Minimax Programs and Bitonic Column Matrices

P. A. Tucker*, T. C. Hu
ptucker@cs.ucsd.edu, hu@cs.ucsd.edu
UCSD Computer Science and Engineering Dept.
Technical Report CS99-624

June 17, 1999

Abstract

This report describes an optimization problem called a minimax program that is similar to a linear program, except that the addition operator is replaced by the maximum operator in the constraint inequalities. The relation of this problem to some well-known problems is clarified. An interesting special case, bitonic columns, is identified, and a new, efficient algorithm is presented for its solution. Also presented is an efficient algortihm for recognition of matrices with the bitonic columns property, which is an extension of the PQ-tree reduction algorithm.

1 Introduction

Over the last fifty years, countless problems of practical interest have been formulated as a linear program. Not only has the linear programming model proven to be widely applicable, but ongoing research has discovered highly effective algorithms for solution of various classes of linear programs. Linear programming represents one of the major achievements of the operations research and mathematical programming community.

In this report we introduce an optimization problem we call a "minimax program" that very much resembles a linear program. The task is still to minimize a linear function, but in the constraint inequalities we replace the *addition* operator by the *maximum* operator. With this change we obtain a problem formulation that straightforwardly captures the structure of some real optimization problems. The problem also turns out to be NP-complete, with a close similarity to set cover.

We describe an efficient algorithm for solving an interesting subclass of minimax programs, those whose constraint matrices have columns with the "mountain property," which is a generalization of the consecutive ones property (*i.e.* the rows can be permuted so that entries in each column first increase, then decrease). We also present an efficient algorithm for recognition of matrices with this properties. The algorithm is an extension of the PQ

*Supported in part by a National Science Foundation Graduate Fellowship.

tree reduction algorithm of Booth and Lueker [1]. More recently developed algorithms (e.g. lex-bfs [10]) are now preferred to the original PQ tree reduction algorithm for, e.g., interval graph recognition, but our extension shows that the original algorithm has previously undescribed adaptability that the newer algorithms do not obviously share.

The rest of this report is organized as follows. Section 2 defines a minimax program. Subsections describe its relation to other problems, and an interesting and practical special case (bitonic columns) that can be solved quickly. Section 2.4 presents an efficient algorithm to solve minimax programs with the proper bitonic columns. Section 3 describes an efficient algorithm for detecting whether a matrix has a bitonic column property. Section 4 describes an example application of minimax programs.

2 Minimax Programs

In a linear program the task is to

$$\begin{array}{rrcll} \min & z & = & \sum c_j x_j & (j = 1, \ldots, n) \\ \text{subject to} & & & & \\ & \sum a_{ij} x_j & \geq & b_i & (i = 1, \ldots, m) \\ & x_j & \geq & 0. & \end{array} \tag{1}$$

The linear program model assumes linearity of both the objective function and the constraint inequalities. Implicit in the linearity of constraints is additivity of the basic vectors in satisfying the requirement vector $\mathbf{b}$.

A simple example of a typical linear programming application is the selection of food servings to satisfy nutritional requirements at minimum cost. Suppose there are n kinds of food to choose from, while m different nutrient requirements are to be fulfilled. In this example, the additivity assumption is justified because the nutritional properties of food are believed to be additive. The daily requirement for protein, for example, can be satisfied by eating a mix of food servings throughout the day whose protein contents sum to the requirement. It is not necessary to eat a single large serving of one food sufficient to satisfy the protein requirement by itself, and each food serving can contribute to the satisfaction of many nutritional requirements.

However, the additivity assumption does not always hold. Consider a slight variation on the problem where instead of purchasing food, our goal is to purchase poisons to kill a mixture of household pests. The essential difference is that we assume the effects of different kinds of poison are independent. We suppose that all of the poisons are to be applied sequentially or even simultaneously (perhaps by fumigating the house), and for each poison there is a lethal dose associated with each species of pest. If we apply at least the lethal dose the pest population will be exterminated, but anything significantly less than that dose will not noticeably harm the pests. Moreover, if we apply a sub-lethal dose of one poison, the lethal dose of a second poison, towards the same pest species, is not reduced. Because the mechanisms of action of different kinds of poison are different, the pests can survive sub-lethal doses of a number of different poisons within a short period of time. In purchasing a minimum cost blend of poisons, we have to ensure that it contains a lethal dose of at least one poison for every pest species.

If we suppose there are n kinds of poison to choose from, and m species of pest to exterminate, the problem can be modeled by the same structure as a linear program, except that we replace the addition operator in the constraint inequalities with the maximum operator.

$$\begin{array}{rrcll} \min & z & = & \sum c_j x_j & (j = 1, \ldots, n) \\ \text{subject to} & & & & \\ & \max_j(a_{ij} x_j) & \geq & b_i & (i = 1, \ldots, m) \\ & x_j & \geq & 0. & \end{array} \tag{2}$$

We call this problem formulation a *minimax program.* Minimax programs can arise as a natural problem formulation in domains such as software testing by a mixture of test methods [7], an application to be discussed Section 4.

It is convenient to define a standard form for minimax programs. First it should be observed that there is no need for the presence of negative coefficients in the cost or requirement vectors, or in the constraint matrix A. If negative coefficients are present we can eliminate them through arithmetic manipulation, or replace them by 0 without affecting the optimal solution, or they cause the problem to be unbounded and hence not well-formulated. So without loss of generality we can require that all coefficients be non-negative. Then, since there is no integer restriction on coefficients, we can normalize the cost and requirement vectors to $\vec{\mathbf{1}}$. The result is a standard form in which the minimax problem is completely described by its constraint matrix A.

$$\begin{array}{rrcll} \min & z & = & \sum x_j & (j = 1, \ldots, n) \\ \text{subject to} & & & & \\ & \max_j(a_{ij} x_j) & \geq & 1 & (i = 1, \ldots, m) \\ & x_j & \geq & 0 & \end{array} \tag{3}$$

As an illustration, let the constraint matrix A in (3) be as follows.

$$\begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 & \frac{1}{3} & \frac{1}{8} \\ \frac{1}{4} & 0 & \frac{1}{3} & 0 & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{8} & 0 & 0 & \frac{1}{4} & \frac{1}{10} & \frac{1}{4} \end{array} \tag{4}$$

Then the minimax program (3) with constraint coefficients (4) has many feasible solutions such as

$$\begin{array}{ll} (i) & x_1 = 8 \quad \text{with total cost 8} \\ (ii) & x_2 = 2, x_3 = 3, x_4 = 4 \quad \text{with total cost 9} \\ (iii) & x_4 = 4, x_5 = 4 \quad \text{with total cost 8} \\ (iv) & x_5 = 3, x_6 = 4 \quad \text{with total cost 7.} \end{array}$$

2.1 Relation to Other Problems

Consider the usual integer program formulation of an arbitrary instance of weighted set cover.

$$\begin{array}{rcl} \min & z & = \sum c_j x_j \\ \text{subject to} & & \\ & \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} \vec{x} & \geq \vec{1} \\ & x_j & \in I^+ \end{array}$$

The constraint matrix A is $(0,1)$, and the requirement vector $\mathbf{b}$ is $\vec{1}$. If we take these same parameters and put them into the minimax program model, in other words change the constraint inequalities to

$$\begin{array}{llllll} \max(& x_1, & 0, & x_3, & 0 &) \geq 1 \\ \max(& x_1, & x_2, & 0, & x_4 &) \geq 1 \\ \max(& x_1, & x_2, & x_3, & 0 &) \geq 1 \\ \max(& 0, & x_2, & x_3, & x_4 &) \geq 1 \end{array}$$

retaining the cost vector $\mathbf{c}$ and eliminating the integer requirement on x_j, then the optimal solutions are unchanged.

Theorem 1 *Let* $\min cx$ *such that* $Ax \geq \vec{1}$, $x_j \in I^+$ *where* $a_{ij} \in \{0,1\}$, $c_j \in R^+$ *be an integer program instance of weighted set cover, and let* $\min cx$ *such that* $\max_j(a_{ij}x_j) \geq 1$, $x_j \in R^+$ *be a minimax program (not necessarily in standard form) with the same cost vector* c *and constraint matrix* A*. Any feasible solution to the integer program is feasible for the minimax program. Any feasible solution to the minimax program maps onto a feasible solution to the integer program of lesser or equal cost, in* $O(m)$ *time.*

Proof. In the first direction, let $x_1, x_2, \ldots, x_m$ be any feasible solution to the integer program. Consider any row i of the constraint matrix; since A is $(0,1)$, there must be some $x_j \geq 1$ corresponding to some $a_{ij} = 1$ to satisfy the row constraint. This same value of x_j then also satisfies the row i constraint in the minimax program since $a_{ij}x_j \geq 1$. Hence the solution is also feasible for the minimax program.

