be computed starting from *any* legal solution to D, and one legal solution can be computed in $O(n^3)$ time by one network flow computation. So the adversary simply finds one legal solution x to D and then uses x *as if* it were the original solution. Exactly the same upper and lower bounds result. Hence the adversary's problem reduces to the Data Collector's problem.

3.1 Speeding up the computation of tightest bounds

If each bound is computed independently as above, and there are $\Theta(n^2)$ suppressed cells in an n by n table, the best implied time bound for finding all the bounds would be $O(n^5)$. However, there is a great deal of interdependence between the bounds, and it is not necessary to compute each one independently. One reflection of the interdependence of the bounds is that there can never be more than $2n-1$ distinct upper bound *values* in any n by n table, no matter how many cells are suppressed; similarly there are never more than $2n-1$ distinct lower bound values. This was initially noted in [G]; below we will present a simpler direct proof. This dependence will allow us to compute the tight upper bounds using much fewer than $|X|$ network flows in general.

Definition: For two nodes i and j in G(D) define β(i,j) as the minimum of the flow values FG(i,j) and FG(j,i).

Lemma 1: Let (i,j) be a suppressed cell in D. Then in G(D), FG(i,j) > FG(j,i), where i ∈ R, and j ∈ C. Hence $\beta(i,j) = FG(j,i)$.

Proof: This is trivialy true because there is a direct edge from i to j with large capacity M, hence FG(i,j) ≥ M, while by Theorem 1, FG(j,i) is the tightest upper bound on the value of cell (i,j) which is certainly bounded by R*(i) < M. □

Lemma 1 and Theorem 1 together imply:

Lemma 2: For a suppressed cell (i,j) in D, the tightest upper bound, $u(i,j)$, on the value of cell (i,j), is equal to β(i,j), defined on G(D).

Given Lemma 2, we can now prove that the number of distinct upper bound values is at most $2n-1$.

Theorem 2: For any directed graph G with k nodes, the number of distinct β values is at most $k-1$. Hence the number of distinct upper bounds in an n by n table is at most $2n-1$.

Proof: First, suppose all the β values are known and let G' be the undirected complete graph[1] on k nodes, where the weight of any edge (i,j) is set to $\beta(i,j)$. A *maximum spanning tree* of G', denoted MST, is a tree that touches all the k nodes of G' such that the total weight of all its edges is as great as the total weight of any other tree in G'. We will show that the weight on any edge not in MST must be equal to the weight of one of the edge in MST. This will prove the theorem since there are exactly $k-1$ edges in MST.

Consider any pair of nodes (u,v) such that edge (u,v) is not in MST. The addition of edge (u,v) to MST creates exactly one cycle, call it C. The minimum weight edge, (x,y), on C must have weight greater or equal to $\beta(u,v)$, i.e., $\beta(x,y) \geq \beta(u,v)$. If not, then the spanning tree constructed by deleting (x,y) and adding edge (u,v) would have total weight greater than MST, which is impossible.

We will show the converse, that $\beta(x,y) \leq \beta(u,v)$. By the definition of $\beta(u,v)$ and by the Max-Flow Min-Cut Theorem, either the minimum cut from u to v, or the minimum cut from v to u in G has weight $\beta(u,v)$. Suppose that it is the cut $(K,\bar{K})$ from u to v. Let P be the unique path on MST from u to v. Since $u \in K$ and $v \in \bar{K}$, there must be a pair of consecutive nodes (z,w) on P such that $z \in K$ and $w \in \bar{K}$. Therefore the minimum cut from z to w has capacity less than or equal to $\beta(u,v)$. By the Max-Flow-Min-Cut Theorem again, this implies that $\beta(z,w) \leq \beta(u,v)$. Since (x,y) is the edge of smallest β on P, $\beta(x,y) \leq \beta(z,w) \leq \beta(u,v)$ as claimed.

Combining the two facts, we get $\beta(u,v) = \beta(x,y)$, and so every β value is equal to one of the $k-1$ β values in MST. □

Schnorr [SC] showed that for any directed graph on n nodes, all the $\Theta(n^2)$ β values between all pairs of nodes could be computed with only $O(n \log n)$ network flow computations. The β value for any pair of nodes which is not a source-sink pair in one of the $O(n \log n)$ flows can easily be inferred from those initial $O(n \log n)$ β values. This method is used in [G] to compute all the tightest upper bounds with only $O(n \log n)$ flows.

