Minimax Programs

T. C. Hu and P. A. Tucker*
Department of Computer Science and Engineering,
School of Engineering,
University of California, San Diego
La Jolla, CA 92093
`hu@cs.ucsd.edu` `ptucker@cs.ucsd.edu`

Technical Report CS97-547

June 16, 1997

Abstract

We introduce an optimization problem called a minimax program that is similar to a linear program, except that the addition operator is replaced in the constraint equations by the maximum operator. We clarify the relation of this problem to some better-known problems. We identify an interesting special case and present an efficient algorithm for its solution.

1 Introduction

Over the last fifty years, thousands of 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.

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

In this paper 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 with 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.

In this paper 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). A separate paper [7] introduces an efficient algorithm for recognition of matrices with this property.

The rest of this paper is organized as follows. Section 2 defines a minimax program. Section 3 describes its relation to other problems. Section 4 identifies an interesting and practical special case that can be solved quickly. Section 5 presents an efficient algorithm for that case. Section 6 describes a sample application.

2 A Minimax Program

In a linear program the task is to

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

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 more than 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.

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

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 [4], an application to be discussed Section 6.

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.

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

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

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

3 Relation to Other Problems

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

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

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

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

and eliminate the integer requirement, then the optimal solutions are unchanged. When the matrix is $(0,1)$ and the requirement vector is $\vec{1}$, using the maximum operator in the constraints has the same effect as the addition operator in combination with the integer requirement. Any feasible solution to the minimax program is feasible for the integer program, and vice versa. Consequently there is a straightforward reduction of any set cover problem, using the parameters of its integer program formulation, into a minimax program. It follows 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 reduction technique involves expanding each column of the minimax constraint matrix into a series of $(0,1)$ columns, one for each row in the original matrix, and assigning appropriate cost coefficients to the columns. In particular, for every constraint coefficient a_{ij} of a minimax program in standard form, the corresponding integer program contains a column j_i where

$$c_{j_i} = 1/a_{ij}$$

and

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

for all rows $1 \leq k \leq m$. The following simple example illustrates this reduction. The minimax program with unit costs and constraint inequalities

$$\max \left[\begin{array}{ccc} x_1 & x_2 & x_3 \\ 5 & 1 & 1 \\ 2 & 4 & 3 \\ 1 & 1 & 4 \end{array} \right] \quad \geq \quad \begin{array}{c} \\ 1 \\ 1 \\ 1 \end{array}$$

$$x_j \geq 0$$

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

$$
\text{sum} \left[\begin{array}{ccccccccc}
\frac{1}{5}x_{1_1} & \frac{1}{2}x_{1_2} & 1x_{1_3} & 1x_{2_1} & \frac{1}{4}x_{2_2} & 1x_{2_3} & 1x_{3_1} & \frac{1}{3}x_{3_2} & \frac{1}{4}x_{3_3} \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1
\end{array} \right] \begin{array}{c} \\ \\ \geq \\ \\ \end{array} \begin{array}{c} \\ 1 \\ 1 \\ 1 \end{array}
$$

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

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.

A minimax problem can be solved by expanding it into an integer program, then applying any solution technique appropriate to a set cover problem in that formulation. However, the expansion incurs a penalty by inflating the problem representation size by a factor of $O(m)$. 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.

4 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 [5] and vertex cover [6]).

The interval graph property is intimately related to the consecutive ones property of $(0, 1)$ matrices [2]. 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 [2] 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.