In the other direction, let $x_1, x_2, \ldots, x_m$ be any feasible solution to the minimax program, and let x' be an m-length vector where $x'_j = \lfloor x_j \rfloor$. Consider any row i of the $(0,1)$ matrix A; for the minimax row constraint to be satisfied, there must exist some $x_j \geq 1$ corresponding to some $a_{ij} = 1$. Then $x'_j \geq 1$ and row i of the integer program constraints is also satisfied. Hence x' is feasible for the integer program and $\sum_j x'_j \leq \sum_j x_j$. ∎

A corollary of this theorem is that optimization of minimax programs is NP-complete.

There is also a reduction of *any* minimax program to an equivalent set cover problem in integer program formulation. The general reduction technique involves expanding each column of the minimax constraint matrix into m $(0,1)$ columns, and assigning appropriate cost coefficients to the new columns.

Theorem 2 *Let A be the constraint matrix of a minimax program in standard form, and let*

$$c'_{j_i} = 1/a_{ij}$$

and

$$a'_{k,j_i} = \begin{cases} 1 \text{ if } a_{kj} \geq a_{ij} \\ 0 \text{ otherwise} \end{cases}$$

define the cost vector and constraint matrix of an integer program $\min c'x$ *such that* $A'x \geq \vec{1}$, $x \in I^+$. *Then any feasible solution to the minimax program maps onto a feasible solution to the integer program, of lesser or equal cost, and vice versa.*

Proof. Let $x_1, x_2, \ldots, x_m$ be any feasible solution to the minimax program. Consider the constraint imposed by any row i of A; it must be satisfied by at least one x_j such that $a_{ij}x_j \geq 1$. Within column j of A, examine the entries in other rows and let k be $\min_i a_{ij}$ such that $x_j a_{ij} \geq 1$ (i.e. x_j satisfies row k), then set

$$x'_{j_i} = \begin{cases} 1 \text{ for } i = k \\ 0 \text{ for } i \neq k \end{cases}$$

among the integer program solution components corresponding to x_j. Observe that $a_{ij} \geq a_{kj}$, so $a'_{i,j_k} = 1$ by the definition above, and $a'_{i,j_k}x'_{j_k} = 1$ and the constraint imposed by row i of A' is satisfied in the integer program. The cost of the solution to the integer program is

$$\begin{aligned} \sum_j \sum_i c'_{j_i}x'_{j_i} &= \sum_j 1/a_{kj} \geq 1/a_{ij} \text{ where } k \text{ is selected for each } j \text{ as above} \\ &\leq \sum_j x_j. \end{aligned}$$

In the other direction, consider any feasible solution to the integer program, x'. For each associated set of variables $x'_{j_1}, \ldots, x'_{j_m}$, if any is non-zero, take the non-zero x'_{j_i} associated with maximum c'_{j_i}, and assign $x_j = c'_{j_i}$. If none of $x'_{j_1}, \ldots, x'_{j_m}$ are non-zero, then assign $x_j = 0$. If $a'_{i,j_k}x'_{j_k} \geq 1$ (i.e. the ith row constraint is satisfied), then $x_j \geq c'_{j_k} = 1/a_{kj}$. By the construction of this theorem, $a'_{i,j_k} = 1$ only if $a_{ij} \geq a_{kj}$, so $c'_{j_k} \geq 1/a_{ij}$, and $x_j a_{ij} \geq 1$, satisfying the ith row constraint in the minimax program. The cost of the derived solution to the minimax program is

$$\begin{aligned} \sum_j x_j &= \sum_j \max_i c_{j_i}x_{j_i} \\ &\leq \sum_j \sum_i c_{j_i}x_{j_i}. \end{aligned}$$

∎

In illustration of the reduction given by the theorem, the minimax program with unit costs and constraint inequalities

$$\max \begin{bmatrix} x_1 & x_2 & x_3 \\ 5 & 1 & 1 \\ 2 & 4 & 3 \\ 1 & 1 & 4 \end{bmatrix} \geq \begin{matrix} 1 \\ 1 \\ 1 \end{matrix}$$

$$x_j \geq 0$$

is equivalent to an integer program with the following $(0,1)$ constraint matrix and cost coefficients.

$$\text{sum}\begin{bmatrix} \frac{1}{5}x_{1_1} & \frac{1}{2}x_{1_2} & 1x_{1_3} & 1x_{2_1} & \frac{1}{4}x_{2_2} & 1x_{2_3} & 1x_{3_1} & \frac{1}{3}x_{3_2} & \frac{1}{4}x_{3_3} \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{matrix} \geq \\ \\ \\ \\ \end{matrix} \begin{matrix} 1 \\ 1 \\ 1 \\ \\ \end{matrix}$$

$$x_{j_i} \in I^+$$

This reduction shows that a minimax program can be viewed as a compact representation of a set cover instance whose coefficients allow columns of the constraint matrix to be packed together.

It follows from Theorem 2 and its proof that any solution to the integer program can be converted in polynomial time to an equally good solution to the minimax program. Hence, a corollary is that *any exact or ϵ-approximation algorithm for set cover or a general integer program yields an equivalently accurate algorithm for the minimax program.* For example, the greedy algorithm for set cover gives an $O(\log n)$ approximation. This and other approximation and special case results for set cover are surveyed in [5].

However, the reduction to an integer program incurs a penalty by inflating the problem representation size by a factor of $O(m)$; if the size of the original minimax program is mn, the size of the derived set cover problem is $O(m^2 n)$. Hence it is desirable to solve a minimax program directly by an efficient algorithm. In a special case to be considered in the next section, *an exact solution to the minimax program can be obtained with a time complexity less than that of the reduction to set cover.*

2.2 A Fast Special Case

Among NP-complete problems that have a graph representation it is known that instances with the interval graph property can often be solved efficiently, even in linear time (*e.g.* Hamilton circuit [8] and vertex cover [9]).

The interval graph property is intimately related to the consecutive ones property of $(0,1)$ matrices [3]. A matrix has this property if its rows can be permuted such that all ones in every column are consecutive.

Although having the interval property is a significant restriction on the general class of graphs, interval graphs have the virtue of corresponding to many problems that arise naturally in the real world. The graph proximity corresponding to the intervals can model physical or temporal proximity constraints in many application contexts. For example, Fulkerson and Gross's paper [3] arose out of a study of genetic mutations where the interval property modeled adjacency in a DNA strand. A number of scheduling problems, whether for rooms or processors, turn out to have the interval graph property when represented as a graph coloring problem where edges indicate incompatibilities. Interval graphs also play a part in practical algorithms for synthesis and physical layout of circuit designs. See Golumbic [4] for a survey.

2.2.1 Linear Time Set Covering for Consecutive Ones

As an illustration of the efficiency with which problem instances with the interval graph property can be solved, and in preparation for the broader special case of minimax programs we next address, we present a simple linear time algorithm for unweighted set covering when the problem matrix has consecutive ones.

Using the integer program formulation of set cover, we assume that the constraint (set membership) array A is presented in a conforming permutation so that all ones are consecutive in every column. The algorithm first scans the array once to record the number of consecutive ones beginning in each position and continuing down in the same column. It then adopts a greedy strategy of scanning uncovered rows in order. It picks the column that covers the most consecutive rows starting with the first row, and then repeats the same strategy from the topmost uncovered row.

Assume that *covers* is a temporary array of dimension equal to A, and initially, all $x_j = 0$.

```
CONSECUTIVE-ONES-SET-COVER:
    for (i ← m down to 1) do
        for (j ← 1 to n) do
            if (mat[i, j] = 0) then covers[i, j] ← 0
            else if (i = m) then covers[i, j] ← 1
            else covers[i, j] ← covers[i + 1, j] + 1
    i ← 0
    while (i ≤ m) do
        Scan row covers[i] for the max entry covers[i, j].
        x_j ← 1
        i ← i + covers[i, j]
```

At the completion of this algorithm a minimal sized cover has been found, in time $O(mn)$ which is linear in the size of the input. This simple algorithm does not work for weighted set cover. Hoffman [6] identifies a broader range of cases (including weights) for which a greedy algorithm does find the optimal solution to a combinatorial problem in the integer program formulation.