We now show that all the β values can be inferred from only $2n-1$ initial β values, and hence only $O(n)$ network flows are needed to obtain all the tightest upper bounds in D, no matter how many suppressed cells there are. This result comes from exploiting a recent result by Cheng and Hu [CH] on ancestor cut trees, which we briefly describe.

[1] A complete graph one where there is an edge between every pair of nodes.

4 The Cheng-Hu method for ancestor cut-trees

We begin by discussing a problem on undirected graphs. Let G be an undirected graph on N nodes. We want to find and represent the minimum cut values between all $\binom{N}{2}$ pairs of nodes, but we want to do only $O(N)$ explicit flow computations, i.e., we want to directly compute the minimum cut values of only $O(N)$ pairs; the other values will be inferred from these first ones.

We will represent the minimum cut values of a graph G, containing N nodes, with a binary tree T. Each internal vertex of T will be labeled with a source-sink pair (p,q), and will be associated with a minimum (p,q) cut in G; each leaf of T will be labeled by one node in G, and each node in G will label exactly one leaf of T. Hence T has $2N-1$ vertices. Further, for any two nodes i,j in G, if the *least common ancestor* of i and j in T is labeled by the pair (p,q), then the associated minimum (p,q) cut is a minimum (i,j) cut as well. Hence this tree represents the minimum cut values for every pair of nodes, and allows the retrieval of one minimum cut for any pair of nodes.

As an example, the graph shown in Figure 1a has an ancestor cut-tree shown in Figure 1e.

Note that for clarity, the word "node" refers to a point in G, while "vertex" refers to a point in an ancestor cut-tree.

The algorithm builds successive trees $T_0, T_1, \ldots T_{N-1} = T$, each containing one more leaf than its predecessor. The following fact about any T_i will be proved in Lemma 3 in the appendix.

Fact: If v is any internal vertex of T_i labeled with the pair (s,t), and its two children are labeled with the pairs (i,j) and (p,q), then i and j are together on one side of the (s,t) minimum cut associated with v, and p and q are together on the other side of the cut.

In order to describe the algorithm, we first define T_i' to be the subgraph of T_i consisting of the internal vertices of tree T_i, and describe how to place the leaves of T_i, given tree T_i'. This process is called *sorting* the nodes of G into T_i'.

Starting at the root of T_i', we separate the nodes of G according to the cut specified at the root node. For example, if the cut at the root is an (s,t) cut, then we place on one branch out of the root all the nodes of G on the s side of the cut, and on the other branch out of the root we place all the nodes of G on the t side of the cut. There still is a question of which edge to use for which set. Suppose the the two children of the root are labeled with the pairs (i,j) and (p,q). Given the fact stated above, we use the following rule to assign the two parts of the (s,t) cut to the two edges out of the root of T_i': The part of the (s,t) cut containing i and j is placed on the edge from the root to its child labeled with the (i,j) cut, and the part containing p and q is placed on the other edge out of the root.

In general, at any vertex x of T_i', we split the nodes of G that are on the edge leading to x into the two edges out of x, according to how the cut at x separates these nodes. To decide which set goes on which of the two edges out of x, we follow the same rule stated for the root. If x is a leaf of T_i', then the nodes of G on the edge entering x are split into two children of x according to how the cut labeling x splits these nodes. The children of x are then leaves in T_i.

For example, in Figures 1b through 1e, the nodes written on the edges of the intermediate trees show the sorting process.

An added feature of any tree T_i, which will be maintained inductively, is that after each iteration of the algorithm, the set of nodes contained in any leaf will have exactly one designated node called the *representative* of that set.

The full algorithm is now the following:

4.0.1 The Cheng-Hu Algorithm

Set k to 0.

The initial tree T_0 consists of a single leaf containing all the nodes of G. Arbitrarily set one of the nodes to be the representative of this leaf.