4.1 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 ($i \leftarrow m$ down to $1$) do
        for ($j \leftarrow 1$ to $n$) do
            if ($mat[i,j] = 0$) then $covers[i,j] \leftarrow 0$
            else if ($i = m$) then $covers[i,j] \leftarrow 1$
            else $covers[i,j] \leftarrow covers[i+1,j] + 1$
    $i \leftarrow 0$
    while ($i \leq m$) do
        Scan row $covers[i]$ for the max entry $covers[i,j]$.
        $x_j \leftarrow 1$
        $i \leftarrow i + covers[i,j]$
```

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 [3] identifies a broader range of cases (including weights) for which a greedy algorithm does find the optimal solution to a combinatoric problem in integer program formulation.

4.2 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 C where $c_{ij} = 1/a_{ij}$. 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. We conjecture that bitonic column properties imply efficient solution techniques for other problems as well.

5 Solving Minimax Programs with Valley Costs

If the cost matrix C 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. A companion paper [7] gives an $O(mn \log m)$ algorithm for finding such a permutation, if one exists. Here we assume that the input is presented as a cost matrix in conforming permutation.

The algorithm falls into the dynamic programming paradigm; consequently it can be conceptualized as iteratively operating on a table. We describe it as constructing and then operating on a network in order to illustrate its similarity to a shortest-path algorithm.

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 $c_{ij} = 1/a_{ij}$. We assume that the rows of the cost matrix are in a permutation where the valley property holds. If necessary, the algorithm given by Tucker [7] can be used as preprocessing to obtain a conforming permutation.

The 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 vertex i to vertex 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.

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

$$C \;=\; \begin{array}{ccc} 5 & 4 & 1 \\ 3 & 2 & 1 \\ 2 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 4 & 4 \end{array}$$

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.

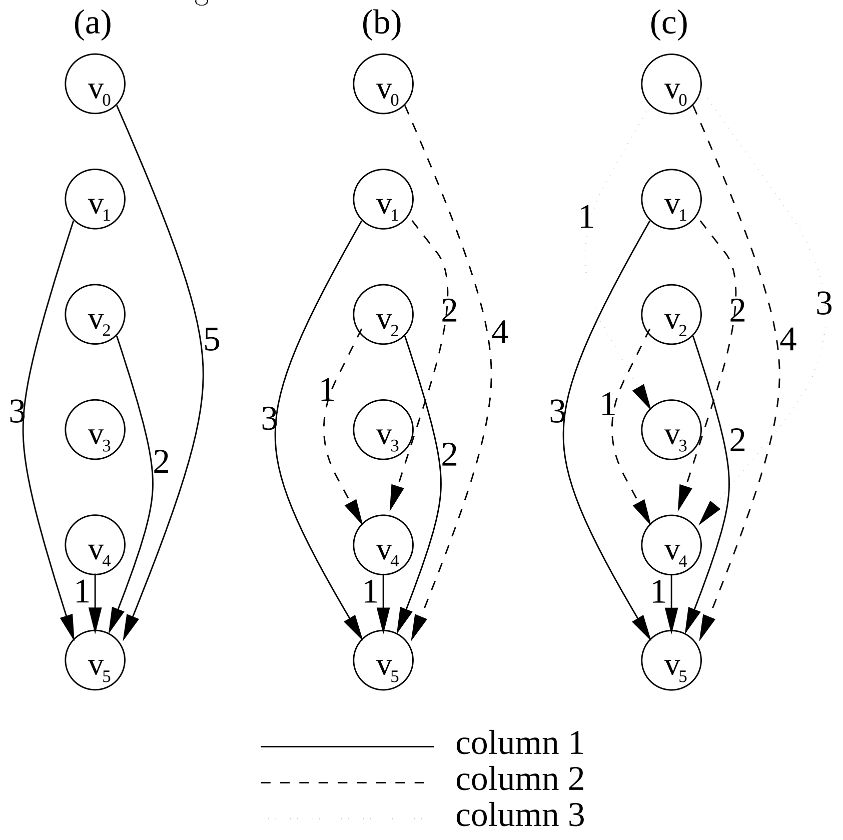
Once the row cost network has been constructed, discovery of the optimal solution to the original problem essentially corresponds to finding a shortest path from v_0 to v_m, interpreting the arc weights as distances. In general this