2.3 Bitonic Columns

One natural generalization of the consecutive ones property of a matrix is what we call the *mountain property*: for some permutation of the rows the values in every column are non-decreasing to some midpoint, then non-increasing thereafter (examining entries from top to bottom). The profile of each column looks like a mountain peak. More formally, a matrix has the mountain property if its rows can be permuted so that for each column j there exists an index i such that

$$a_{1j} \leq a_{2j} \leq \ldots \leq \quad a_{ij} \quad \geq a_{i+1,j} \geq \ldots \geq a_{mj}. \tag{5}$$

The midpoint index i can be different for each column. A symmetric property is the *valley property* in which the values in all columns decrease to a midpoint, then increase on the

other side (in some row permutation). Since in each case the values in a column must first monotonically progress in one direction, then monotonically progress in the opposite direction, we refer to these properties collectively as *bitonic column* properties.

One could further generalize the bitonic column concept to allow for a combination of mountain and valley columns in the same matrix, but in this paper we only make use of homogeneous bitonic column properties, where every column is a mountain, or every column is a valley.

A bitonic column property is significant because when the matrix is in conforming permutation we are guaranteed that each column has no non-adjacent local maxima or minima. Thus, the property is somewhat analogous to the concept of convexity, and similarly leads to efficient algorithms.

In solving a minimax program it is more convenient, for computational purposes, to deal with a *cost matrix* W where $w_{ij} = 1/a_{ij}$. For example, the cost matrix corresponding to the constraint matrix in (4) is

$$\begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ 3 & 2 & \infty & \infty & 3 & 8 \\ 4 & \infty & 3 & \infty & 4 & 4 \\ 8 & \infty & \infty & 4 & 10 & 4. \end{array} \tag{6}$$

It is easily seen that when the constraint matrix has the mountain property, the corresponding cost matrix has the valley property.

In the next section we present a fast algorithm for solving minimax programs when the cost matrix has the valley property. Its time complexity is linear when the number of columns is at least as large as the number of rows. It is conjectured that bitonic column properties imply efficient solution techniques for other problems as well.

2.4 Solving Minimax Programs with Valley Costs

If the cost matrix W corresponding to a minimax program has the valley property, then its rows can be permuted so that within any column the k smallest entries are all adjacent, for any k. This situation is amenable to solution by an efficient algorithm, described below. Section 3 gives an $O(n\text{SORT}(m)+mn)$ algorithm for finding such a permutation, if one exists. Here we assume that the input is presented as a cost matrix in conforming permutation.

We present the algorithm as constructing a network and then solving for the shortest path between two distinguished vertices.

Recall that a minimax program in standard form (3) has been normalized so that all RHS coefficients b_i and all objective function coefficients c_j are 1. The algorithm takes as input a cost matrix obtained from the constraint matrix by the identity $w_{ij} = 1/a_{ij}$. We assume that the rows of the cost matrix are in a permutation where the valley property holds. If necessary, the algorithm given in Section 3 can be used as preprocessing to obtain a conforming permutation.

The present algorithm begins by constructing a network data structure consisting of $m+1$ vertices, indexed $0, 1, \ldots, m$. Vertices will be joined by weighted, directed arcs from vertices of smaller index to vertices of larger index.

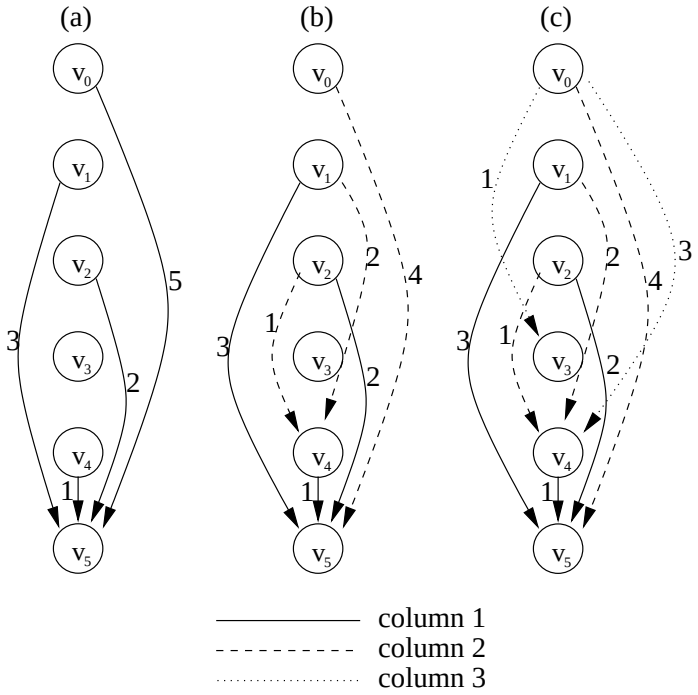
Each column can potentially contribute m arcs. An arc from v_i to v_j indicates the cost of "covering" rows $i+1$ to j (*i.e.* satisfying the constraints imposed by those rows) by the associated column variable. When two or more columns would contribute an arc joining the same pair of vertices, we select only the best arc (*i.e.* the arc of minimum cost).

The network arcs are constructed from the matrix entries by repeating the following process for each column, taking the columns in any order. (An illustrative example follows.) First we scan down the column to discover the minimum value. In case of ties, any minimum value is fine. From that midpoint we then start two pointers scanning in opposite directions, towards the top and bottom of the column. We iteratively move these pointers farther apart so as to discover successively higher cost coefficients in increasing order. Each new higher value identifies a span of rows (between the two pointers) that is covered by setting the column variable to that cost. As each successively larger span is discovered, we update the network with an arc from the vertex before the first covered row, to the vertex corresponding to the last covered row, whose weight is the covering cost. If an arc (i,k) already exists, we set its cost to the minimum of its existing cost and the covering cost represented by the current column. In addition, we annotate each arc with the index of the column variable whose covering capability it represents. Clearly, we consider the addition of $O(mn)$ arcs to the network, and no more than $O(m^2)$ will exist at any time, including after all columns have been processed.

An algorithmic presentation of this procedure is given as BUILD-NETWORK. We assume that the network is represented as an upper triangle adjacency matrix, so existence of an arc between two vertices can be checked in constant time.

BUILD-NETWORK:
Begin with square matrices w and *column* indexed for pairs of
vertices in $v_0, \ldots, v_m$ initialized $w(i,j) \leftarrow \infty$
and an $m \times n$ cost matrix W.
for $(1 \le j \le n)$ **do**
  Scan $W[,j]$ from the top to find the minimum entry $W[z,j]$.
  Set $min\_entry \leftarrow W[z,j]$; $top \leftarrow z$; $bot \leftarrow z$.
  while $(top > 1$ **or** $bot < m)$ **do**
    Decrement top and increment bot until $W[top,j]$
      and $W[bot,j]$ are the farthest apart two column entries
      that are each $\ge min\_entry$ and
      $\le$ (the smallest column entry $> min\_entry$).
      {In the first iteration, both $W[top,j], W[bot,j]$
      must be equal to min_entry. At any time,
      if either top or bot reaches 1 or m, then only increment
      or decrement the other.}
    The arc $(top-1, bot)$ is a potential arc: update $w(top-1, bot)$
      to the minimum of its existing weight and $W[z,j]$.
    If we updated the arc weight, also set
      $column(top-1, bot) \leftarrow j$.
    Set $min\_entry \leftarrow \max(W[top,j], W[bot,j])$.

Figure 1: Initial row cost network



An example of applying this procedure to the following cost matrix W is shown in Figure 1.

$$W = \begin{matrix} 5 & 4 & 1 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 4 & 4 \end{matrix}$$

Only arcs with sub-infinite cost are shown. Figure 1(a) shows the result after processing the first column; sub-figures (b) and (c) show the result after processing the second and third columns.

