Project Number: DH-8901

Computational Methods for the Nucleolus

A Major Qualifying Project Report

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the requirements for the
Degree of Bachelor of Science

By

Leslie Ann Reed

Leslie Ann Reed

Date: April 30, 1989

Approved:

[illegible]

Professor David Housman, advisor

ACKNOWLEDGEMENTS

I would like to give thanks and gratitude to Professor David Housman for his support, patience, enthusiasm and continuous guidance throughout this project. The dedication he gives to his students goes beyond the "call of duty" and the many hours of energy devoted to this project are appreciated more than I can let him know.

I would also like to thank Kristina Szwaya for her guidance during the development of the initial Pascal implementation.

ABSTRACT

Two approaches for computing the nucleolus of a cooperative game are analyzed. The first is suggested by the definition of the nucleolus and involves the solution of successive linear programs. The second is suggested by the consistency of the nucleolus on reduced games and involves a recursive procedure. The second approach is shown to be untenable while the first approach is implemented in pascal and the complexity is determined theoretically and through numerical experimentation.

Contents

INTRODUCTION

DEFINITIONS:

The following are definitions of the key terms used in this project.

n-person cooperative game- A pair (N,v) where $N=\{1,2,3...n\}$ is the set of players and v is a real valued coalition function on the set of all subsets of N with $v(\phi) = 0$. The notation used will be v(1), v(12), etc.. rather than $v(\{1\})$ or $v(\{1,2\})$.

superadditive- A cooperative game is superadditive if $v(S \cup T) \geq v(S) + v(T)$ for all coalitions S and T satisfying $S \cap T=\phi$. For this project, assume v(S) is interpreted as the value the players in S can jointly share if they cooperate as a group. With this in mind, superadditivity has the interpretation that two disjoint coalitions are able to obtain as much jointly as each can separately. (ie. $v(1) + v(2) \leq v(12)$)

allocation- An allocation for the game (N,v) is a vector in R^n. If x is an allocation for the game (N,v), then we call x_i the "payoff" to player i.

Imputation- An imputation is an allocation vector where the sum of all of the individual allocations is equal to V(N).

lexicographically- A vector y is lexicographically smaller than a vector z if there is a j for which $y_j<z_j$, and $y_i=z_i$ for all $i<j$. This means that in a dictionary of vectors, y would appear before z.

excesses- Given a cooperative game (N,v) let $e(x,S) = v(S) - \sum_{i \epsilon S} x_i$ be the excess of coalition S relative to the cost allocation x; this is a measure of how much coalition S is likely to complain about the allocation x. Let $e(x)$ be the vector of excesses $e(x,S)$, $S \neq \phi, N$, ordered from highest to lowest.

nucleolus- The imputation of (N,v) that minimizes $e(x)$ lexicographically. In other words, the nucleolus is that allocation for which the largest complaint is as small as possible and is voiced by as few coalitions as possible, the next largest complaint is as small as possible and is voiced by as few coalitions as possible, etc...

balanced- A set of coalitions $\{S_1, \ldots, S_m\}$ is said to be balanced if there exist positive numbers $y_1, y_2, \ldots, y_m$ such that for each $i \epsilon N$, $\sum_{\substack{j \\ i \epsilon S_j}} y_j = 1$.

example:

Consider the set of coalitions $\{1, 2, 12\}$, $Y_1 = 1/2$, $y_2 = 1/2$ and $y_3 = 1/2$. Try player 1; $y_1 + y_3 = \frac{1}{2} + \frac{1}{2} = 1$.

ILLUSTRATION BY EXAMPLE:

To better understand these definitions, consider the following 3-person game:

$$\begin{aligned} v(1)=v(2)=v(3)&=0\\ v(12)&=1\\ v(13)&=2\\ v(23)&=3\\ v(123)&=4 \end{aligned}$$

Note that this game is superadditive. To solve for the nucleolus, choose an allocation and determine the values for the vector of excesses. Suppose $x = [1\ 1\ 2]$. Recall that $e(x,S)=v(s)-\Sigma x_i$. Now find the excess for each coalition S.

S	1	2	3	12	13	23
e(x,S)	-1	-1	-2	-1	-1	0

$$e(x) = [0\ -1\ -1\ -1\ -1\ -2]$$

Coalition 23 has the most to complain about while 3 has the least. The next step is to choose another x and then another, if necessary, to try to make all coalitions complain as little as possible while at the same time, trying to make the degree of their complaints as uniform as possible.

Choose $x = [.5\ 1.25\ 2.25]$. Now find the excess for each coalition S.

S	1	2	3	12	13	23
e(x,S)	-.5	-1.25	-2.25	-.75	-.75	-.5

$$e(x) = [-.5\ -.5\ -.75\ -.75\ -1.25\ -2.25]$$

Note that this excess vector is lexicographically smaller than the first excess vector because -.5 is less than 0.

This allocation is the nucleolus.

Proof:

The nucleolus, as defined, is the allocation that minimizes the largest complaint {largest excess}. Therefore, if there is a better allocation, we will be able to minimize the excess for both coalitions 1 and 23. Can we do this?

$$\begin{array}{llll}
\text{Coalition 1:} & V(1) - x_1 \le -.5 & & \\
 & 0 - x_1 \le -.5 & \Rightarrow & x_1 \ge .5 \\
\text{Coalition 23:} & V(23) - x_2 - x_3 \le -.5 & & \\
 & 3 - x_2 - x_3 \le -.5 & \Rightarrow & x_2 + x_3 \ge 3.5
\end{array}$$

Because $V(N) = 4$, we know $x_1 = .5$ and $x_2 + x_3 = 3.5$. We cannot determine the specific values of x_2 and x_3, so we must now determine if we can minimize the next largest excess.

$$\begin{array}{llll}
\text{Coalition 12:} & V(12) - x_1 - x_2 \le -.75 & & \\
 & 1 - x_1 - x_2 \le -.75 & \Rightarrow & x_1 + x_2 \ge 1.75 \\
\text{Coalition 13:} & V(13) - x_1 - x_3 \le -.75 & & \\
 & 2 - x_1 - x_3 \le -.75 & \Rightarrow & x_1 + x_3 \ge 2.75
\end{array}$$

We already know that $x_1 = .5$ so it follows that $x_2 \ge 1.25$ and $x_3 \ge 2.25$. Allocating each coalition its lower bound, we get $x_1 = .5$, $x_2 = 1.25$ and $x_3 = 2.25$, which satisfies $x_1 + x_2 + x_3 = 4$ and all of the above inequalities. This allocation is also the allocation we were testing. Therefore, $x = [.5\ 1.25\ 2.25]$ is the nucleolus. You cannot get a better solution.

SOLVING FOR THE NUCLEOLUS USING LINEAR PROGRAMMING

DESCRIPTION AND JUSTIFICATION OF THE METHOD:

Because the problem of solving for the nucleolus is minimizing the excesses subject to certain constraints, it is possible to solve these problems using a linear programming approach.