Repeat

Pick a leaf x of T_k which contains more than one node of G; suppose i is the representative of x, and let j be any other node of G in x. Declare j to be a representative.

Find a minimum cut $(X, \bar{X})$ between i and j; let its value be $f(i,j)$, and assume that $i \in X$.

Find the closest ancestor vertex y of x in T_k whose cut value is less than or equal to $f(i,j)$; let z be the vertex below y on the path from y to x in T. Create a vertex labeled with (i,j), and place it between y and z in T_k. Remove all the leaves of T_k, creating tree T_{k+1}'; then sort the nodes of G into T_{k+1}', creating tree T_{k+1}. Set $k := k+1$.

Until each leaf node of T contains only a single node of G.

Note that z may be a leaf of T_k. Note also that at least one of the children of vertex x is a leaf of T_{k+1}. A proof of correctness of the algorithm is contained in the appendix. For the original proof by Cheng and Hu, see [CH] or [CH1].

Once an ancestor cut tree T has been obtained, it is easy to explicitly find $f(i,j)$ for every pair (i,j) in a total of $O(N^2)$ time. One way to do this is to process T from the bottom up: an internal vertex x in T is processed only after its two children x' and $x"$ have been processed. Inductively, after vertex x' has been processed, a list has been compiled of all the nodes of G at leaves in the subtree of x. Suppose that the cut value associated with vertex x if c. To process vertex x we first set $f(i,j) := c$ for every pair of nodes (i,j) where i is in the list for x' and j is in the list for $x"$. Then these two lists are concatenated and the processing of x is finished.

If we only want a subset of the $\binom{N}{2}$ β values (as will be the case in computing upper bounds in D), then we can find each desired β in $O(1)$ time using fast least-common-ancestor algorithms [SV]

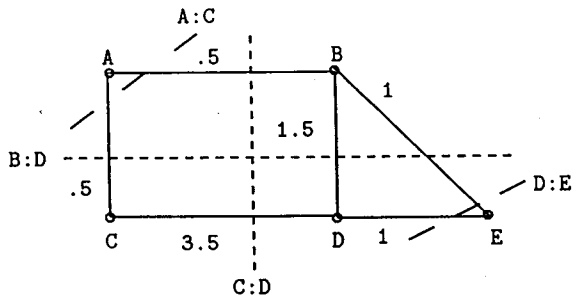


Figure 1a: Graph G and the cuts used by the algorithm.

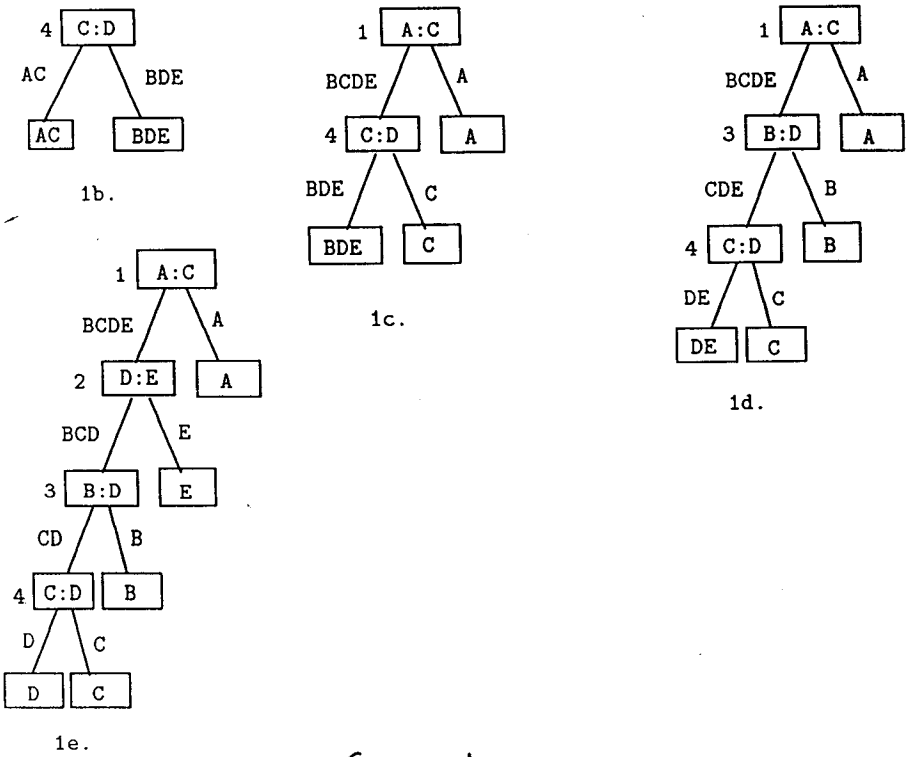
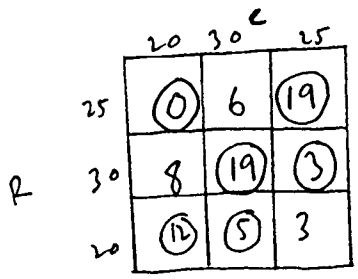
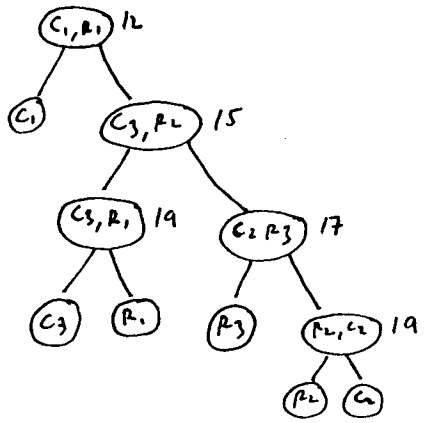


