Enumeration of All Extreme Equilibria of Bimatrix Games with Integer Pivoting and Improved Degeneracy Check*

Gabriel D. Rosenberg

Department of Mathematics, London School of Economics and Political Science
Houghton St, London WC2A 2AE, United Kingdom
email: grosenberg@gmail.com

CDAM Research Report LSE-CDAM-2004-18

December 7, 2005

Abstract: The "Enumeration of All Extreme Equilibria of Bimatrix Games" algorithm of Audet et al. (2001) uses the best response condition of Nash Equilibria to create a search tree in which pure strategies are forced to be either a best response or played with zero probability. Finding sets of constraints with no feasible solution allows the algorithm to avoid searching all game supports and thereby speeds the enumeration process. This paper presents two new improvements to the EEE algorithm. First, the algorithm is implemented in Java using only integer arithmetic, as opposed to previous implementation using floating-point arithmetic. This exact solution of linear programs for the algorithm avoids potential rounding errors. Preliminary running time results of this implementation, determining the relative efficacy of objective functions for the linear program search, and a comparison to another enumeration algorithm are reported. Second, the degeneracy check is improved, drastically cutting running time for certain classes of games and making the algorithm theoretically clearer. The new approach introduces constraints until the feasible set consists of only one strategy or is empty. The combination of these two improvements increases EEE's usefulness as a tool for bimatrix game solution.

Note (October 2009): The comparison with the original algorithm EEE-o in this paper mistakenly refers to a version which is different from, and behaves worse than, that of Audet et al. (2001). The Audet et al algorithm pivots only on positive variables and has running times that are comparable with EEE-I. As a result, the time and node comparisons on p. 43 of this paper, for example, are not valid, though the theoretical improvements still hold. For a better description of the Audet et al algorithm and an extension of the present results see David Avis, Gabriel D. Rosenberg, Rahul Savani and Bernhard von Stengel, Enumeration of Nash Equilibria for Two-Player Games, Economic Theory 42 (2010).

* Updated version of dissertation for M.Sc. degree in Applicable Mathematics, September, 2005. Winner of the 2005 Haya Freedman Prize for best M.Sc. dissertation in the Department of Mathematics.

Contents

Acknowledgements

An enormous thank you is due to my advisor, Dr. Bernhard von Stengel of the Department of Mathematics at the London School of Economics, for his incredible help with this paper. The paper is a direct result of the immeasurable time he spent helping me understand the subject matter and providing insight into the problem and its solutions. My introduction under his watch to the world of mathematical research made this project the highlight of my academic year. Thank you also to Rahul Savani of LSE for his help in answering my questions relating to the subject matter.

I had the fortunate opportunity to meet and discuss this work with several of the authors of related papers. Thank you to David Avis of McGill University and Charles Audet and Pierre Hansen of the Groupe d'études et de Recherche en Analyse des Décisions. These group discussions contributed greatly to portions of the paper, including clarifying various choices for the objective function and understanding the problem of degeneracy.

Lastly, I would like to thank the entire Department of Mathematics at LSE for a fantastic inaugural year of the MSc program in Applicable Mathematics.

1 Bimatrix Games and Nash Equilibria

A bimatrix game is a two-player, normal-form game in which the matrices A and B hold the payoffs to players I and II, respectively. In a normal-form game, each player simultaneously chooses an action either deterministically (a pure strategy) or through a probability distribution over all possible actions (a mixed strategy). Payoffs are described by a matrix for each player in which the entry at position (i, j) is the payoff when Player I chooses action i and Player II chooses action j. Player I's pure strategies are represented by the rows of the matrix and Player II's by the columns; the pure strategies of Player I are described as belonging to the set $M = 1, 2, \ldots m$ and those of Player II as belonging to the set $N = 1, 2, \ldots n$. The sets M and N need not be of equal size but, by definition of the payoff structure, both matrices A and B must be of the same size.

Player I's choice of strategy, pure or mixed, can be described as a probability vector over the elements of M; the space of all such probability vectors is denoted X and any particular vector as an $x \in X$. A pure strategy is an $x \in X$ such that one element of x has value 1 and the rest have value 0. Similarly, Player II's space of strategy vectors is denoted Y with any particular choice $y \in Y$. Both x and y are denoted as column vectors; the former of size $m \times 1$ and the latter of size $n \times 1$.