The optimization problem is as follows:

$$\text{Objective: } \min_{\substack{\{x:\Sigma x_i=v(n) \\ x_i \geq v(i)\}}} \quad \max_{\phi \subset S \subset N} e(x,S)$$

Solving this optimization problem won't necessarily determine a unique solution. If the initial solution is not unique, you must fix the values of excesses that cannot be lowered and then solve the new problem to minimize the maximum of the rest of the allocations. Continue in this manner until a unique optimal solution is obtained. This optimization problem can be transformed into a linear programming problem by letting $\alpha = \max e(x,S)$ and introducing linear constraints.

The procedure computes the nucleolus of a game by solving a sequence of linear programs until the set of feasible solutions is a singleton. Specifically, let $\alpha_0 = 0$, $NF_0=F_0=\{N\}$, $SF_0 = \phi$, $u_0 = v$, and $k = 0$. Let (x^{k+1},α_{k+1}) be an optimal solution to the primal linear program

$$\begin{array}{lllll}
\text{Objective:} & \min \alpha & & & \\
\text{Constraints:} & \sum_{i\epsilon S} x_i & = & u_k(S) & \text{for } S \in F_k \\
& \sum_{i\epsilon S} x_i + \alpha & \geq & u_k(S) & \text{for } S \notin F_k \\
& x_i & = & v(i) & \text{for } i \in SF_k \\
& x_i & \geq & v(i) & \text{for } i \notin SF_k
\end{array}$$

Let (y^{k+1}, z^{k+1}) be a corresponding optimal solution to the dual linear program

$$\text{Objective:} \quad \max \sum_{S} u_k(S) y_S + \sum_{i} v(i) z_i$$

$$\begin{array}{llll} \text{Constraints:} & \sum_{S \ni i} y_S + z_i & = 0 & \text{for } i \in N \\ & \sum_{S \notin F_k} y_S & = 1 & \\ & y_S & \geq 0 & \text{for } S \notin F_k \\ & z_i & \geq 0 & \text{for } i \notin SF_k \end{array}$$

If (x^{k+1}, α_{k+1}) is the unique <u>feasible</u>, and so optimal, solution to the primal linear program, then x^{k+1} is the nucleolus.

If (x^{k+1}, α_{k+1}) is not the unique feasible solution, the LP must be updated in the following manner. If a dual optimal variable from this solution has positive value, the corresponding inequality becomes an equality, the value of α_{k+1} is substituted into the equation and the variable is labeled a new free variable.

$$NF_{k+1} = \{S \notin F_k : y^{k+1}{}_S > 0\},$$

$$F_{k+1} = F_k \cup NF_{k+1},$$

$$SF_{k+1} = \{i \notin SF_k : z^{k+1}{}_i > 0\}, \text{ and}$$

$$u_{k+1}(S) = \{ \begin{array}{ll} u_k(S) - \alpha_{k+1} & \text{for } S \in NF_{k+1} \\ u_k(S) & \text{for } S \notin NF_{k+1} \} \end{array}$$

Now increment k and solve the new linear program.

Because the primal LP will have a large number of constraints, it may require a lot of computational time when it is solved. Therefore, the dual LP will be solved using the revised simplex method. The dual will have more variables but less constraints which will decrease calculation time.

The trade-off is that determining the initial basis when solving the dual LP is more difficult than when solving the primal LP but the calculating time saved is worth the difficulty.

The revised simplex method found in Chvatal(1980) was used to solve the dual LP, with a few alterations. Rather than solving $Ax = b$, we solved $x = A^{-1}b$. The former is solved in order n^3 while the latter is solved in order n^2. By using a modification of the algorithm in James Endersby's (Central State University, 1983) paper "Toward Efficiency In Linear Programming", we are ensured integer values so we won't encounter the round-off errors that tend to occur when updating inverse matrices. Therefore, it isn't necessary to use more difficult triangular factorization and updating procedures.

James Endersby proves that integer values can be ensured when solving linear programs when a common denominator for all elements is used. The rules to control the denominator are: 1. Denominator (D) of the original tableau is 1. 2. Off pivot elements become $(A_{rk}A_{ij} - A_{rj}A_{ik}) / D$ where A_{rk} is the pivot element and D is the common denominator. Make no changes to the pivot row. 3. The pivot element is the denominator for the new tableau.

Another alteration made to Chvatal form is that we use, store and update a free variable matrix which becomes the start of the initial basis for the next LP. Free variables are put into the basis immediately and then they never exit.

The last alteration was made because a unique solution may not be found after the initial LP is solved. The original primal LP must be updated and solved again. This is done by changing the primal constraints that correspond to the dual variables that have positive value from inequalities to equalities and labeling these dual variables as new free variables. If the optimal value for a dual variable is positive, the primal equation that corresponds to that dual variable must be an equality for all primal optimal solutions. For these constraints, the value for alpha must also be inserted. Now we have a new primal and dual with a new α. (This α is interpreted to be the next largest excess value.) This new LP must be solved to try to reach a unique primal optimal solution.

The initial basis for the new LP is determined in the following manner:

Note: when a variable is entering the basis, the constraint coefficient column corresponding to that variable is added to the basic variable matrix to determine whether or not it is linearly independent of the other basic variables.

step 1- Put the free variables from the previous LP's basis in the new basis.

step 2- If rank equals n then go to step 9. If not <u>and</u> there is still at least one variable whose dual optimal value in the previous LP was positive, then choose one of the those variables to enter the basis. This is a new free variable.
Also change the value of the corresponding objective function coefficient. If there aren't any more new free variables, go to step 4.

step 3- Do a column reduction to determine whether or not the new variable is independent of the others. If it is, add it to the basis. Otherwise, don't add it to the basis. Return to step 2.

step 4- Put the variables whose dual optimal value in the previous LP equals zero in the nonbasic variable list.

step 5- Choose the first nonbasic variable in the list and its complement to try to enter them into the basis.
The following rules apply:

1. If the complement is already a basic variable, add the nonbasic variable being considered to the basis . Then go to step 6.
2. Otherwise, check to see if the complement is redundant. If it isn't, add the nonbasic variable and its complement to the basis and go to step 6. If it is redundant, do step 5 again with the next nonbasic variable.

step 6- If rank equals n+1 go to step 8. If not choose a nonbasic variable to enter the basis.

step 7- Do a column reduction to determine whether or not the new variable is independent of the others. If it is, add it to the basis. Otherwise, don't add it to the basis. Return to step 6.

step 8- solve next LP.

step 9- optimal solution is found.