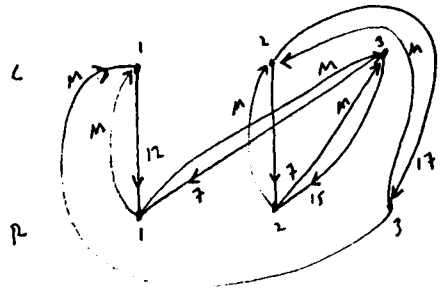
Figure 1



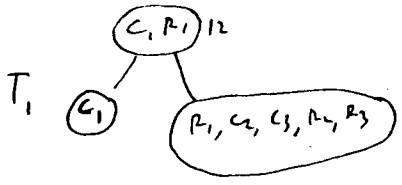
2a. Full table D. The suppressed cells are circled.



2b. The ancestor cut tree for the tightest upper bounds of the suppressed cells in D.

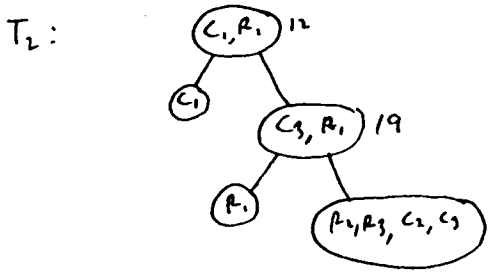


2c. Graph G(D).

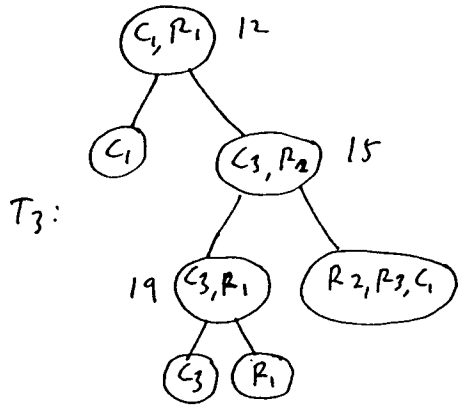


2d.

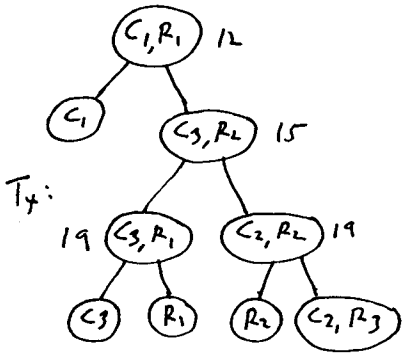
2d. The first pair used to build the ancestor cut tree is (C_1, R_1) with cut $\{C_1\}\{C_2, C_3, R_1, R_2, R_3\}$ of value 12.