The choice of a particular $x \in X$ for Player I and $y \in Y$ for Player II defines an expected payoff for each of the players. Player I chooses a pure strategy in M based on the probabilities assigned in x, Player II chooses a pure strategy in N based on the probabilities assigned in y. While the actual payoff to Player I is an element of A and that to Player II is the corresponding element of B, the expected payoff is the weighted average of the payoffs of all possible outcomes, based on the probability of occurring. The probability of any particular payoff A_{ij} to Player I is just the probability that Player I plays i and Player II plays j: $x_i y_j$. Therefore, the expected payoff to Player I is the probability of each payoff occurring multiplied by its value, or $x^\top Ay$. Player II's expected payoff is $x^\top By$.

In all the bimatrix games that follow, the assumption is made that each player seeks to maximize his or her own expected payoff without regard to the other player's expected payoff. Both agents are risk-neutral. The payoffs in the matrices are meant to already reflect any risk preferences the players may have over the payoffs. Much of this description of bimatrix games and Nash Equilibria follows the survey by von Stengel (2002).

A Nash Equilibrium (NE) is a pair of strategies for which both players maximize expected payoff given the strategy of the other player. An NE in a bimatrix game is a pair of strategies (x^*, y^*) such that no other $x \in X$ provides a higher expected payoff

for Player I than x^* given that Player II's strategy is y^*, and no $y \in Y$ provides a higher expected payoff for than y^* for Player II given that Player I's strategy is x^*. Given the strategy of the other player neither wishes to change his strategy. Nash proved that every bimatrix game has at least one such equilibrium.

Nash Equilibria may share a strategy of either Player I or II while the strategy of the other player varies. Given NE pairs $(\hat{x}, \hat{y})$ and $(\hat{x}, \hat{y}')$, all linear combinations of the form $(\hat{x}, \alpha\hat{y} + (1-\alpha)\hat{y}') \quad \forall \alpha \in [0,1]$ are also NE. These equilibria are known as degenerate equilibria; bimatrix games containing degenerate equilibria are referred to as degenerate games.

Determining whether a point is a NE is easy and requires only checking what is known as the best response condition: at a NE, the distributions x^* and y^* must play with non-zero probability only those pure strategies which are best responses to the other player's mixed strategy. A given y^* determines Player I's payoff vector Ay^*. Each entry $(Ay^*)_i$ is the payoff to Player I of choosing strategy $i \in M$. Any probability Player I places on a strategy corresponding to a non-maximal element of Ay^* could be used to further increase the expected payoff through a shift to a best response. More formally, the best response condition states that (x^*, y^*) is a Nash Equilibrium if and only if $x_{\hat{i}} > 0$ implies that $\hat{i} = \arg\max_{i \in M}(Ay^*)_i$ and $y_{\hat{j}} > 0$ implies that $\hat{j} = \arg\max_{j \in N}(x^{*\top}B)_j$ for all $i \in M$ and $j \in N$.

The best response condition of NE can also be described as a set of equations that must hold for a pair of strategies (x^*, y^*) to be a NE. Define α to be the maximum payoff to Player I given y (the maximum of the vector Ay) and β to be the maximum payoff to Player II given x (maximum of the vector $x^\top B$). Then, since each strategy must be either played with zero probability or be a best response to the other player's strategy, each $i \in M$ and $j \in N$ must satisfy the equations $x_i(\alpha - (Ay)_i) = 0$ or $y_j(\beta - (x^{*\top}B)_j) = 0$, respectively.

1.1 Linear Programming

A player's best response to the chosen strategy of the other player is the vector $x \in X$ or $y \in Y$ that maximizes his expected payoff. As such, computing a best response falls into a class of problems known as linear programs; a linear program seeks to maximize a given linear function of variables subject to a number of linear constraints on those variables. This discussion of linear programming follows Chvátal's well-known 1983 text, incorporating bimatrix game issues from von Stengel's survey. Given $1 \times k$ vectors f and b of constants and a $g \times k$ constraint matrix C, a linear program is the problem of maximizing the product $z = f \cdot v$ subject to $C \cdot v \leq b$ by choosing a $k \times 1$ vector v. The function z is called the objective function and any vector v that satisfies $C \cdot v \leq b$

is said to be a feasible solution of the linear program. A solution to the linear program is a feasible solution that maximizes the objective function.