Lemma 3 *When the constraint matrix of a standard form minimax program has the valley property, the network constructed by* BUILD-NETWORK *contains an arc* (i, j) *with sub-infinite weight and* $column(i, j) = k$ *iff the constraints of all rows* i *through* j *can be satisfied by setting* $x_k = w(i, j)$*, no lesser value of* x_k *will satisfy all these same constraints, and no other row constraints are so satisfied.*

Once the row cost network has been constructed, discovery of the optimal solution to the original problem essentially corresponds to finding a shortest path from v_0 to v_m, interpreting the arc weights as distances. In general this can be done in $O(V^2)$ or $O(E \log V)$ time by Dijkstra's algorithm. For a DAG with a single source, such as our cost network, there is a well

known algorithm (as described in [2]) that merely iterates over the vertices in topologically sorted order ($v_0, v_1, \ldots, v_{m-1}$ for our network) relaxing each outgoing arc once, and requires only $O(E)$ time. (*Relaxation* of an arc (i, j) involves conditionally updating the distance of the terminal vertex, $d(j)$, to $d(i) + w(i, j)$ if that operation decreases $d(j)$.)

We will assume use of the DAG algorithm to find the shortest path, however it only finds paths consisting of explicitly represented arcs, and the actual solution to the problem may involve **implicit** arcs of the network just constructed, because our interpretation of the network is that each arc corresponds to satisfying one or more consecutive row constraints by a single x_j. For example, in Figure 1(c) there is an arc with weight 1 from v_0 to v_3 indicating that assigning $x_3 = 1$ covers the first three rows; implicitly there should also be weight 1 arcs connecting all pairs of vertices in $\{v_0, v_1, v_2, v_3\}$ of increasing index. The shortest path explicitly represented in the network is $(v_0, v_4), (v_4, v_5)$ (among others) with distance 4, but implicitly there is a covering path of distance 3 corresponding to arcs $(v_0, v_3), (v_2, v_5)$.

The number of implicit arcs corresponding to each explicit arc (i, j) is $O((j-i)^2)$, therefore naively inserting a representation for each one into the network during the processing of each column in Build-Network could be very expensive. Instead, we update the result of Build-Network to explicitly represent only some implicit arcs, leaving out those which we can prove will never be a necessary component of a shortest path. This modification can be done in $O(m^2)$ time.

Every explicit arc (i, j) has implicit arcs (i', j'), where $i \leq i' < j' \leq j$ and $w(i', j') = w(i, j)$. We say an arc (i', j') is **dominated** if there exists an explicit arc (h, k) where $h < i'$, $k \geq j'$ and $w(h, k) \leq w(i, j')$. Observe that (i, j) dominates its own implicit arcs (i', j') where $i < i'$, but not those where $i = i'$. An explicit arc may also be dominated by another.

Lemma 4 *When all implicit arcs are included in the network, there exists a shortest path from v_0 to v_m that does not utilize any dominated arcs.*

Proof. It is well known that every prefix of a shortest path is itself a shortest path (c.f. Floyd-Warshall all-pairs shortest path algorithm), so we only need to show that there exists a shortest path to any intermediate vertex on the shortest path from v_0 to v_m that does not utilize any dominated arc. We argue by induction on the length of a path prefix.

By the definition above, arc $(0, j)$ is never dominated, so the first arc in the shortest path cannot be dominated. Consider the shortest path prefix $e_1, e_2, \ldots, e_k$ from v_0 to $v_{j'}$, where $e_k = (i', j')$. By inductive hypothesis, none of the first $k-1$ arcs are dominated. Suppose e_k is dominated. Then there exists some (h, l) where $h < i'$ and $j' \leq l$ and $w(h, l) \leq w(i', j')$.

Case 1: There exists an arc $e_g = (h, l')$ in the path prefix, where $l' < j'$. Then the sequence of arcs $e_g, \ldots, e_k$ can be replaced by the single arc (h, j') which is not dominated by (h, l), and which does not increase the length of the path, since $w(h, j') \leq w(i', j')$.

Case 2: There exists an arc $e_g = (a, b)$ in the path prefix, where $a < h$ and $b > h$. Then there also must exist an implicit or explicit arc (a, h) where $w(a, h) \leq w(a, b)$. Hence there must exist a shortest path from vertex 0 to vertex h of no more than $k - 1$ arcs, with total length no more than the length of the path prefix $e_1, e_2, \ldots, e_g$. This path can be extended, by addition of arc (h, j'), which is not dominated by (h, l), into a path to vertex j' of no greater length than $e_1, e_2, \ldots, e_k$. ∎

The cost network constructed by BUILD-NETWORK can be updated to contain all non-dominated arcs in $O(m^2)$ time by the algorithm below. The key observation is that if for each v_i we consider all possible arcs (i, j) where $i < j$ in order of decreasing j, we can update their weights to *delete all dominated explicit arcs and add all non-dominated implicit arcs* among them, in $O(m - i)$ time. Every new arc gets the column label of the corresponding explicit arc.

ADD-ARCS:
Operates on cost network w generated by BUILD-NETWORK.
Assume that $w(i, j) = \infty$ if not explicitly set by that procedure.
for i from $m - 1$ down to 0 **do**
$x \leftarrow w(i, m)$
$col \leftarrow column(i, m)$
for j from m down to $i + 1$ **do**
If $w(i, j) > x$ then $w(i, j) \leftarrow x$ and $column(i, j) \leftarrow col$
If $x > w(i, j)$ then $x \leftarrow w(i, j)$ and $col \leftarrow column(i, j)$

Lemma 5 *The procedure* ADD-ARCS *updates the cost network so that all non-dominated implicit arcs are explicitly represented. Only implicit arcs of existing explicit arcs are added.*

Proof. For each value of i in the outer loop, x gets its first sub-infinite value from a pre-existing explicit arc. Subsequently, an arc (i, j') is added (i.e. has a new weight assigned) only as an implicit arc of an existing explicit arc from which x has been set.

To see that all non-dominated implicit arcs are added, consider an arbitrary implicit arc (i, j') of some explicit arc (i, j). Since (i, j') is not dominated, neither is (i, j), so as arcs out of v_i are processed by the procedure, x will be set to $w(i, j)$ as that arc is considered. Neither can any arc (i, k) exist, where $j' < k < j$ and $w(i, k) < w(i, j)$, otherwise (i, j') would be dominated, so x will still equal $w(i, j)$ as the procedure considers arc (i, j') and updates its value to x. ∎

With this post-processing of the cost network, an optimal solution to the original minimax program can be discovered by the shortest-path algorithm for a single-source DAG.

Theorem 6 *If the cost matrix of a standard form minimax program has the valley property, and $e_1, e_2, \ldots, e_k$ are the arcs constituting the shortest path from v_0 to v_m in the cost network constructed by* BUILD-NETWORK *and updated by* ADD-ARCS*, then*

$$x_j = \begin{cases} w(e_i) \text{ if } x_j = column(e_i) \text{ for some } e_i \text{ in the path} \\ 0 \text{ otherwise} \end{cases}$$

is an optimal solution to the minimax program.

Proof. If the minimax program constraint matrix has the valley property, then it can be put in a permutation where any setting of some column variable that satisfies some row constraint necessarily satisfies the constraints of a consecutive group of one or more rows. In this permutation (which is the input to BUILD-NETWORK), any feasible solution covers consecutive rows with each column variable that covers any row, and hence any optimal solution has the same property. Moreover, we can uniquely associate each row to a covering variable in any optimal solution, such that all rows associated with a variable are consecutive.

By Lemma 3 the network constructed by BUILD-NETWORK contains all the arcs corresponding to maximal consecutive groups of rows that can be covered by a minimal setting of each x_j. By the definition of implicit arcs it follows that adding implicit arcs to the network results in an implicit arc for every consecutive group of rows that can be covered by one column variable, at minimal cost. Hence, for every possible way of covering a consecutive group of rows by one variable, there is an explicit or implicit arc e associated with that variable where $w(e)$ is the least possible value of the variable covering all rows in the group. Each such arc is directed from the vertex of lower index to the vertex of higher index. Consequently, for any complete and exhaustive partition of rows into consecutive groups that can be satisfied by some variable assignment, there is a path from v_0 to v_m, and the least cost feasible assignment corresponds to the shortest such path.

By Lemma 4 it follows that dominated arcs need not be examined in order to find a shortest path in the network, and by Lemma 5 we know that ADD-ARCS adds all non-dominated arcs to the network constructed by BUILD-NETWORK. Therefore, the shortest-path in the resulting network (which can be found by the standard algorithm for DAGs) corresponds to the optimal solution to the minimax program, in the manner described by the statement of this theorem. ∎

2.5 Complexity

BUILD-NETWORK does $O(m)$ work for each column, since it scans each column twice and does no more than a constant amount of work for each entry. The network it constructs has $m+1$ vertices and $O(\min(m^2, mn))$ sub-infinite arcs. Including data structure initialization, BUILD-NETWORK runs in $O(mn+m^2)$ time and uses $O(m^2)$ space. ADD-ARC does constant work for each entry of the matrices representing the network, consuming $O(m^2)$ time. The shortest-path algorithm used to find the least-weight sequence of arcs connecting v_0 to v_m requires $O(m^2)$ time. The complete algorithm runs in $O(mn + m^2)$ time.