2e. The next pair is (C_3, R_1) with cut $\{R_1\}\{R_2, R_3, C_1, C_2, C_3\}$ of value 19.



2f. The next pair used is (C_3, R_2) with cut $\{C_2, R_1\}\{R_2, R_3, C_1, C_2\}$ with cut value 15.



2g. The next pair is (C_2, R_2) with cut $\{C_1, C_2, R_3\}\{C_3, R_1, R_2\}$ with value 19.

2h. The final pair is (C_2, R_3) with cut $\{C_2\}\{R_1, R_2, R_3, C_1, C_3\}$ of value 17.

The final ancestor cut tree T is shown in figure 2b. Notice that only five cut computations were required, although there are six suppressed cells.

Figure 2

[HT]. Therefore to find $|X|$ particular β values, once the ancestor tree has been constructed, requires only $O(N + |X|)$ time.

4.1 Extension to arbitrary cut capacities

In [CH] the following generalization for an undirected cut capacity is considered. Suppose that instead of defining the capacity of an undirected cut $(K, \bar{K})$ as the sum of the edge capacities crossing the cut, we assume that the undirected cut is assigned some *arbitrary* value as its capacity. Then for a pair of nodes (i, j) we define $f(i, j)$ as the minimum capacity of all the undirected cuts separating i from j. Close examination of the proof of correctness of the CH method given in the appendix shows that the proof remains correct for this more general definition of cut capacity (see [CH] for the original proof). That is the above method correctly builds an ancestor cut tree for the more general notion of cut capacity and function $f(i, j)$. So, the ancestor cut tree for any such f can be constructed with the computation of $f(i, j)$ for exactly $N - 1$ (i, j) pairs. Of course, depending on how the cut values $(K, \bar{K})$ are defined and represented, computing $f(i, j)$ for a pair (i, j) may not be efficient. None-the-less, only $N - 1$ such computations are required to represent all the $\binom{N}{2}$ f values in an ancestor tree.

5 Using ancestor trees to compute β values

In this section we will apply the above generalized notion of cut capacities to show how to model the β values.

In a *directed* graph G with N nodes and edge capacities, let $C(K, \bar{K})$ be defined as the sum of the capacities of the edges crossing from K to $\bar{K}$, and let $C(\bar{K}, K)$ be the sum of the capacities of edges from $\bar{K}$ to K. Then we define the *capacity* of the cut (partition) $(K, \bar{K})$ to be the minimum of $C(K, \bar{K})$ and $C(\bar{K}, K)$, and we define $f(i, j)$ as before to be the minimum *capacity* of all the cuts which separate i from j (that is, put i and j in different sets in the partition). Note that under this definition, $f(i, j) = \beta(i, j)$.

So, the Cheng-Hu method can be used to find all the β values, provided that when any pair (i, j) is specified, the required $f(i, j)$ can be computed and a cut of that value separating i from j can be found. Now by the Max-Flow Min-Cut Theorem $f(i, j) = min[F(i, j), F(j, i)]$, where $F(i, j)$ is the maximum flow value from i to j, and $F(j, i)$ is the maximum flow value from j to i. Since G is directed, $F(i, j)$ need not be equal to $F(j, i)$. Hence $\beta(i, j) = f(i, j)$ can be computed by two network flow computations on the directed graph G.

Clearly then, with the above definitions, the f values can be computed by the Cheng-Hu algorithm, and the tree then represents all the $\binom{N}{2}$ β values. So specializing this to graph $G(D)$, we can find all the β values of the node pairs in $G(D)$ with only $4n - 1$ network flow computations ($G(D)$ has $N = 2n$ nodes, and each $f(i, j)$ value computed by the Cheng-Hu method requires two network flow computations) plus an additional bottom up processing of T. Then if (i, j) is a suppressed cell in D, it tightest upper bound is given by $\beta(i, j)$. In summary we have

Theorem 3: All the $|X|$ tightest upper bounds can be computed using only $4|X| - 1$ network flow computations on a graph with $2n$ nodes, plus $O(|X|)$ or $O(n^2)$ additional time.