Calculating a best response corresponds to solving the linear program whose objective function is the expected payoff and whose constraint is that the variables are nonnegative and sum to 1. For Player I, the linear program is

$$\begin{aligned} \max_{x} \; & z = x^\top A y \\ \text{s.t.} \; & x \cdot \mathbf{1} = 1 \\ & x_i \geq 0 \;\; \forall i \in M \end{aligned}$$

and similarly for Player II

$$\begin{aligned} \max_{y} \; & z = x^\top B y \\ \text{s.t.} \; & \mathbf{1} \cdot y = 1 \\ & y_j \geq 0 \;\; \forall j \in N. \end{aligned}$$

where $\mathbf{1}$ is a row vector of ones such that the product involved is scalar.

At a Nash Equilibrium, both players must play best responses to each others' strategies; the vector y used in Player I's objective function must be the solution to Player II's linear program and vice versa.

Consider the linear program

$$\begin{aligned} \max_{x_1, x_2} \; & z = 7x_1 + 3x_2 \\ \text{s.t.} \; & 3x_1 + x_2 \leq 11 \\ & x_1 + 2x_2 \leq 7 \\ & x_1, x_2 \geq 0 \end{aligned}$$

This can be easily altered to equality form in the constraints by introducing two slack variables, r_1 and r_2, which take a value such that the two numerical constraints are tight. Following the convention of placing all variables on one side of the equation, subtract the objective function from z and require that it equal 0; in essence the goal is to maximize z subject to minimizing the opposite of the sum of the variables.

$$\begin{aligned} \max_{x_1, x_2} \; & z = 7x_1 + 3x_2 \\ \text{s.t.} \; & 3x_1 + x_2 + r_1 \leq 11 \\ & x_1 + 2x_2 + r_2 \leq 7 \\ & x_1, x_2, r_1, r_2 \geq 0 \end{aligned} \quad = \quad \begin{aligned} \min_{x_1, x_2} \; & -7x_1 - 3x_2 \\ \text{s.t.} \; & 3x_1 + x_2 + r_1 \leq 11 \\ & x_1 + 2x_2 + r_2 \leq 7 \\ & x_1, x_2, r_1, r_2 \geq 0 \end{aligned}$$

Representing this system in matrix form yields

$$\max_{x_1,x_2,r_1,r_2} \begin{pmatrix} 1 & -7 & -3 & 0 & 0 \end{pmatrix} \begin{pmatrix} z & x_1 & x_2 & r_1 & r_2 \end{pmatrix}^\top$$

$$\text{s.t.} \quad \begin{pmatrix} 3 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 & x_2 & r_1 & r_2 \end{pmatrix}^\top = \begin{pmatrix} 11 \\ 7 \end{pmatrix}$$

It is usual to concatenate this into a tableau, of which one type is of the form

$$\left(\begin{array}{c|c} C & b \\ \hline -f & z \end{array}\right)$$

where C is the coefficients of the C matrix, b is the vector of constants, f is the coefficients of the objective function, and z is the value of the objective function ($f \cdot x$ for the particular values of x in question. Here, this matrix is initially

$$\begin{pmatrix} 3 & 1 & 1 & 0 & 11 \\ 1 & 2 & 0 & 1 & 7 \\ -7 & -3 & 0 & 0 & 0 \end{pmatrix}.$$

The solution of linear programs involves a partition of the choice variables v into those that can take non-zero values, known as the basis, and those which are set to zero, known as the cobasis. Each constraint is expressed such that each basic variable is present in one, and only one, row of the tableau. As a result, the cobasic variables serve as a representation of the basic variables. With this formulation the value of any basic variable is just the value of b in that column, which is the maximum value that the variable can take and still have the constraint satisfied.

It follows that the basic variables constitute an identity submatrix of the tableau. In the example above, the rows corresponding to r_1 and r_2 are an identity matrix; thus the basis here is r_1 and r_2. The variables x_1 and x_2 constitute the cobasis.

The simplex algorithm solves linear programming problems by pivoting elements between the basis and cobasis in an attempt to maximize the given function. At each step, the algorithm chooses the member of the cobasis that can contribute most to the objective function, the variable with largest (in absolute value) negative coefficient in the objective function row of the tableau. The variable to exit the basis is chosen by determining the strictest bound on the value of that variable; this minimum ratio test is meant to ensure that pivoting does not render a choice of v which is infeasible. The tableau is renormalized to assure the basis constitutes an identity submatrix. The algorithm repeats until there are no positive coefficients left in the objective function row;

at this point making the value of any cobasic variable non-zero (i.e. moving it into the basis) would result in a lower value of the objective function. Thus, a maximum has been reached.