ILLUSTRATION BY EXAMPLE:

To illustrate how this method works, consider the 3-person game from above:

$$\begin{aligned} v(1)=v(2)=v(3)&=0 \\ v(12)&=1 \\ v(13)&=2 \\ v(23)&=3 \\ v(123)&=4 \end{aligned}$$

For this game, the LP is as follows:

$$\begin{array}{ll} \min & \alpha \\ \text{S.T.} & \alpha \ge 0 - x_1 \\ & \alpha \ge 0 - x_2 \\ & \alpha \ge 0 - x_3 \\ & \alpha \ge 1 - (x_1 + x_2) \\ & \alpha \ge 2 - (x_1 + x_3) \\ & \alpha \ge 3 - (x_2 + x_3) \\ & x_1 + x_2 + x_3 = 4 \\ & x_1, x_2, x_3 \ge 0 \end{array}$$

$$\begin{array}{llllll} \min & \alpha & & & & \\ \text{S.T.} & x_1 & & & + \alpha & \ge 0 \\ & & x_2 & & + \alpha & \ge 0 \\ & & & x_3 & + \alpha & \ge 0 \\ & x_1 + & x_2 & & + \alpha & \ge 1 \\ & x_1 + & & x_3 & + \alpha & \ge 2 \\ & & x_2 + & x_3 & + \alpha & \ge 3 \\ & x_1 + & x_2 + & x_3 & & = 4 \\ & x_1, x_2, x_3 \ge 0 & & & & \end{array}$$

The dual that corresponds to this LP is:

$$\begin{array}{ll} \max & 0y_1 + 0y_2 + 0y_3 + y_{12} + 2y_{13} + 3y_{23} + 4y_{123} + 0S_1 + 0S_2 + 0S_3 \\ \text{S.T.} & y_1 + 0y_2 + 0y_3 + y_{12} + y_{13} + 0y_{23} + y_{123} + S_1 \qquad\qquad = 0 \\ & 0y_1 + y_2 + 0y_3 + y_{12} + 0y_{13} + y_{23} + y_{123} + \qquad S_2 \qquad = 0 \\ & 0y_1 + 0y_2 + y_3 + 0y_{12} + y_{13} + y_{23} + y_{123} + \qquad\qquad S_3 = 0 \\ & y_1 + y_2 + y_3 + y_{12} + y_{13} + y_{23} + 0y_{123} \qquad\qquad\quad = 1 \\ & \quad y_1, y_2, y_3, y_{12}, y_{13}, y_{23}, S_1, S_2, S_3 \ge 0 \\ & \qquad y_{123} \text{ is a free variable} \end{array}$$

Solving the dual using the revised simplex method enables us to determine the optimal primal solution as well as the optimal dual solution.

The first dual and primal optimal solutions are:

dual		primal	
y_1	$= 1/2$	α	$= -1/2$
y_{13}	$= 0$	x_1	$= 1/2$
y_{23}	$= 1/2$	x_2	$= 3/2$
y_{123}	$= -1/2$	x_3	$= 2$

Note: These solutions give the values of the basic variables. The dual optimal solution is unique. The primal optimal solution is one of many optimal solutions that satisfy the equations $x_1=1/2$, $x_2 + x_3=7/2$, $x_2 \ge 1$, $x_3 \ge 2$. Since there is more than one allocation that satisfies these equations, we need to update the LP and solve it again. The equations that correspond to y_1 and y_{23} become equalities and the value of alpha (-1/2) is inserted into the equations that these variables represent. These variables become new free variables.

The new Primal LP is:

$$\begin{array}{llllll}
\text{min} & \alpha_2 & & & & \\
\text{S.T.} & x_1 & & & & = 1/2 \\
 & & x_2 & & + \alpha_2 & \geq 0 \\
 & & & x_3 & + \alpha_2 & \geq 0 \\
 & x_1 & + x_2 & & + \alpha_2 & \geq 1 \\
 & x_1 & + & x_3 & + \alpha_2 & \geq 2 \\
 & & x_2 & + x_3 & & = 7/2 \\
 & x_1 & + x_2 & + x_3 & & = 4 \\
 & \multicolumn{5}{l}{x_1, x_2, x_3 \geq 0}
\end{array}$$

The dual that corresponds to this LP is:

$$\begin{array}{l}
\text{max } 1/2y_1 + 0y_2 + 0y_3 + y_{12} + 2y_{13} + 7/2y_{23} + 4y_{123} + 0S_1 + 0S_2 + 0S_3 \\
\text{S.T.} \quad y_1 + 0y_2 + 0y_3 + y_{12} + y_{13} + 0y_{23} + y_{123} + S_1 = 0 \\
\quad 0y_1 + y_2 + 0y_3 + y_{12} + 0y_{13} + y_{23} + y_{123} + S_2 = 0 \\
\quad 0y_1 + 0y_2 + y_3 + 0y_{12} + y_{13} + y_{23} + y_{123} + S_3 = 0 \\
\quad 0y_1 + y_2 + y_3 + y_{12} + y_{13} + 0y_{23} + 0y_{123} = 1 \\
\qquad y_2, y_3, y_{12}, y_{13}, S_1, S_2, S_3 \geq 0 \\
\qquad y_1,\ y_{23},\ y_{123} \text{ are free variables}
\end{array}$$

In solving this new LP, it would be best if we could let the variables in the initial basic feasible solution consist of the basic variables in the optimal solution from the previous LP. Unfortunately, for this new LP, having y_1, y_{13}, y_{23}, and y_{123} in the basis isn't feasible because the constraint coefficient columns that correspond to these variables are not linearly independent. Therefore, we must change at least one of the variables.

We find that the first basic matrix for this dual is:

y_1	y_2	y_{13}	y_{123}	RHS
0	0	1	0	1/2
1	0	0	0	0
0	1	0	0	1/2
0	0	0	1	-1/2
0	0	0	0	1

(note: y_1, y_2, y_{13} and y_{123} are the basic variables.)

We determined this initial feasible basis using the eight step procedure from above. Step 1 put y_{123}, the only free variable from the previous LP, into the basis. Step 2 put y_1 into the basis because it is a new free variable. y_{23} is also a new free variable but is determined through step 3 that it is redundant so it is not put into the basis. Step 4 puts y_2 and its complement y_{13} into the basis as they are nonbasic variables and not redundant.

Now we find the second dual and primal optimal solutions:

dual		primal	
y_1	$= -1/2$	α	$= -3/4$
y_{13}	$= 1/2$	x_1	$= 1/2$
y_{12}	$= 1/2$	x_2	$= 5/4$
y_{123}	$= -1/2$	x_3	$= 9/4$

This solution suggests the allocation $x_1=1/2$, $x_2=5/4$, $x_3=9/4$. Since this is the only allocation possible, we have found the nucleolus. Therefore, $(N,v) = (1/2, 5/4, 9/4)$.

PASCAL IMPLEMENTATION OF THE ALGORITHM:

The program is separated into four files: Nuc.pas, Initial.pas, Setuplp.pas and Solvelp.pas. Nuc.pas contains the main program and calls the other three files to perform various tasks. In Initial.pas, dual variable linked lists for basic and nonbasic variables are initialized. In Setuplp.pas, the basic feasible solutions for the first LP and all updated LP's, if needed, are determined. This file also updates LP's. In Solvelp.pas, the revised simplex method, with the modifications discussed earlier, is performed.