The advantage of this method becomes more compelling as n grows and when $|X|$ (which can grow as $\Theta(n^2)$) is much larger than $4n$.

Although the example is much too small to illustrate the computational efficiency of the method verses independent computation of each bound, in Figure 2 we show the table from [DEN page 363], along with the ancestor cut tree for the tightest upper bounds for suppressed cells.

5.1 Additional Comments

There are two additional observations that speed up in practice the above method when specialized to $G(D)$. First, whenever the Cheng-Hu method requires computing $f(i, j)$ where (i, j) is a suppressed cell in D, then $f(i, j) = F(j, i)$ (by Lemma 1) so only one network flow computation is required. Second, since we really are only interested in knowing the β values for the $|X|$ node pairs in G corresponding to suppressed cells in D, we might not need run the Cheng-Hu algorithm to completion. In order to describe an early stopping rule, we define S_k as the set of node pairs for which f has been explicitly computed, through the k'th iteration of the Cheng-Hu algorithm. That is, these are the pairs which label the internal vertices of T_k. We can then prove that if i and j are in S_k (they need not be in the same label), then $f(i, j)$ is the value at the least common ancestor of the leaves containing i and j in T_k. It is easy therefore to maintain a count of how many of the $|X|$ tightest upper bounds in D can be obtained from each successive T_k. Let us denote that number as B_k. When $|X| - B_k \leq 4n - 2|S_k|$, then we abandon the Cheng-Hu algorithm and compute all the remaining upper bounds independently.

Using just the first practical speedup, the tree in Figure 2 is computed with only five network flows rather than the six needed for independent computation of each bound. Of course, a savings of one flow is not the goal, but rather a savings of $|X| - 4n$, when that value is large.

It is interesting to note the meaning of $\beta(i, j)$ in $G(D)$ for a pair (i, j) which is not a suppressed cell in D. $\beta(i, j) + D(i, j)$ is the tightest upper bound on cell (i, j) that would result from suppressing cell (i, j), assuming all other aspects of D remain unchanged.

We should note that the $O(n \log n)$ network flows of Schnorr's method can be implemented to run in $O(n^4)$ overall time [SC] (the same as if only $O(n)$ flows were needed), but the implementation needed to achieve this amortized time bound is involved, and experiments with that method have shown it to be disappointingly slow. With only $4n - 1$ flows, the same worst case time bound is achieved, but the new method described here doesn't need any of the involved implementation details that the Schnorr method uses to achieve its amortized bound. Further, when the graph has special properties (for example it is sparse, or the capacities are small) allowing a specialized faster-than-general network flow method to be used, the time to build the ancestor tree is automatically improved, while the Schnorr method may not be

able to exploit the special properties of the network.

Although we have considered square tables in this paper, any n by m table can easily be handled. In all the results presented, simply replace $2n$ with $n+m$. For example, the method must compute at most $n+m-1$ initial β values.

6 Related results on 2-D data protection

Related work of note appears in in [K] and [KG]. In particular, one may ask whether a given linear function of the suppressed cells is invariant over all the legal solutions of D, i.e., does the function take on the same value for any legal solution of the table. In this paper we essentially considered the simplest linear function, that of a single cell value, and discussed how to compute bounds on its value. In [KG] it is shown that given *any* linear function $g(S)$ of the variables corresponding to a set S of suppressed cells in D, one can determine in linear time if $g(S)$ is an invariant, i.e., has the same value in every legal solution of D. If $g(S)$ is an invariant, then its exact value can be deduced by an adversary even if the adversary cannot deduce or closely approximate the values of the individual cells in S.

7 Acknowledgment

I would like to thank Dalit Naor for helpful conversations concerning the Cheng-Hu method, and for contributing to the proof presented in the appendix.

8 References

BCS Brackstone, Chapman, and Sande. Protecting the Confidentiality of Individual Statistical Records in Canada, Statistics Canada, March 1983.

CH C.K. Cheng, T.C. Hu. Maximum Concurrent Flow and Minimum Ratio Cut. Computer Science and Engineering Technical Report no. CS88-141, Dec. 1988, University of California, San Diego.