3 Testing for Bitonic Columns

The mountain and valley properties of matrix columns were defined in Section 2.3. In this part we present an efficient algorithm to identify whether a matrix has either or these properties, and if so, a permutation that puts it into a conforming configuration. The algorithm is an extension of the original PQ-tree reduction algorithm [1].

Figure 2: P-node

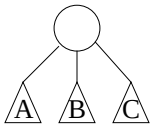
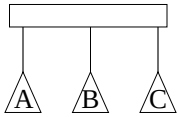


Figure 3: Q-node



3.1 PQ-trees

Booth and Lueker [1] showed how to test for the consecutive ones property (and related properties such as being an interval graph) in linear time with a data structure called a PQ-tree. A PQ-tree is capable of efficiently representing all permissible permutations of a set subject to constraints in the form of subsets which must occur as consecutive subsequences.

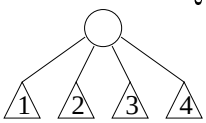
A PQ-tree consists of three kinds of nodes: P-nodes, Q-nodes and leaves. A P-node has two or more children and represents any permutation of those children. For example, Figure 2 shows a P-node with three children: the leaves A, B, C. It represents all 3! possible permutations of those leaves. (P-nodes are drawn as circles.) A Q node has three or more children and represents either of the two permutations that preserve the linear arrangement of the children. Figure 3 shows a Q-node with three children, the leaves A,B,C. It represents the permutations ABC and CBA. (Q-nodes are drawn as rectangles.)

Booth and Lueker presented a basic algorithm for processing PQ-tree constraints that takes two arguments, a PQ-tree T and a set $S \subset U$ (U is the complete set of leaves), and returns a modified tree $\overline{T}$ representing the subset of permutations represented by T in which the elements of S occur consecutively. If no such permutations exist, $\overline{T}$ will be the empty tree representing no permutations. By starting with an initial tree consisting of a single P-node with all leaves as children, and iteratively applying the algorithm to members of a set $\mathcal{S}$ of sets S_i of leaves, a tree representing all permutations in which all members of each set S_i are consecutive can be obtained. Booth and Lueker prove that their algorithm runs in time $O(m + n + SIZE(\mathcal{S}))$ where $m = |U|$, $n = |\mathcal{S}|$, and $SIZE(\mathcal{S}) = \sum |S_i|$. (We follow the convention that m denotes the number of rows and n the number of columns in the matrix.)

Section 3.4 presents a modified PQ-tree algorithm for efficiently testing for a bitonic column property. The algorithm is not described in its entirety, but as a set of modifications to the algorithm of Booth and Lueker. In order to fully comprehend the algorithm, such as for implementation, their paper should be consulted. However, the remainder of this section gives a general outline of their algorithm together with a small example so that the modifications and proofs described in subsequent sections can be understood at a high level.

The basic PQ-tree reduction algorithm consists of two phases: *bubble-up* and *reduce*. The

Figure 4: Initial PQ-tree



bubble-up phase identifies the *pruned pertinent subtree* containing S. The pertinent subtree with respect to S is the minimal subtree of T whose leaves contain all of S; the pruned pertinent subtree is generated by removing all subtrees not containing any member of S. The procedure BUBBLE starts with the leaves in S and ascends T, marking each node with the number of children that must be processed before it in the reduction phase. In the reduction phase the subtrees of the pruned pertinent subtree are matched against a small fixed set of templates and transformed according to the first one that matches, bottom up. The procedure REDUCE begins by putting all leaves in S on a queue, then processes the queue in FIFO order, placing the parents of transformed nodes on the end of the queue when all of their pertinent children have been reduced, until the entire pruned pertinent subtree has been handled. If at any point there is no applicable template, then the tree is transformed to NIL, since there are no permutations meeting all the constraints.

3.1.1 Consecutive Ones Example

Suppose we are given the $(0,1)$ matrix

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

and wish to determine whether it has the consecutive ones property (row indices are shown for convenience). Using PQ-trees we begin by forming the initial tree shown in Figure 4 with a single P-node and a leaf for each row index. Then we reduce the tree by the set of all row indices corresponding to ones, for each column.

The reduction set of column 1 is $S_1 = \{3, 4\}$. Figure 5 illustrates the reduction. The P-node labeled a has two pertinent children, the leaves whose labels are members of S. Those leaves are shown shaded, to indicate they are "full". The applicable reduction is to create a new P-node, labeled b, unifying all and only the full children of a. The resulting node b is full because all of its children are full (*i.e.* all of the leaves in its subtree are in S).

The reduction set of column 2 is $S_2 = \{1, 4\}$. Figure 6 illustrates the reduction. Initially, leaves 1 and 4 are full, node b has one pertinent child, and node a has two pertinent children. Node b is temporarily reduced to a 2-child Q-node c (during reduction the 3-child minimum can be violated in an intermediate result), then when node a is reduced its full children are moved onto the full end of its partial Q-node child. (A "partial" node is a Q-node with some full and some empty children.)

Figure 5: Column 1 reduction

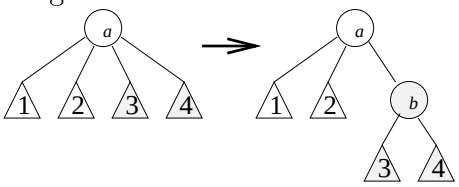
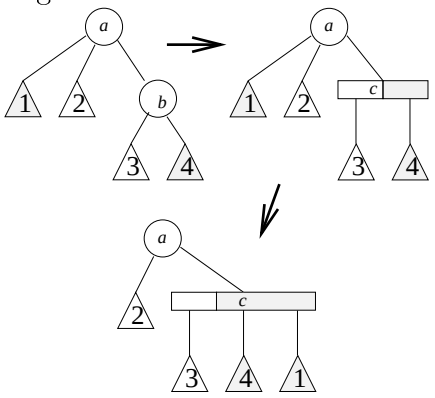


Figure 6: Column 2 reduction



The reduction set of column 3 is $S_3 = \{2, 3\}$. Figure 7 illustrates the reduction. Initially, leaves 2 and 3 are full, node c has one pertinent child, and node a has two pertinent children. The reduction of c is just to mark it as partial. The reduction of a is to move its full child onto the full end of c, then promote c to root, since a is no longer necessary.

The final tree shows that two satisfying row permutations of the matrix exist: $2, 3, 4, 1$ and $1, 4, 3, 2$. If at any time during the process there was no applicable reduction for a node in the pruned pertinent subtree, then the tree would have reduced to NIL, and we would know that the matrix does not have the consecutive ones property.

3.2 Expressing a Bitonic Constraint in a PQ-tree

A single P-node with all leaves as children represents all possible permutations. The constraint that a leaf subset $S \subset U$ must appear consecutively is easily expressed with a single additional P-node as illustrated in Figure 8. A bitonic constraint can be represented similarly, using an unbalanced tree of P-nodes. For example, suppose that a column contains the set of values $\{1, 2, 3, 3, 4, 5, 5, 5\}$, then all permutations of those values with the bitonic valley property can be expressed by the tree shown in Figure 9. Basically, the children of each node are the leaves representing all members of an equal-value set together with the P-node whose children are in the immediately preceding set. The only exception is that, since a P-node must have at least two children, if there is only one member of the first (according to the "$\leq$" relation) equal-value set, then the bottom-most P-node has the members of the

Figure 7: Column 3 reduction

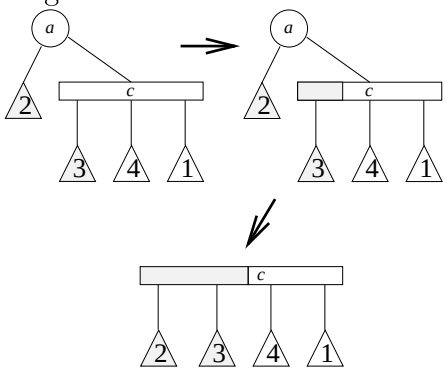


Figure 8: Consecutive subset schema

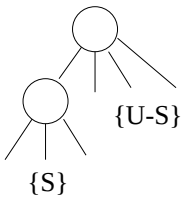
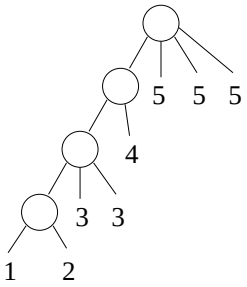


Figure 9: Bitonic constraint by P-nodes



first two equal-value sets as its children. (In the rest of the paper, in such a situation it is assumed that the first two equal-value sets are merged.)