The pascal implementation uses the following variable names to represent the symbols of the linear program being solved(p.5):

<u>Algorithm</u>	<u>Pascal Implementation</u>	<u>Explanation</u>
α	Alpha	
S	Basic^.index, Nonbasic_head^.index	indicates coalition name.
constraint coefficients	Procedure	accessed through column.
x	x	
F_k, SF_k	Free	Matrix of free variables.
$U_k(S)$	Basic[j]^.obj, Dual_variable_pointer	Right hand side of equation. $U_k(S) - \alpha$ if new free; otherwise, $U_k(S)$.
y_S	Basic_value[j]	if Basic[j]^.index=S and Basic[j]^.slack =false.
z_i	Basic_value[j]	if Basic[j]^.index=i and Basic[j]^.slack = true.
V(I)	Basic^.obj, Nonbasic_head^.obj	Right hand side of equation for slack variables. Equals the value of the singleton it represents.

THE PROGRAM:
Nuc.pas:

```
Procedure Nucleolus_simplex( var G: Game; var x: Allocation );
           {This program assumes: Σv ≤ v(N)}
        {This procedure calculates the nucleolus using the}
                    {revised simplex method.}

Type    Vector                   = array[player] of integer;
        Matrix                   = record
                                    num   :array[player,player] of integer;
                                    den   :integer;
                                    rank  :player;
                                   end;
        Dual_variable_pointer    = ^Dual_variable;
        Dual_variable            = record
                                    index: Coalition;
                                    slack: Boolean;
                                    obj:   Real;
                                    comp:  Dual_variable_pointer;
                                    next:  Dual_variable_pointer;
                                   end;
        Basic_array              = array[Player] of Dual_variable_pointer;


Var     Alpha                        : Real;
        Basic                        : Basic_array;
        Basic_value                  : Vector;
        Nonbasic_head                : Dual_variable_pointer;
        AB                           : Matrix;
        AB_inverse                   : Matrix;
        Free                         : Matrix;
        Free_inverse                 : Matrix;
        Basic_variables              : Player;
        Solution_found               : Boolean;


{$i nucwrite}
{$i initial}
{$i setuplp}
{$i solvelp}

begin
  Write_Setup;
  Basic_variables := G.n + 1;
  Setup_dual_variables;{initialize the basic and nonbasic linked lists.}
  setup_initial_linear_program; {determines an initial LP.}
  repeat
     Solve_linear_program;        {executes the revised simplex method}
     Setup_linear_program(solution_found);  { Sets up next LP}
     Write_LP;
  until Solution_found;       {repeat until a unique optimal solution}
                                  {for the nucleolus is found.}

  Write_Time;
end;  {of this procedure}
```

Initial.pas:

```
procedure setup_dual_variables;

          {This procedure puts all variables except the}
           {grandcoalition into nonbasic head and puts}
                 {the grandcoalition into basic}

var first_q,next_p,next_q :dual_variable_pointer;
    p,q,l                 :dual_variable_pointer;
      m,f                 :integer;
        c                 :coalition;
        i                 :player;

begin
  new(l);
  basic[1] := l;
  with l^ do                           {put to grand coalition}
    begin                          {into basic variable linked list.}
      index := grand(G);
      slack := false;
      obj   := G.v[grand(G)];
      comp  := nil;
      next  := nil;
    end;

  new(next_p);
  new(next_q);
  nonbasic_head := next_p;
  first_q := next_q;

  for c := 1 to ((grand(G) + 1) div 2 - 1) do {put all other variables}
    begin                                      {into the nonbasic linked}
      p := next_p;                                      {list.}
      q := next_q;
      new(next_p);
      new(next_q);

      with p^ do
        begin
          index := c;
          slack := false;
          obj   := G.v[c];
          comp  := q;          {this keeps track of the complement}
                                  {of the variable in question.}
          next  := next_p;
        end;

      with q^ do
        begin
          index := grand(G) - c;
          slack := false;
          obj   := G.v[grand(G) - c];
          comp  := p;
          next  := next_q;
        end;

    end; {of the for c loop}
  for i:= 1 to (G.n - 1) do
    begin
      q := next_q;
      new(next_q);
      with q^ do
        begin
         index := singleton(i);       {put the slack vars into the}
                                     {nonbasic variable linked list}
         slack := true;
         obj   := G.v[singleton(i)];
         comp  := nil;
         next  := next_q;
        end;

    end;
  q := next_q;
  with q^ do
    begin
      index := singleton(G.n);        {end the linked list with the}
                                          {final slack variable}
      slack := true;
      obj   := G.v[singleton(G.n)];
      comp  := nil;
      next  := nil;
    end;

  p^.next := first_q;                  {link the two lists together}
  dispose(next_p);
end; {of the procedure}

procedure column(d:dual_variable_pointer; var a:vector;
                 free_variable:boolean);

             {This procedure generates columns.}

Var  i,j :  player;

begin
  for i := 1 to G.n do                        {component i of the vector}
    Begin                                    {a is one if player i is in}
      if element(i,d^.index) then a[i] := 1   {the coalition d.index}
      else a[i] := 0;                         {otherwise, it is zero.}
    end;

  if free_variable or d^.slack then a[G.n + 1]:=0  {if the variable is}
                                            {free or slack, this row, that}
                                       {corresponds to α should have a 0.}
  else a[G.n + 1]:=1;

end; {of this procedure}
```