```
Build-Network:
    Begin with vertices $v_0, \ldots, v_m$ initialized for no arcs,
    and an $m \times n$ cost matrix $c$.
    for $(1 \leq j \leq n)$ do
        Scan $c[, j]$ from the top to find the minimum entry $c[z, j]$.
        Set $min\_entry \leftarrow c[z, j]$; $top \leftarrow z$; $bot \leftarrow z$.
        while $(top > 1$ or $bot < m)$ do
            Decrement $top$ and increment $bot$ until $c[top, j]$
                and $c[bot, j]$ are the farthest apart two column entries
                that are each $\geq min\_entry$ and
                $\leq$ (the smallest column entry $> min\_entry$).
                {In the first iteration, both $c[top, j], c[bot, j]$
                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 $(v_{top-1}, v_{bot})$ is a potential arc: add it to the network
                only if it does not already exist.
            Then set $w(v_{top-1}, v_{bot})$ to the minimum of its pre-existing weight
                ($\infty$ if it didn't exist) and $c[z, j]$.
            If we added the arc or updated its cost,
                also set $column(v_{top-1}, v_{bot}) \leftarrow j$.
            Set $min\_entry \leftarrow \max(c[top, j], c[bot, j])$.
```

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 network, there is a well known algorithm (as described in Cormen, *et al.* [1]) that merely takes the vertices in order $v_0, \ldots, v_{m-1}$, relaxing all arcs once, and requires only $O(E)$ time. (*Relaxation* of an arc (i, j) involves conditionally updating $d(j)$ to $d(i) + w(i, j)$ if that operation decreases $d(j)$.)

However, these standard algorithms find only paths consisting of explicitly represented arcs, and *our network contains implicit arcs as well.* 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

Figure 1: Initial Row Cost Network



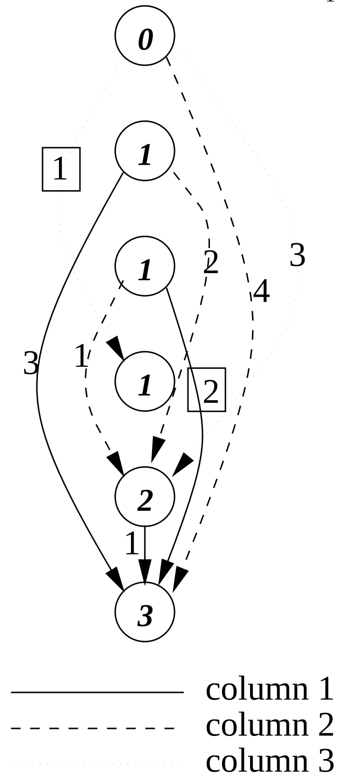
are also weight 1 arcs connecting all pairs of vertices in $\{v_0, v_1, v_2, v_3\}$ of increasing index. The shortest path explicitly represented in the network is $(v_0, v_4), (v_4, v_5)$ (among others) with distance 4, but in fact there is a covering path of distance 3 corresponding to arcs $(v_0, v_3), (v_2, v_5)$. Our network construction algorithm only builds a minimal set of arcs. If we were to explicitly build all arcs, the amount of work required in construction would be $O(m^2 n)$. Alternatively, extending the standard shortest path algorithm naively to relax all implicit arcs could require $O(m^3)$ time. Our Shortest-Path procedure relaxes only non-dominated implicit arcs, in addition to the explicit arcs, and runs in $O(m^2)$ time.

Shortest-Path iterates over all vertices in order of increasing index. For each vertex, it iterates over all arcs originating in that vertex, in order of increasing length. Each arc is relaxed, along with all of its (shorter) implicit arcs that are not dominated by (*i.e.,* more expensive than) another explicit arc originating in the same vertex. At termination $d(m)$ gives the minimum cost of a solution, and the actual variable assignments can be recovered by tracing backwards along the $predecessor(i)$ values, from m.