CH1 C.K. Cheng, T.C. Hu. Ancestor Tree for Arbitrary Multi-Terminal Cut Functions. Computer Science and Engineering Technical Report no. CS88-148, 1989, University of California, San Diego.

COX75 Cox, Lawrence. Disclosure Analysis and Cell Suppression, Proceedings of the American Statistical Association, Social Statistics Section, 380-382, 1975.

COX77 Cox, Lawrence. Suppression Methodology in Statistical Disclosure, Proceedings of the American Statistical Association, Social Statistics Section, 750-755, 1977.

COX78 Cox, Lawrence. Automated Statistical Disclosure Control, Proceedings of the American Statistical Association, Survey Research Methods Section 177-182, 1978.

COX80 Cox, Lawrence. Suppression Methodology and Statistical Disclosure Control, Journal of the American Statistical Association, Theory and Methods Section, June 1980, Vol. 75, Number 370.

DEL Tore Dalenius. Controlling Invasion of Privacy in Surveys. Department of Development and Research, Statistical Research Unit, Statistics Sweden.

DEN Denning, Dorothy. Cryptography and Data Security. Addison- Wesley 1982.

FEL I.P. Fellegi, On the question of statistical confidentiality. Journal of the American Statistical Association, March 1972, vol. 67, no. 337, p. 7-18.

FF Ford, L. and Fulkerson, D. Flows in Networks, Princeton University Press, 1962.

G Gusfield, Dan. A Graph Theoretic Approach to Statistical Data Security, SIAM Journal on Computing, vol. 17, No. 3, June 1988, p. 552-571.

HT D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. SIAM J. Computing, 13 (1984) p. 942-946.

K M. Kao, Systematic Protection of Precise Information on Two Dimensional Cross Tabulated Tables. Doctoral Dissertation, Yale University, December 1986.

KG M. Kao, and D. Gusfield. Efficient Detection of Leaked Information in Cross Tabulated Tables: Linear Invariant Test. Technical report no. 255, Indiana University, Computer Science Department, July 1988.

SC C.P. Schnorr, Bottlenecks and edge connectivity in unsymmetrical networks, SIAM J. Computing, Vol. 8, No. 2, May 1979.

SAN G. Sande, Automated cell suppression to preserve confidentiality of business statistics. Statistical Journal of the United Nations ECE 2(1984) 33-41, p. 33-41.

SV B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. SIAM J. on Computing, 17 (1988), p. 1253-1262.

USDC U.S. Department of Commerce (1978), Statistical Policy Working Paper 2: Report on Statistical Disclosure and Disclosure-Avoidance Techniques, Washington, D.C.: U.S. Government Printing Office.

9 Appendix: Correctness of the Cheng-Hu Algorithm

The key to the correctness of the algorithm is that every intermediate tree T'_{k+1} is sortable. For a tree to be sortable, we need that the (s,t) cut (say) at any internal vertex x partitions the nodes on the edge coming into x such that all nodes in labels of the internal vertices in the left subtree of x are on one side of the (s,t) cut, and all nodes in labels in the right subtree of x are on the other side of the cut. If this condition is satisfied, then

the tree is sortable. To prove that the tree is always sortable, we start with the following definition.

Definition: For a vertex x in T_k, let p and q be any two nodes of G which are each used in some label (not necessarily the same label) of a vertex in the subtree of T_k rooted at x. We say that p and q are *connected* in the subtree of x, if there exists a sequence $(p, v_1), (v_1, v_2), ..., (v_j, q)$, where the second node in each pair is the first node in the succeeding pair, and each pair is a label of a vertex in the subtree of x.

For example, in Figure 1e let x be the root of the tree, and let p be A and q be B. Then p and q are connected in the subtree rooted at x through the path $(A, C), (C, D), (D, E)$.

Lemma 3 Any intermediate tree T'_{k+1} produced by the algorithm is sortable. In addition, all of the nodes in pairs labeling the vertices in the subtree of x are connected.

Proof We prove the Lemma inductively. T_0 is clearly sortable, and since it has no internal nodes, it has the claimed connectedness property. Suppose T'_k is sortable and has the connectedness property. Let (i, j) be the source-sink pair used by the algorithm to create T_{k+1}, where i is a representative of a leaf w in T_k. Let x denote the new vertex created, labeled with the pair (i, j).