Setuplp.pas:

```
 function First_nonzero( var  M   :Matrix;
                         var  j   :Player) : Player;

  {This procedure determines the first nonzero row of column j.}
{If there are no nonzeros, this function returns basic_variables + 1.}
```

```
var i  :player;

begin
  i := 1;
  while (M.num[i,j] = 0) and (i <= basic_variables) do i := i + 1;
  first_nonzero := i;
end;

procedure Column_pivot( var M1, M2   :matrix;
                            i, j     :player);

   {                                              }
   {This procedure performs a pivot on |M1| with respect}
   {to the (i,j) element of M1.        |M2|            }
   {                                              }

var k, l: Player;
    M1_il:integer;
    M1_ij:integer;
    key  :char;

begin
  M1_ij := M1.num[i,j];
  for l:=1 to M1.rank do  {reduce all nonpivot element}
     if l <> j then         {values in column l to 0}
       begin
         M1_il:=M1.num[i,l];
         for k:=1 to basic_variables do
          begin
            M1.num[k,l] := (M1_ij*M1.num[k,l] - M1_il*M1.num[k,j])
                             div M1.den;
            M2.num[k,l] := (M1_ij*M2.num[k,l] - M1_il*M2.num[k,j])
                             div M1.den;
          end;
       end;

  M1.den := M1_ij;      {the denominators equal the pivot element}
  M2.den := M1_ij;

end; {end of this procedure}

procedure Rearrange_columns(var M1, M2                  :matrix;
                            var j, k                    :player);

     {This procedure rearranges cyclicly, columns j,j+1,...k}
     {                                                      }
     {        of M=|M1| in the order k,j,j+1,...k-1.         }
     {             |M2|                                      }
     {                                                      }

Var     tempv1, tempv2      :vector;
        i,l                 :integer;

begin
  for i := 1 to basic_variables do           {put column k of M}
    begin                                  {into temporary storage.}
     tempv1[i] := M1.num[i,k];
     tempv2[i] := M2.num[i,k];
    end;

  for l := k-1 downto j do                    {Move columns j,...,k-1}
    begin                                        {over one column.}
      for i := 1 to basic_variables do
        begin
          M1.num[i,l+1] := M1.num[i,l];
          M2.num[i,l+1] := M2.num[i,l];
        end;
    end;

  for i := 1 to basic_variables do        {Put the column to be moved}
    begin                                   {into the correct space.}
      M1.num[i,j] := tempv1[i];
      M2.num[i,j] := tempv2[i];
    end;
end;    {of this procedure}

procedure Column_insert(var M1, M2 : matrix; var A : vector);

         {This procedure adds new columns to the basic matrix.}

var i,j,k,l  :player;
    temp     :integer;

begin
  k := M1.rank + 1;
  M1.rank := k;               {Put column k into the basic matrix.}
  M2.rank := k;
  for i:=1 to basic_variables do
    begin
      M1.num[i,k] := A[i] * M1.den;
                          {Setup the identity}
                  {matrix column corresponding to column k}
      if i = k then M2.num[i,k] := M2.den else M2.num[i,k] := 0;
    end;

  for j:= 1 to M1.rank-1 do       {pivot on element i,j if}
    begin                          {element i,k is nonzero}
      l := first_nonzero(M1,j);
      if M1.num[l,k] <> 0 then
        begin
          for i := 1 to basic_variables do
            begin
              M1.num[i,k] := M1.num[i,k] - A[l]*M1.num[i,j];
              M2.num[i,k] := M2.num[i,k] - A[l]*M2.num[i,j];
            end;
        end;
    end;
  {find the first nonzero row in the kth column of M1}
  i := first_nonzero(M1,k);

  if i <= basic_variables then
        {if a nonzero row i element is found in col k,}
        {rearrange columns in lower triangular form}
             {and pivot on the nonzero element;}
    begin
      j := 1; while first_nonzero(M1,j) < i do j := j + 1;
      if j < k then Rearrange_columns(M1,M2,j,k);
      Column_pivot(M1, M2, i, j);
    end
  else                {otherwise, remove the column of zeros}
    begin
      M1.rank := M1.rank - 1;
      M2.rank := M2.rank - 1;
    end;
end; {end of procedure}

Procedure Insert_new_free_variables(var solution_found  :boolean);

{This procedure puts the variables that have positive value in the}
{previous solution into the basic matrix and the free matrix and }
  {changes the value of the corresponding objective coefficients.}

var x_column                :vector;
    i,old_free_rank         :player;
    temp                    :dual_variable;
    p                       :dual_variable_pointer;

begin
  i:= Free.rank + 1;       {start looking at the basic variables}
                           {that come after the old free variables}

  while (i <= basic_variables) and not solution_found do
    begin
      if (basic_value[i]*AB_inverse.den > 0) then {if the value of the}
        begin                                   {dual variable is positive,}
          column(Basic[i],x_column,true);          {generate the column}
          old_free_rank := Free.rank;
          column_insert(Free,Free_inverse,x_column);

          if old_free_rank <> Free.rank then  {if the variable isn't}
            begin                            {redundant, add it to the}
              If not Basic[i]^.slack then      {basic variable list.}
              Basic[i]^.obj := Basic[i]^.obj - Alpha;
              if i > Free.rank then basic[Free.rank] := basic[i];
            end;

          solution_found := ( Free.rank = G.n)
        end
      else
        begin
          basic[i]^.next := nonbasic_head; {if the dual variable}
          nonbasic_head := basic[i];        {equals zero, put it}
                                          {back into nonbasic_head.}
        end;

      i:=i+1;
  end;

end;  {of this procedure}

Procedure move_free_variables;

    {This procedure puts the free matrix from the previous LP plus}
     {the new free variables into the basic matrix for the new LP}
```

```
var i,j    : player;

begin
  for j:= 1 to Free.rank do
    begin
      for i:=1 to basic_variables do
        begin                              {copy the free variables from}
          AB.num[i,j] := Free.num[i,j];        {previous LP's into the}
          AB_inverse.num[i,j] := Free_inverse.num[i,j];  {basic matrix}
        end;
    end;

   AB.rank := Free.rank;                  {Update ranks and denominators}
   AB_inverse.rank := Free_inverse.rank;
   AB.den := Free.den;
   AB_inverse.den := Free_inverse.den;

end; {of this procedure}

Procedure remove_nonbasic_variable(var x  :dual_variable_pointer);

        {This procedure removes x from the nonbasic linked list.}
    {x_previous immediately precedes x in the nonbasic variable list.}

Var  x_previous, p  : Dual_variable_pointer;
     found_x_prev   : boolean;

Begin
  If x = nonbasic_head then nonbasic_head := x^.next
  else
    begin
      p := nonbasic_head;
      while p^.next <> x do p := p^.next;
      p^.next := x^.next;
    end;
end;  {of this procedure}

Procedure Insert_complement_pairs;

    {This procedure inserts a nonbasic variable and its }
   {complement into basic if the complement is not already}
   {in basic and they are not redundant.  If the complement}
         {is in basic, insert the nonbasic variable.}

Var p,q                  :  Dual_variable_pointer;
    In_basis             :  Boolean;
    X                    :  Player;
    Old_AB_rank          :  Integer;
    a                    :  Vector;

Begin
  p := nonbasic_head;
  Old_AB_rank := AB.rank;
  while (Old_AB_rank = AB.rank) do
    begin
      If (p^.comp <> nil) then
        begin
          Column(p,a,false);                 {generate a nonbasic column}
        Column_insert(AB,AB_inverse,a);

        In_basis := false;                              {check to see}
        X := 1;                                       {if the complement}
        while (X < AB.rank) and (not In_basis) do          {of the}
          begin                                         {nonbasic var}
            In_basis := (basic[X] = p^.comp);             {generated}
            X := X + 1;                               {above is in basic}
          end;

        If In_basis then                               {If it is in Basic,}
          begin                                     {add the nonbasic column}
            Basic[AB.rank] := p;                    {to basic and remove it}
            remove_nonbasic_variable(p);              {from nonbasic_head.}
          end
        else
          begin
            q := p^.comp;
            column(q,a,false);                      {If it is not in Basic,}
            column_insert(AB,AB_inverse,a); {generate complement and}
            If Old_AB_rank + 2 = AB.rank then {try to insert into}
              begin                       {AB. If not redundant, add the}
                Basic[AB.rank - 1] := p;     {nonbasic var and its}
                remove_nonbasic_variable(p); {complement to Basic and}
                Basic[AB.rank] := q;            {remove them from}
                remove_nonbasic_variable(q);     {nonbasic_head.}
              end
            else
              AB.rank := Old_AB_rank;         {If redundant, try next}
                                                {nonbasic variable.}
          end;
      end; {if p^.comp loop}

        p := p^.next;

    end; {while p loop}
end; {of the procedure}

Procedure Insert_nonbasic_variables;

   {This procedure inserts eligible nonbasic variables into the basic}
   {matrix and then puts the variable into the basic variable linked}
     {list and removes the variable from the nonbasic linked list.}

var  p              :dual_variable_pointer;
     a              :vector;
     old_ab_rank    :integer;
     i              :player;

begin
  p:=nonbasic_head;
  while (p <> nil) and ( AB.rank < basic_variables) do
          {while not at end of linked list and more variables}
                     {are needed for the basis,:}
     begin
        column(p,a,false);          {generate a nonbasic variable column}
        old_ab_rank :=  AB.rank;
        column_insert(AB,AB_inverse,a);
        if old_ab_rank <> AB.rank then {If the column is not redundant,}
          begin                        {add it to the basic variable list.}
            basic[AB.rank] := p;
            remove_nonbasic_variable(p);
                       {remove the new basic variable from the}
                          {nonbasic variable linked list.}
          end;

        if p=nil then p := nonbasic_head
        else p:=p^.next;
      end;
end;  {of this procedure}

procedure find_dual_variable_values;

var   i  : player;

begin

  for i := 1 to Basic_variables do
    begin
      basic_value[i] := AB_inverse.num[i,basic_variables];
    end;

end; {procedure}

procedure setup_initial_linear_program;

 {This procedure initializes the first linear program}

var   a    :vector;
      i    :player;

begin
  column(basic[1],a,true);               {generate grandcoalition column}
  for i := 1 to basic_variables do
    begin
      Free.num[i,1]:=a[i];
      if i = 1 then Free_inverse.num[i,1]:=1  {row corresponding to α}
      else Free_inverse.num[i,1] := 0;  {consists of 0's if variables}
                                        {are free and 1's if not free}
    end;

  Free.rank := 1;                      {initialize ranks, denominators,}
  Free.den  := 1;                             {and free matrix.}
  Free_inverse.rank := 1;
  Free_inverse.den := 1;
  move_free_variables;
  Insert_complement_pairs;
  Insert_nonbasic_variables;    {setup the initial basic variable list}
  Find_dual_variable_values;  {calculate values of the dual variables.}

end;     {of the procedure.}

procedure Setup_linear_program(var solution_found  :boolean);

{This procedure sets up the basic variables before the LP is solved}
```

```
begin
   Solution_found := false;
   Insert_new_free_variables(solution_found);
           {Insert variables in current LP with value > 0}
                         {into the matrix.}
   Move_free_variables;                    {save the free variable matrix}
   If  AB.rank < G.n then
     begin
        If  AB.rank < basic_variables then     {if you need more basic}
          begin                      {variables to execute the revised}
           Insert_complement_pairs;                  {simplex method,}
           Insert_Nonbasic_variables;              {search for eligible}
          end;                                     {nonbasic variables.}
     end

     else                          {if you have G.n independant equations,}
       begin                        {the optimal solution has been found.}
         Solution_Found := true;
       end;

    Find_dual_variable_values;      {Calculate dual variable values.}
end;  {of this procedure}
```