Lemma 7 *A matrix has a bitonic column property just in case there exists a permutation of its rows in which for every column, given a partitioning of its elements into equal-value sets $E_1, \ldots, E_k$ of increasing or decreasing value, all members of $\bigcup_{j=1}^{i} E_j$ are consecutive, for $1 \leq i \leq k$.*

Proof. Suppose such a permutation exists; then column entries are monotonically decreasing (increasing) from the outermost equal-value set to the center of the innermost. Suppose the bitonic valley (mountain) property holds; then we can identify nested increasing (decreasing) equal-value sets in each column. ∎

3.3 A Naive Algorithm

A matrix can be tested for the bitonic valley property by using the original PQ-tree algorithm of Booth and Lueker as follows:

1. Start with a tree T consisting of a single P-node with all the row indices as leaves.
2. For each column do the following:
   (a) Sort the elements into ascending equal-value sets $E_1, ..., E_k$.
   (b) For i from 1 to k, let S be the row indices of elements in $E_1 \cup \ldots \cup E_i$; perform the bubble-up and reduce phases to obtain $\overline{T}$.
3. If the final $\overline{T}$ is not NIL, then the bitonic valley property holds and $\overline{T}$ encodes the row permutation(s) of interest.

Naively applying the Booth and Lueker algorithm in this way results in a time complexity of $O(mn \log m + m^2 n)$. The first term, $(mn \log m)$, is the cost of sorting each column to find the equal-value sets. The second term is the cost of running their algorithm $O(m)$ times on each column, with increasingly larger sets S, so that the sum of all input set sizes is $SIZE(\mathcal{S}) = O(m^2 n)$.

The next section presents an extension of their algorithm that preserves its linear time complexity when repeatedly applied to equal-value sets within a column. The resulting time complexity is

$$O((n\text{SORT}(m) + mn)$$

where SORT(m) is the time to sort m elements in one column. When a linear time sorting method is applicable, bitonic column testing can be performed in linear time.

3.4 The Extended Algorithm

Suppose that the equal-value sets of elements in each column have already been identified, and let us focus on the work done to process a single column. The results of work done so far has been expressed in a PQ-tree T whose leaves hold the row indices of the matrix. In the original algorithm of Booth and Lueker, the task of the algorithm is to ensure that the leaves in S (the set of all rows with a 1 in this column) are adjacent in all permutations represented by T. The extended algorithm must ensure that all leaves in $\bigcup_{j=1}^{i} E_j$ are consecutive in all permutations, for $1 \leq i \leq k$ where $E_1, \ldots, E_k$ are the disjoint equal-value sets for the column. Note that $\sum_{j=1}^{k} |E_j| = m$.

The original algorithm processes each column with one invocation of BUBBLE, followed by one invocation of REDUCE. The essential idea of the extended algorithm is to break a single iteration of this sort into k BUBBLE/REDUCE passes, and preserve the linear running time by ensuring that no pass does more than a constant amount of work in a part of the PQ-tree that has been processed by an earlier pass. That this is possible is suggested by the following observations.

(1) Reduction does not alter any node that is not part of the pruned pertinent subtree.

(2) If T has already been reduced with respect to S, then reduction of T with respect to $S' \supset S$ does not change more than one node in the pruned pertinent subtree of T with respect to S.

The truth of (1) is clear from inspection of the algorithm. Observation (2) will be proven as Lemma 8 in Section 3.6.

Although the basic idea is simple, the details of data structure maintenance necessary for implementation are somewhat involved. The rest of this section outlines the modifications necessary, giving more details on Booth and Lueker's basic BUBBLE/ REDUCE iteration as appropriate. Afterwards, a concrete example is given in illustration.

Last Subtree Root Each column is processed by k passes through BUBBLE/ REDUCE for the k equal-value sets. At the end of each pass, REDUCE saves a pointer to the result of the last node reduction. That node is denoted R, and it is the root of the pertinent subtree with respect to all equal-value sets processed so far. Both BUBBLE and REDUCE use R during their initialization to quickly recover a state as though they had just finished reducing all previous equal-value sets in a single pass, then continue with the new members of U added in this pass. (R is initially null, at the beginning of the first pass.) Hence, the parameters to BUBBLE and REDUCE are (T, R, E_i), rather than (T, S).

Persistence of State The original algorithm annotates the tree with *marks*, *labels* and *counters* during BUBBLE and REDUCE, then reinitializes them all before the next column iteration. With a very few exceptions, these annotations are all in the pruned pertinent subtree. In the extended algorithm, additional nodes are added to the pruned pertinent subtree with each pass within a column. With a very few exceptions, annotations are preserved between passes and only erased when the entire column has been processed. However, only

Figure 10: Pseudo-node application

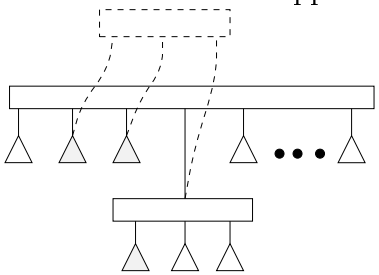
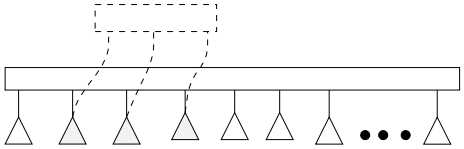


Figure 11: Reduction result



the annotations on R are actually examined in subsequent passes. Consequently, the amount of work involved in making and erasing all of these annotations is approximately the same as if they were all done in a single pass.

Limited Overlap At the beginning of pass $i+1$, one of two situations can obtain: either some leaf in E_{i+1} is a descendent of R, or none is. The first case can only arise if R is not labeled full (*i.e.* not every leaf below R is in $\bigcup^i E_j$). If R is a P-node, then a property of the algorithm (following from case analysis of the reduction templates) is that R is either full or has exactly one non-empty (*i.e.* none of its leaves are in $\bigcup^i E_j$) child. If R is a Q-node, it can only be full or partial (children on one side are full and the rest are empty). If the last node reduced in pass i is a non-full P-node, then we set R not to that node, but to its only non-empty child, which can be only a Q-node or a full P-node. The result is that R is always the least ancestor of all leaves in $\bigcup^i E_j$, and that some leaf in E_{i+1} can lie below R only if R is a partial Q-node.

Pseudo-Nodes and Parent Pointers An exception to the foregoing is that R can sometimes be a pseudo-node, a temporary node type used to attain linear running time of the original algorithm. In that algorithm, it can happen that a leaf set S might lie entirely below internal (*i.e.* not endmost) children of some Q-node, and still have a consecutive permutation. Figure 10 illustrates such a situation, with the leaves in S shown shaded. In this case the desired reduction is to raise and merge the child Q-node into its parent, with the result shown in Figure 11. For greater efficiency, the original algorithm does not guarantee correctness of the parent pointer of a child of a Q-node unless that child is an endmost child. Children of Q-nodes are always correctly linked to their immediate siblings,

then the ends of the sibling chain are linked to the parent. Repeatedly finding the root of the pertinent subtree in cases like that illustrated in Figure 10 could violate linear time constraints by scanning too many siblings. And the parent is not really necessary, since the reduction can be performed just by adjusting sibling pointers. So, the original algorithm uses a pseudo-node, shown in dotted lines, which shadows the real Q-node parent. The relevant subsequence of Q-node children is temporarily linked to the pseudo-node for the duration of one REDUCE invocation. The extended algorithm uses pseudo-nodes in the same situations, but must consequently be able to update correctly the annotations of the real Q-node parent if a pseudo-node is used in any pass but the last (since then R will be that pseudo-node).

Initialization of BUBBLE As explained in the previous paragraph, nodes in a PQ-tree do not always have valid parent pointers, but parent pointers must be correct in the pruned pertinent subtree in order to do bottom-up reduction. Therefore, a role of BUBBLE is to establish correct parent pointers in the pruned pertinent subtree that it discovers. The routine BUBBLE uses the marks *blocked* and *unblocked* to denote that a node is waiting for or has, respectively, a valid parent pointer.