```
Shortest-Path:
    for ($i \leftarrow 1$ to $m$) do
        $d(v_i) \leftarrow \infty$
    $d(v_0) \leftarrow 0$
    for ($i \leftarrow 0$ to $m-1$) do
        $lower\_lim \leftarrow i$
        for (each arc $(i, j)$ in order of increasing $j$) do
            for ($k \leftarrow j$ down to $lower\_lim$) do
                if $d(i) + w(i, j) < d(k)$ then
                    $d(k) \leftarrow d(i) + w(i, j)$
                    $predecessor(k) \leftarrow i$
            $lower\_lim \leftarrow j$
```

Figure 2 shows the results of computing the shortest path in the network constructed in Figure 1. The distance of each vertex is shown within the vertex, and the weights of the two arcs constituting the shortest path are

Figure 2: Shortest Path Computed



marked by squares. In this case, (v_0, v_3) from column 3 and (v_2, v_5) from column 1 are the optimal row covers, so the optimal solution is

$$x_1 = 2, x_2 = 0, x_3 = 1; \; \sum x_j = 3.$$

5.1 Correctness

We will give an informal proof that BUILD-NETWORK followed by SHORTEST-PATH correctly yields an optimal solution to a minimax program whose cost matrix has the valley property.

Note that an optimal solution to a minimax program corresponds to a minimum cost covering of the cost matrix rows. We say x_j covers row i if $x_j \geq c_{ij}$ because then $x_j \geq 1/a_{ij}$ and the ith constraint is satisfied:

$$\max_j (a_{ij} x_j) \geq 1.$$

We say that a collection of arcs covers the row cost network if for every adjacent pair of vertices v_i, v_{i+1} the collection contains an arc (v_h, v_k) such that

$$h \leq i < k.$$

Therefore, in order to prove correctness it suffices to establish the following:

1. If the cost matrix C has the valley property, then a minimal cost covering of all of C's rows corresponds one-to-one with a minimal weight set of covering arcs in the row cost network constructed by BUILD-NETWORK.

2. SHORTEST-PATH finds a minimal weight set of covering arcs in the row cost network.

First, consider an arbitrary cost matrix with the valley property. In consequence of the valley property definition, if x_j covers any rows, it covers a consecutive group of rows. It should be easy to see that for such a matrix BUILD-NETWORK constructs maximum length, minimum weight arcs in the row cost network where the span and weight of an arc exactly correspond to the rows covered by setting some x_j to that weight value. In case exactly the same span can be covered by two different variables, the arc will have the lighter covering weight and be annotated with the column index of the cheaper covering variable. Consequently, if $x_a, \ldots, x_b > 0$ and all other $x_j = 0$ is a feasible solution, there must exist some (not necessarily distinct) arcs $e_1, \ldots, e_k$ in the network such that $w(e_1) \leq x_a, \ldots, w(e_k) \leq x_b$ and collectively the arcs cover the network. Simultaneously, if $e_1, \ldots, e_k$ is a sequence of arcs that cover the network, then $x_a = w(e_1), \ldots, x_b = w(e_k)$ is a feasible solution, where $a = column(e_1), \ldots, b = column(e_k)$. From all this it follows that an optimal solution to the minimax program corresponds to a minimum weight collection of arcs that cover the row cost network.

Next we show that SHORTEST-PATH finds such a minimum weight collection of arcs. A minimum weight covering collection in the network corresponds to a shortest path from v_0 to v_m if we interpret arc weights as distances and allow any arc (v_h, v_k) to be used to pass from any v_i to any v_j so long as $h \leq i < j \leq k$. In effect, a covering collection corresponds to a shortest path when we allow that each explicit arc in the collection represents a set of implicit arcs.

SHORTEST-PATH essentially differs from the standard shortest-path algorithm for DAGs only in its handling of implicit arcs. The standard algorithm walks through vertices $v_0, v_1, \ldots, v_m$ in order and relaxes all outgoing arcs. By the justification establishing correctness of that algorithm, SHORTEST-PATH would be correct if it relaxed all implicit arcs in addition to the explicit arcs. However, for an explicit arc (i, k), our algorithm only relaxes implicit arcs (i, j) where $w(i, j) = w(i, k)$ and $j < k$ if there does not exist a dominating arc (i, h). Arc (i, h) dominates (i, j) if

$$h \geq j \text{ and } \; w(i, h) < w(i, j).$$

Our algorithm is correct if any minimum cost path can be constructed solely from explicit arcs and the subset of implicit arcs that we relax.

Consider any arbitrary dominated implicit arc (i, j): since it is dominated there must exist a closest fitting dominating explicit arc

$$(h, k) \text{ where } w(h, k) < w(i, j) \text{ and } h \leq i < j \leq k,$$

such that there does not exist any distinct dominating arc

$$(h', k') \text{ where } h \leq h' < k' \leq k.$$

Since, in our row cost network, vertex distances are monotonically increasing with vertex index, $d(h) \leq d(i)$. By the rules for relaxation of explicit arcs, the implicit arc (h, j) will be relaxed, so

$$d(j) \leq d(h) + w(h, k).$$

Relaxation of (i, j) cannot set $d(j)$ lower than

$$d(i) + w(i, j) > d(h) + w(h, k),$$

so relation of dominated implicit arcs cannot improve the solution.

5.2 Complexity

It should be clear that 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. (Checking whether an arc exists, and possibly updating its entry can be done in constant time if we keep an $O(m^2)$ size array of arc data.) The network it constructs has $m + 1$ vertices and $O(\min(m^2, mn))$ arcs. Hence BUILD-NETWORK runs in $O(mn)$ time and uses $O(m^2)$ space. At each vertex i SHORTEST-PATH performs exactly one relaxation for every vertex j such that an explicit arc (i, k) exists where $i < j \leq k$. Since the arcs out of i can be found in order of decreasing length by scanning one $O(m)$-length row of the arc array, SHORTEST-PATH does $O(m)$ work at each vertex for an overall time complexity of $O(m^2)$.

Therefore, the optimal solution of a minimax program whose cost matrix has the valley property, and has been put in a conforming permutation, can be discovered in $O(mn + m^2)$ time using $O(m^2)$ space, which is linear in the size of the input when $n \geq m$.

6 Applications

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

Huang's thesis [4] 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

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

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 $T \ge T_0$ is attained. This problem can be converted to a minimax program of form (3) where $x_j = N_j$ and

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

Acknowledgments

We are very grateful to Dr. Alan J. Hoffman for references and comments that helped to clarify this material.

References

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

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

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

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

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

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

[7] Tucker, P. A. Efficient testing for a bitonic column property. Tech. Rep. CS97-546, Department of Computer Science and Engineering, University of California, San Diego, June 1997.