Solvelp.pas:

```
 {This is the revised simplex algorithm applied to the dual to}
       {the minimize the maximum excess linear program.}

Procedure Step1;

    {This procedure executes step 1 of the revised simplex method}
       {which solves for the value of the primal variables.}

var i,j    :player;
    sum    :real;

Begin

  for i:=1 to basic_variables do  {counter for column of}
    begin                          {inverse basic matrix}
      sum:=0;
      for j:=1 to basic_variables do  {counter for row of inverse}
                {basic matrix and element of obj. coeff. vector}
        begin
          sum:=sum + (AB_inverse.num[j,i]*Basic[j]^.obj);
                {mult. obj coeff values by inv. bas. mat.}
        end;

      x[i]:=sum / AB_inverse.den;
    end;

end; {of this procedure}

Procedure Step2(var enter                 :vector;
                var before_enter_var      :dual_variable_pointer;
                var enterpointer          :dual_variable_pointer;
                var LP_solution_found     :boolean);
```

```
{This procedure calculates step 2 of the revised simplex method}
        {which determines the entering variable.}

var p            :dual_variable_pointer;
    i,j          :player;
    sum, answer, zero :real;

 Begin
  answer:=0;
  p:=nonbasic_head;
  before_enter_var := nil;
  Zero := (1E-10 * G.v[grand(G)]); {reset 0 because of round-off}

  while (p <> nil) and (answer <= Zero) do
              {while there is an unexamined}
             {nonbasic variable and it is not}
            {profitable to enter that variable:}
    Begin
      column(p,enter,false);      {generate the nonbasic column}

      sum := 0;
      for i:= 1 to basic_variables do
       begin
         sum:=sum + (enter[i] * x[i]);  {get product of nonbasic}
       end;                                   {col and x}
      answer:= (p^.obj) - (sum);
            {find the diff between obj coeff val and sum}

      If answer <= Zero then
        begin
          before_enter_var := p; {need to know which variable preceeds}
          p :=p^.next;                   {the entering variable.}
        end;
    end;

  LP_solution_found := (p=nil);

  enterpointer := p;

end; {of the procedure}

Procedure Step3(var D,enter  :vector);

{This procedure calculates step 3 of the revised simplex method}
          {which determins the D vector.}

var sum   :integer;
    i,j   :player;

begin
  for i := 1 to basic_variables do
    begin
      sum := 0;
      for j := 1 to basic_variables do
        begin
          sum:=sum + (AB_inverse.num[i,j] * enter[j]);
          {find product of entering col and basic matrix inverse}
        end;
      D[i] := sum;  {store the answer product in D vector}
    end;
end;  {of this procedure}

Procedure Step4(var exit_index  :player;
                    D           :vector);

   {This procedure calculates step 4 of the revised simplex method}
             {which determines the exiting variable.}

var t,temp   : real;
    i        : player;
    D_i      : real;

begin

  t:=-1;   {signal first ratio computation}

  for i:=(Free.rank + 1) to basic_variables do
             {for each element in the vector}
                  {that is not free}
    begin
      D_i := D[i]/AB_inverse.den;
      If D_i > 0 then
      begin
       temp:= basic_value[i]/D[i];

       if (temp < t) or (t = -1) then    {if the temp is}
         begin                          {less than current t then}
           t:= temp;               {put the smaller temp value in t}
           exit_index:=i;  {and mark that element as exiting variable}
         end;
      end;
     end;
end;  {of this procedure}

Procedure step5(var before_enter_var :dual_variable_pointer;
                var enterpointer     :dual_variable_pointer;
                var exit_index       :player;
                var D                :vector);

   {This procedure calculates step 5 of the revised simplex method}
    {which updates the basic matrix and the dual variable values.}

var i,j,m     :player;
    temp      :dual_variable_pointer;
    D_i,D_m   :integer;

begin
  m:=exit_index;                               {indexes pivot row}
  D_m := D[m];
  for i := 1 to basic_variables do {until the last row of the matrix:}
   begin
      D_i := D[i];
      if i<> m then                           {skip the pivot row}
       begin
        basic_value[i] := ((D_m*basic_value[i])-
                              (D_i*basic_value[m])) div AB_inverse.den;
```

```
              {pivot affects dual_variable values.}

        for j := 1 to basic_variables do
          begin
            AB_inverse.num[i,j] := ((D_m*AB_inverse.num[i,j]) -
                                     (D_i*AB_inverse.num[m,j]))
                                      div AB_inverse.den;
              {pivot affects inverse matrix.}
          end;
       end;
    end;
   AB_inverse.den := D_m;                           {update denominators.}

   temp := basic[exit_index];                  {put entering variable into}
   basic[exit_index] := enterpointer;  {basic array and put exiting}
   If before_enter_var <> nil then
      before_enter_var^.next := temp          {variable into the}
   else
     nonbasic_head := temp;
   temp^.next := enterpointer^.next;      {nonbasic linked list.}

end; {of this procedure}

Procedure Solve_for_alpha;

   {this procedure calculates the value of alpha (the solution)}

var i   :integer;
    sum :real;

begin
  sum := 0;
  i   := 1;

  while i <= basic_variables do   {until the last variable,}

    begin  {sum the product of the obj coeff value}
                 {and dual variable value}
      sum := sum + (basic_value[i] * basic[i]^.obj);
      i   := i+1;
    end;

  alpha := sum / AB_inverse.den;  {this sum is the value of alpha}

end;  {of this procedure}

Procedure Solve_linear_program;

      {This procedure executes the revised simplex method.}

var  enterpointer,before_enter_var  :dual_variable_pointer;
     exit_index                     :player;
     enter, D                       :vector;
     LP_solution_found              :boolean;

Begin
  repeat
    step1;
    Write_Basis;
    step2(enter,before_enter_var,enterpointer,LP_solution_found);
    if not LP_solution_found then
      begin
        step3(D,enter);
        step4(exit_index,D);
        step5(before_enter_var,enterpointer,exit_index,D);
      end;
  until LP_solution_found;
  Alpha := x[basic_variables];
end;  {of this procedure}
```