In the first pass R is null, and BUBBLE behaves as in the ordinary algorithm. Otherwise, special initialization is performed according to the value of R.

1. If R is a pseudo-node, then it anchors a sequence of full nodes not adjacent to either endmost child of some Q-node. These nodes are all blocked (in essence; the pseudo-node reduction function makes sure that at least the ends are so marked) so BUBBLE is initialized with block-count = 1.

2. If R is an empty P-node, we reset R to be its only non-empty child, as described previously.

3. If R is a non-empty, non-pseudo-node, we check its parent type and if R turns out to be a non-endmost child of a Q-node we mark it blocked and initialize a block-count of 1.

4. Finally, if R is not a pseudo-node or a non-endmost child of a Q-node, we mark R unblocked, initialize it and its parent with a pertinent-child-count of 1, and queue the parent.

Then, all leaves in E_i are enqueued, and processing of the queue begins.

Unblocking in BUBBLE Whenever a previously blocked node is unblocked by its sibling (an endmost child with a valid parent pointer was discovered), we must check to see if R is a pseudo-node and the node is linked to it. If so, we don't proceed down the line of siblings as usual, but jump over all nodes remaining blocked from the last pass by following pointers threaded through R. At this time, accumulated annotations on R are used to update the real Q-node parent with any necessary annotations, and R is reset to be that Q-node, for use in REDUCE.

Termination of BUBBLE The termination conditions of the extended algorithm are the same as those for the original algorithm. If a pseudo-node is needed, and R is a pseudo-node, then it is re-used, and existing pointers are used to avoid traversing blocked children remaining from the last pass. Upon termination of BUBBLE, all nodes in the new pruned pertinent subtree have valid parent pointers and pertinent-child counters. An exception is that R's pertinent-child counter is one greater than its number of pertinent children. In essence, R will be treated as an additional leaf by REDUCE.

Modification to REDUCE When R is null, REDUCE behaves as in the original algorithm. Otherwise, it begins by decrementing the pertinent-child count of R.

1. If the result is non-zero, then some part of E_i lies below R. R will be queued for reduction when that part of E_i has been reduced, so nothing more is done with it now.

2. If the result is zero, then no part of E_i lies below R, and R does not need to be reduced again. We then decrement the pertinent-child counter of R's parent, and are done with R.

Then all of the leaves in E_i are queued and queue processing begins. Termination of REDUCE is the same as in the original algorithm, except that the last node to be reduced is identified by pertinent-leaf-count reaching $|E_i| + 1$, and we set R to the result of its reduction.

Modification to Templates Exactly the same templates and reduction transformations are used. However, in the original algorithm the time complexity of template testing and application is directly proportional to the number of full children of a node. In the extended algorithm we need to avoid paying again for nodes reduced in previous passes, so it is necessary for transformations to add some additional state (a few extra pointers) to Q-nodes and pseudo-nodes that enables template testing to skip over runs of adjacent children that were reduced in a previous pass.

3.5 An Example

The following example illustrates the behavior of the extended algorithm in a simple case.

Suppose we want to test the following 8×2 matrix for the bitonic valley property.

$$\begin{array}{cc} \textit{1} & \left[\begin{array}{cc} 1 & 3 \end{array}\right. \\ \textit{2} & \left.\begin{array}{cc} 2 & 2 \end{array}\right. \\ \textit{3} & \left.\begin{array}{cc} 2 & 3 \end{array}\right. \\ \textit{4} & \left.\begin{array}{cc} 3 & 3 \end{array}\right. \\ \textit{5} & \left.\begin{array}{cc} 3 & 3 \end{array}\right. \\ \textit{6} & \left.\begin{array}{cc} 2 & 2 \end{array}\right. \\ \textit{7} & \left.\begin{array}{cc} 1 & 1 \end{array}\right. \\ \textit{8} & \left.\begin{array}{cc} 1 & 1 \end{array}\right] \end{array}$$

The initial tree T is shown in Figure 12; the leaves hold the matrix row indices. The equal-value sets of column one are

$$E_1 = \{1, 7, 8\}, E_2 = \{2, 3, 6\}, E_3 = \{4, 5\}.$$

Figure 12: Initial tree

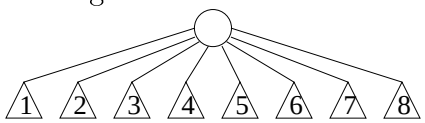
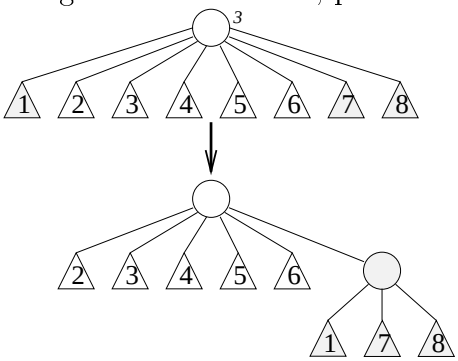


Figure 13: Column 1, pass 1



Figures 13, 14, 15 show the three passes of reduction by these equal-value sets.

The nodes labeled full in each pass are shown lightly shaded, and the number of pertinent children of each internal node is shown in italics, next to the node. The first tree in each figure shows the state after BUBBLE and at the beginning of REDUCE. Subsequent trees show the state as REDUCE works from bottom up. For example, in Figure 13 the leaves $1, 7, 8$ are in the equal-value set, so they are shaded in the first tree, and the root node is annotated with a *3*. The second tree, the result of the reduction, has an additional shaded (full) P-node that has become the parent of those three leaves.

Full nodes that persist from former passes are shown in darker shading, and the last pertinent subtree root is marked R. In Figure 14 the leaves $1, 7, 8$ and their parent P-node are darker, while the members of the new equal-value set $\{2, 3, 6\}$ are lightly shaded. R counts as a pertinent child, so the root P-node is annotated *4*.

The final result of column 1 reduction, shown in Figure 15, is the unbalanced tree representing the nesting of its equal-value sets, in accordance with Lemma 7.

The equal-value sets of column 2 are $\{7, 8\}, \{2, 6\}, \{1, 3, 4, 5\}$. Figures 16,17,18 show the subsequent reduction of the tree by those sets.

The final result shows that a number of row permutations, including the one shown below,

Figure 14: Column 1, pass 2

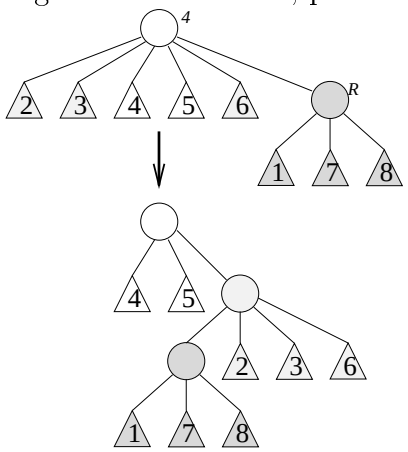


Figure 15: Column 1, pass 3

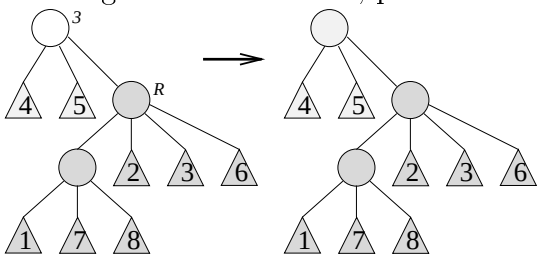


Figure 16: Column 2, pass 1

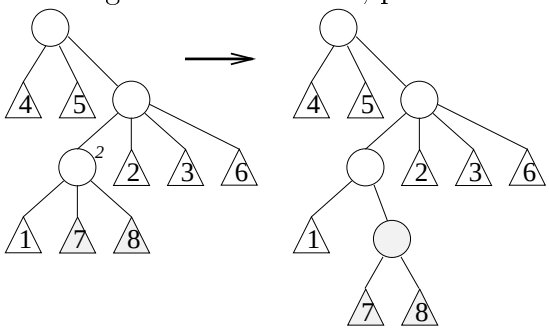


Figure 17: Column 2, pass 2

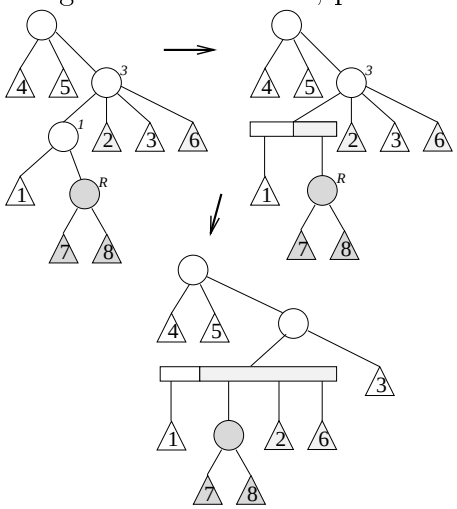
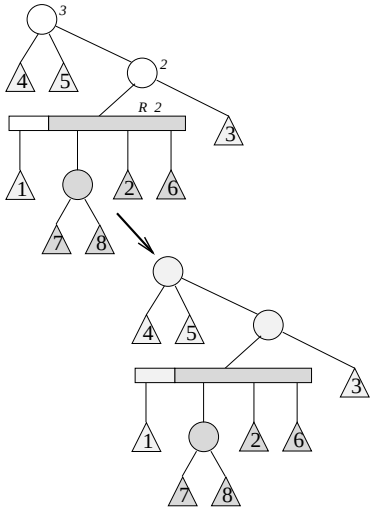