Suppose first that in T'_{k+1}, x takes the place that w occupied in T_k, then T'_{k+1} is sortable since T'_k was, and the new (i, j) cut simply splits the nodes coming into x into two branches. Further, the label of the parent of x must contain either i or j (by the way that representatives of leaves are created, and the fact that T'_k was sortable). Suppose, w.l.o.g., its label is (i, s). Then, there is a sequence $(s, i), (i, j)$ in T_{k+1}, so j is also connected to s in T_{k+1}. Since the connectedness property holds for T_k, s is connected to all nodes in labels of the subtree of x in T_{k+1}.

Now suppose that x is inserted between two internal vertices y and z, where y is the parent of z. We first show that T'_{k+1} is sortable. All nodes that were on the incoming edge into z in T_k are now on the incoming edge into x in T'_{k+1}. Hence, all vertex labels of the subtree rooted at x are contained on the set of nodes coming into x. In particular, i and j are on that edge.

For any internal vertex, labeled (s, t), in the subtree rooted at x, $f(s, t) > f(i, j)$, and therefore the (i, j) cut cannot separate s from t. Further, by induction, all nodes used in labels in this subtree are connected, so it follows that the (i, j) cut cannot separate any two nodes p and q which are used as labels in this subtree. For suppose that the (i, j) cut did separate p from q. Then the (i, j) cut must also separate two nodes in a label on the chain of labels connecting p and q, a contradiction. Thus, the cut at x partitions the incoming nodes into two sets, such that one set contains all labels in the subtree of x. The part of the cut containing these nodes is passed on to z, while the other part becomes a leaf of T_{k+1} below vertex x. From z downwards, the tree is certainly sortable as before.

To show that the connectedness property holds for T_{k+1}, consider an arbitrary subtree of T_{k+1}. If it does not contain the new vertex x, then the claim clearly holds for it. If it does contain x, then it must contain the immediate ancestor of leaf w in T_k. As before, assume that the label of that ancestor is (i, s). Now $(i, s), (i, j)$ is a sequence in the subtree of T_{k+1} rooted at x, and by the induction hypothesis, in T_k, s and t are connected in the subtree of z for any node t in a vertex label in the subtree of z. It follows that i and t and j and t are connected in the subtree of x in T_{k+1}, and the connectedness property holds.

Lemma 4 If the least common ancestor of nodes i and j has label (p, q), then the associated minimum (p, q) cut separates i and j, and so $f(i, j) \leq f(p, q)$.

The proof follows immediately from the sorting process.

Lemma 5 Let $S = (i, v_1), (v_1, v_2), ..., (v_k, j)$ be a sequence of node pairs, where the second node of each pair is the first node of the succeeding pair. Then $f(i, j) \geq min[f(x, y) : (x, y)$ is a pair in $S]$.

Proof Let $(X, \bar{X})$ be a minimum (i, j) cut. Since $i \in X$ and $j \in \bar{X}$, there must be a pair $(v_h, v_{h+1}) \in S$ such that $v_h \in X$ and $v_{h+1} \in \bar{X}$, and hence $f(v_h, v_{h+1}) \leq f(i, j)$, and the Lemma follows.

Theorem 2: For any nodes i and j in G, the cut (p, q) at the least common ancestor x of i and j in T is a minimum (i, j) cut in G.

Proof: By Lemma 4, $f(i, j) \leq f(p, q)$. Now consider the set of vertex labels in the subtree of T rooted at x. By applying the connectedness property shown in Lemma 3, we can connect *all* the vertex labels in the subtree of x into a single sequence $(i, v_1), (v_1, v_2), ..., (v_k, j)$ (vertex labels may be repeated). Then by Lemma 5, $f(i, j) \geq min[f(u, v) : (u, v)$ is a vertex label in the subtree of $x]$. But by construction of T, x has a smaller associated cut than any vertices in its subtree, and so $f(p, q) \leq f(u, v)$ for any label (u, v) in the subtree of x. Therefore $f(i, j) \geq f(p, q)$, and the theorem follows.