COMPLEXITY OF THE LINEAR PROGRAMMING ALGORITHM

There are three multiplicative factors to consider when determining the complexity of the linear programming algorithm. These factors are the number of LP's to be solved, the number of pivots that need to be performed and the number of calculations within each pivot. Each factor is possibly exponential.

THEORETICAL COMPLEXITY:

Number of LP's that need to be solved $\geq 2^{n-1} - n + 1$

Proof: Let $x = (1,1,\dots,1)$ and define a game in the following way:

S	V(S)	e(X,S)
1	$-\alpha + 1$	$-\alpha$
2		
.	.	.
.	.	.
.	.	.
(n-2)	$-\alpha + 1$	
(n-1,n)	$-\alpha + 2$	$-\alpha$
K player coalitions that include both or neither of (n-1) and n	$-\alpha + 2K - n$	$-\alpha + K - n$
K player coalitions that include exactly one of n-1 or n	$-\alpha + 1 - n + K$	$-\alpha + 1 - n$

The game we define must be superadditive as that is required and x=(1,...1) will be the nucleolus . Choose the value of α so that the excesses of each of the 3 subgroups of coalitions differs by one unit. If strict superadditivity $(V(S) + V(T) < V(S \cup T))$ holds for this α, then subgroup 2 can be perturbed so that while x stays the same, there are different alphas for each coalition in this subgroup. There are eight cases to consider to test for strict superadditivity.

(Note: only the first and last lines are included. The steps taken to arrive at the result are omitted.)

1st case:

2 singletons (not n-1 or n)

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 1) + (-\alpha + 1) < -\alpha + 4 - n$$
$$n - 2 < \alpha$$

2nd case:

1 singleton (not n-1 or n) AND
1 K player coalition w/neither or both

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 1) + (-\alpha + 2K - n) < -\alpha + 2(K + 1) - n$$
$$-1 < \alpha$$

3rd case:

1 singleton (not n-1 or n) and (n-1 , n)

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 1) + (-\alpha + 2) < -\alpha + 6 - n$$
$$n - 3 < \alpha$$

4th case:

A K and an L coalition containing neither or both of n-1 and n

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 2K - n) + (-\alpha + 2L - n) < -\alpha + 2(K+L) - n$$
$$-n < \alpha$$

5th case:

A K and an L coalition that contain exactly one of n-1 or n

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 1 - n + K) + (-\alpha + 1 - n + L) < -\alpha + 2(K+L) - n$$
$$2-n-K-L < \alpha$$

6th case:

1 K coalition w/ neither n-1 nor n AND
1 L coalition w/ exactly one of n-1 or n

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 2K - n) + (-\alpha + 1 - n + L) < -\alpha + 1 - n + (K+L)$$
$$K - n < \alpha$$

7th case:

1 singleton (not n-1 or n) and
1 K coalition with exactly one of n-1 or n

$$V(S) + V(T) < V(SUT)$$
$$(-\alpha + 1) + (\alpha + 1 - n + k) < -\alpha + 1 - n + (K+1)$$
$$0 < \alpha$$

8th case:
The coalition (n-1,n) and one K coalition with neither n-1 nor n

$$\begin{array}{ccccc} V(S) & + & V(T) & < & V(S \cup T) \\ (-\alpha + 2) & + & (-\alpha + 2K - n) & < & (-\alpha + 2K + 2 - n) \end{array}$$
$$0 < \alpha$$

The Result: Choose α to be greater then $n - 2$ and strict superadditivity will hold.

Before a solution is found for this defined game, at least one LP for each subgroup must be solved. One LP for each (excess + a small value) in the second subgroup must also be solved. Therefore, the total number of LP's that must be solved is $2^{n-1} - n + 1$.

Proof:

Subgroup	Number of LP's to be solved
I	1
II	K-player coalitions with n-1 and n Total - grandcoalition - (n-1,n) 2^{n-2} - 1 - 1 K-player coalitions without n-1 and n Total - singletons - empty set 2^{n-2} - (n - 2) - 1 Total this subgroup: $2^{n-1} - n - 1$
III	1

Total number of LP's to be solved = $2^{n-1} - n + 1$

This is the upper bound as only in the worst case would it be necessary to solve 1 LP for every coalition in subgroup II before the solution is found.

Number of pivots that must be performed = No worse than 2 raised to the n^2 power.

It is actually $\binom{2^n}{n}$ which, using Sterling's formula, is found to be less than 2 raised to the n^2. It is known that general LP's are exponential; it is not known whether the special structure of this class of LP's improves the bound.

Number of calculations within each pivot = $O(n2^n)$

Step 1	first number n * number of numbers n => $n*n = n^2$.
Step 2	$n * 2^n$ => $O(n2^n)$
Step 3	n^2
Step 4	n^2
Step 5	n^3

The number of calculations within each pivot is at least $O(n2^n)$.

COMPUTATIONAL COMPLEXITY:

The computational complexity was determined using random games. These games were generated in the following manner: V(i) = 0, V(N) = 1 and all other values are determined using a uniform distribution on (0,1). Ten examples were generated for each number of players.

Number of LP's Solved

Number of players	Average	High	Low	S.D.
3	2.8	3	2	.8944
4	3.8	5	2	1.3663
5	3.6	5	2	1.4491
7	4.2	6	2	1.8038
9	4.6	10	2	2.7019

On average, increases linearly or near linearly.