Figure 18: Column 2, pass 3



satisfy the bitonic valley constraint.

$$\begin{array}{cc} \textit{4} & \left[\begin{array}{cc} 3 & 3 \end{array}\right. \\ \textit{5} & \left|\begin{array}{cc} 3 & 3 \end{array}\right. \\ \textit{1} & \left|\begin{array}{cc} 1 & 3 \end{array}\right. \\ \textit{7} & \left|\begin{array}{cc} 1 & 1 \end{array}\right. \\ \textit{8} & \left|\begin{array}{cc} 1 & 1 \end{array}\right. \\ \textit{2} & \left|\begin{array}{cc} 2 & 2 \end{array}\right. \\ \textit{6} & \left|\begin{array}{cc} 2 & 2 \end{array}\right. \\ \textit{3} & \left.\begin{array}{cc} 3 & 3 \end{array}\right] \end{array}$$

3.6 Correctness and Complexity

The correctness of this algorithm for finding all permutations conforming to a bitonic column property follows from Lemma 7, the correctness of Booth and Lueker's original algorithm, and the truth of the assertion that the modifications improve efficiency of the naive procedure described in Section 3.3, but do not alter its semantics. Hopefully, the description of those modifications in the previous section together with the following lemma whose proof was delayed until this point make a persuasive case for that assertion.

Lemma 8 *If T has already been reduced successfully with respect to S, then successful reduction of T with respect to $S' \supset S$ does not change more than one node in the pruned pertinent subtree of T with respect to S.*

Proof. Since T has been reduced with respect to S, the elements of S are consecutive in all permutations represented by T. Therefore, the root of the pertinent subtree with respect to S must be either a P-node that is full with respect to S, or a Q-node whose children contain a single, consecutive subsequence of nodes that are full with respect to to S. By inspection of the reduction templates it can be seen that no template is applicable to a full node, hence no reduction action applies to any of the nodes that were full with respect to to S when reducing S'. The only node in the pruned pertinent subtree of S that can change is the root, if it is a partial Q-node (or a pseudo-node shadow). ∎

The proof of linear time complexity for reduction (not including equal-value set identification) is based on that of Booth and Lueker. It essentially involves showing that the extended algorithm has nearly the same complexity, despite performing k passes in place of 1 for each column. Their proof is built on the following lemmas.

Lemma 9 (Booth and Lueker Lemma 2) *The original bubbling up phase of reduction requires*

$$O(|PRUNED(T, S)|)$$

steps.

Lemma 10 (Booth and Lueker Lemma 3) *The original template-matching phase of reduction requires*

$$O(|PRUNED(T, S)|)$$

steps.

These lemmas extend naturally to the extended algorithm when it is observed that for each column we reduce T with respect to $E_1, \ldots, E_k$ in place of S, where the union of these disjoint sets is U. Since U is the set of all leaves of T, for each column we process the entire tree. (An easy but asymptotically insignificant improvement to the algorithm would be to skip the k'th pass for each column, since in that final pass all nodes become full and the tree topology does not change.)

Lemma 11 *The modified bubbling up and template matching phases of reduction each require $O(|T|)$ steps for all k passes put together.*

Proof. In accordance with Lemma 8, in pass $i+1$ the extended algorithm does not touch any node in the pruned pertinent subtree of $\bigcup^i E_j$ except the root R. When it does touch R it performs no more than a constant number of steps, plus possibly an additional amount $O(|E_{i+1}|)$ (when some part of E_{i+1} lies below R; that additional work falls into the cost accounted for that subtree). Otherwise, the processing of the pruned pertinent subtree with respect to E_{i+1} is extremely similar to, and has the same complexity as, that of the original algorithm. Since the pruned pertinent subtrees of T with respect to $E_1, \ldots, E_k$ exactly span T, intersecting only at the intermediate roots R, the extended algorithm performs $O(|T|)+O(k)$ steps. Since $k \leq m$, and T is a tree of m leaves, the $|T|$ component dominates. ∎

All that remains is to sum all the costs.

Theorem 12 *The extended algorithm requires $O(nSORT(m)+mn)$ steps to test for a bitonic column property in a matrix, where $SORT(m)$ is the number of steps required to discover the equal-value sets in a column.*

Proof. The costs for both the bubble-up and template matching phases of reduction are $O(|T|)$ for each column. Since there are no more than $O(m)$ nodes in the tree, once the equal-value sets have been identified the rest of the algorithm requires only $O(mn)$ steps. ∎

4 Application

We conclude by describing a real application whose description preceded our exploration of this topic, in order to illustrate how the minimax program formulation can naturally capture the structure of a practical optimization problem.

Huang's [7] work on software dependability measurement investigated a metric called trustability (T) that represents the degree of confidence that a program being tested is free of faults. If D represents the probability of detecting a fault by applying a particular stochastic test method, then after N error-free applications of that test method, the trustability of the program is

$$T = 1 - (1 - D)^N.$$

(We have slightly simplified Huang's formulae for this presentation.) More generally, suppose there are m fault classes and n test methods, then trustability after an uninterrupted series of successful tests is

$$T = 1 - \max_{1 \leq i \leq m} \{ \min_{1 \leq j \leq n} \{ (1 - D_{ij})^{N_j} \} \} \tag{7}$$

where D_{ij} is the probability that method j detects a fault in class i, and N_j is the number of times method j has been applied. Equation (7) has a max-min structure because the testing methods are assumed to be probabilistic, so their effects are independent, rather than additive.

An optimization problem that arises in this context is to minimize the amount of effort devoted to testing,

$$E = \sum_{i=1}^{n} c_j N_j$$

subject to the constraint that a minimal value of $T \geq T_0$ is attained. This problem can be converted to a minimax program of form (3) where $x_j = N_j$ and

$$a_{ij} = \frac{\log(1 - D_{ij})}{c_j \log(1 - T_0)}.$$

References

[1] Booth, K. S., and Lueker, G. S. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci. 13*, 3 (Dec. 1976), 335–379.

[2] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[3] Fulkerson, D. R., and Gross, O. A. Incidence matrices and interval graphs. *Pacific Journal of Mathematics 15* (1965), 835–855.

[4] Golumbic, M. C. Interval graphs and related topics. *Discrete Math 55* (1985), 113–121.

[5] Hochbaum, D. S. Approximating covering and packing problems: Set cover, vertex cover, independent set, and related problems. In *Approximation Algorithms for NP-Hard Problems*, D. S. Hochbaum, Ed. PWS, Boston, 1997, pp. 94–143.

[6] Hoffman, A. J. On simple combinatorial problems. *Discrete Mathematics 106/107* (1992), 285–289.

[7] Huang, Y. *Software Dependability Measurement during Testing.* PhD thesis, University of California, San Diego, La Jolla, CA, 1994.

[8] Keil, J. M. Finding Hamiltonian circuits in interval graphs. *Information Processing Letters 20* (1985), 201–206.

[9] Marathe, M. V., Ravi, R., and Rangan, C. P. Generalized vertex covering in interval graphs. *Discrete Applied Mathematics 39* (1992), 87–93.

[10] Rose, D. J., Tarjan, R. E., and Lueker, G. S. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing 5*, 2 (June 1976), 266–283.