Number of Pivots Performed

Number of players	Average	High	Low	S.D.
3	4.4	6	3	1.7889
4	17.2	24	9	9.5149
5	26.1	37	16	11.1232
7	79.3	123	43	28.9485
9	171.6	195	134	17.3000

On average, increases exponentially.

CPU Time (in minutes)

Number of players	Average	High	Low	S.D.
3	.12	.20	.10	.0894
4	.73	1.00	.40	.3834
5	1.91	2.70	1.20	.8272
7	17.42	25.80	10.00	5.8628
9	143.02	179.10	122.90	19.6779

On average, increases close to an order of magnitude.

Note that experimentally, the number of LP's to be solved is, on average increasing linearly or near linearly while theoretically, it is shown to increase at least exponentially. The experimental data for the number of pivots shows that the number increases exponentially which shows that the special structure of this class of LP's does not improve the bound. Finally, the CPU time which is a product of the three theoretical values is exponential. This picks up on the exponential number of LP's, pivots and calculations within each pivot determined theoretically.

SOLVING FOR THE NUCLEOLUS USING THE RECURSIVE ALGORITHM

Description of the method:

Definitions:

Efficient- An allocation method is efficient if all possible value is allocated, $\sum_{j \in N} \theta_j(N,v) = v(N)$.

Symmetric- An allocation method is symmetric if for all permutations π of N and for all players $i \in N$, it follows that $\theta_{\pi(i)}(N,\pi v) = \theta_i(N,v)$ where πv is the value function defined by $\pi v(S) = v(S)$ for all $S \subseteq N$.

Covariant- An allocation method is covariant if for any $\alpha = 0$ and $b \in R^n$, it follows that $\theta(N,u) = \theta_i(N,v) + b$ where u is defined by $u(S) = \theta v(S) + \sum_{i \in S} b_i$ for all $S \subseteq N$.

Sobolev Consistent- An allocation method is Sobolev consistent if $\theta_i(N,v_T,\theta_{(N,v)})$ for all $i \in T$ and T N, where $(T,v_{T,x})$ is the reduced game by $v_{T,x}(T) = \sum_{i \in T} x_i$, and $V_{T,x}(S) = \max_{R \subseteq N-T} \{v(SUR) - \sum x_i\}$ for $S \subset T$.

The recursive algorithm is based upon Sobolev's theorem which states that the Nucleolus is the unique allocation method that is efficient, symmetric, covariant and consistent. Stearns shows that reducing a given game to 2-player games and calculating the nucleolus on each 2-player game will converge to a point in the kernel but not necessarily at the nucleolus. This approach is attractive computationally because the nucleolus for a two person game is easily computed:

$$x_i(N,v) = v(i) + \tfrac{1}{2}[v(12) - v(1) - v(2)] \text{ for } i = 1,2.$$

The recursive algorithm conjecture is that if the nucleolus is calculated on all subsets of a game (ie 3 or 4-player games) we will converge to a smaller subset of the kernel than Stearns or to the nucleolus.

ILLUSTRATION OF STEARNS METHOD BY EXAMPLE:

Consider Shubik's example

$v(i) = 0$
$v(12) = v(34) = 0$ otherwise, $v(ij) = 2$
$v(ijk) = 2$
$v(N) = 4$

The nucleolus is (1,1,1,1) the kernel is $\{(t,t,2-t,2-t) : t=[0,2]\}$
Execute Stearns method:
Choose $x = (2,0,1.5,.5)$
coalition {12}

1. $v(12) = 2$ for the next x,
$v(1) = 1.5$ $x_1=x_2= 1.5 + \frac{1}{2}[2-1.5-1.5] = 1$
$v(2) = 1.5$

$x = (1,1,1.5,.5)$
coalition {34}

2. $v(34) = 2$
$v(3) = 1$ for the next x,
$v(4) = 1$ $x_3=x_4= 1 + \frac{1}{2}[2-1-1] = 1$

3. $x = (1,1,1,1)$ we found the nucleolus.

However, Stearns method doesn't always converge to the nucleolus. The coalitions were chosen in this example in the "right" order so the nucleolus was found. If, in step 2, the coalition {23} were chosen, the following would have occured:

$x = (1,1,1.5,.5)$

2. coalition {23}
$v(23) = 2.5$ for the next x,
$v(2) = 1.5$ $x_2 = 1.5 + \frac{1}{2}[2.5-1.5-1] = 1.5$
$v(3) = 1$ $x_3 = 1 + \frac{1}{2}[2.5-1.5-1] = 1$

$x = (1,1.5,1,.5)$

3. coalition {12}
$v(12) = 2.5$
$v(1) = 1.5$ for the next x,
$v(2) = 1.5$ $x_1=x_2= 1.5 + \frac{1}{2}[2.5-1.5-1.5] = 1.25$

$x = (1.25,1.25,1,.5)$

4. coalition {34}
$v(34) = 1.5$
$v(3) = .75$ for the next x,
$v(4) = .75$ $x_3=x_4= .75 + \frac{1}{2}[1.5-.75-.75] = .75$

5. $x = (1.25,1.25,.75,.75)$
This is a point in the kernel. The values of x won't change by considering any other two player coalition so we can stop.

EXPLANATION OF WHY THE METHOD WON'T WORK:

Again, consider Shubik's example:

Choose $x = (2,0,1.5,.5)$

1. coalition {234}

$v(234) = 2$
$v(23) = 2$
$v(24) = 2$
$v(34) = 0$
$v(2) = 0$
$v(3) = 0$
$v(4) = 0$

2. $x = (2,2,0,0)$

This is a point in the kernel. The values of x won't change by considering any other three player coalition so we can stop.

Note that we haven't reached the nucleolus using 3-player coalitions and we haven't reached a much better solution while the difficulty of calculating the nucleolus has increased. If 4-player coalitions were used, calculating the nucleolus would be even more difficult. We could break the coalition into two 2-player coalitions or use the linear programming method to calculate the nucleolus on the reduced game, but both defeat the purpose. Therefore, calculating the nucleolus on all subsets of a coalition will not, in all cases, yield a better approach.

RESULTS AND QUESTIONS FOR FURTHER STUDY

The pascal implementation has made calculating the nucleolus on n-player games very quick and relatively effortless for nine or less players. Perhaps it would be possible to rework the program to consider larger games. However, due to limits on number size in pascal, this may not be possible.

The experimental data for the number of LP's that must be solved indicates a linear or near linear increase. This leads one to conjecture that a "better" upper bound might be found if the question were looked into further. The theoretical upper bound determined in this paper is exponential.

REFERENCES

Vasek Chvátal, Linear Programming. W.H. Freeman and Company, New York, 1983.

James Endersby, "Toward Efficiency in Linear Programming". From the Journal of Undergraduate Mathematics, March 1983, Volume 15, No. 1.

David Housman and Lori Jew, "Cooperative Game Theory: An introduction", Worcester Polytechnic Institute, Worcester, Massachusetts, 1989.

Guillermo Owen, Game Theory, 2nd ed. Academic Press, Inc., New York, 1982.

H. Peyton Young, "Cost Allocation", Proceedings of Synposia in Applied Mathematics, Volume 33, 1985.