An important theorem in linear programming, the duality theorem, involves solving a linear program by minimizing an upper bound on its objective function. Recall the original form of the linear program from above

$$\begin{aligned} \max_{x_1,x_2} \ & z = 7x_1 + 3x_2 \\ \text{s.t.} \ & 3x_1 + x_2 \leq 11 \\ & x_1 + 2x_2 \leq 7 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Adding twice the first constraint to the second constraint, yields

$$7x_1 + 4x_2 \leq 29 \rightarrow z = 7x_1 + 3x_2 \leq 29$$

which provides an upper bound on the objective function. The duality theorem states that minimizing such an upper bound yields the same result as maximizing the function. The dual of the original linear program is the problem of minimizing this upper bound. The constraints now require that the linear combination of the constraints has each coefficient greater than or equal to the coefficient in the objective function.

The minimized upper bound can be interpreted as the payoff to a player in a bimatrix game, given the formulation of the game in linear programming form. For the remainder of the paper, α will denote the dual variable for Player I (his payoff) while β will denote that for Player II (her payoff).

1.2 Nash Equilibria - Geometrically

While Nash Equilibria are often considered in the algebraic context discussed above, much attention has been recently given to its geometric interpretation (see von Stengel, 2002). A player choosing a probability distribution over h pure strategies chooses a point in h dimensional space subject to the constraint that the sum of the coordinates is 1. As a result, he can choose any point on a $h-1$ dimensional closed set (a polytope) within $\mathbb{R}^h$. Player II, in a normal form game, chooses a point $y \in Y$ on a $n-1$ dimensional polytope in $\mathbb{R}^n$. The set Y is this $n-1$ dimensional polytope; it is the union of all points in $\mathbb{R}^n$ such that the sum of the coordinates is exactly one. For every such point $y \in Y$, each pure strategy $i \in M$ has a determined payoff, which is $(Ay)_i$. The pure strategies with the highest such payoffs are the best responses to y.

Best response diagrams are often employed in the geometric view of bimatrix games

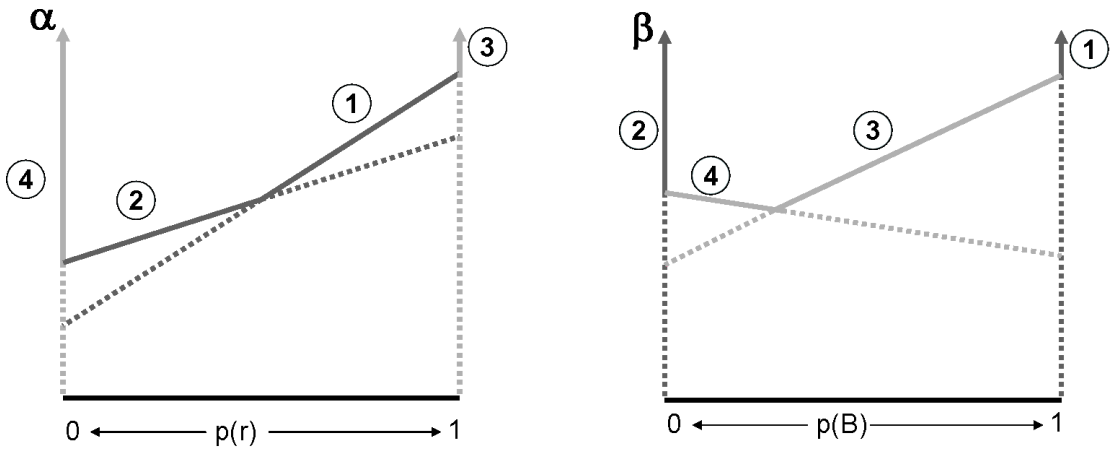


Figure 1: Best response diagrams for Player I (left) and Player II (right)

of small dimension. Consider the bimatrix game

$$A = \begin{pmatrix} 1 & 5 \\ 2 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ 5 & 2 \end{pmatrix}$$

where Player I's strategies are T (top) and B (bottom) and Player II's strategies are l (left) and r (right). Player I's payoffs from playing T or B can be plotted against Player II's strategy $y \in Y$, which is determined by a single value p, the probability of playing the strategy r. Player II's choice lies along a line segment from $p = 0$ to $p = 1$. The payoff to Player I's pure strategy T varies linearly from A_{11} to A_{12} as p changes from 0 to 1, and is therefore a line segment in $\mathbb{R}^2$. The same is true for the bottom strategy. The left plot in Figure 1 results, in which the dark gray line segments represent Player II's pure strategies $p = 0$ and $p = 1$, while the light gray line segments represent Player I's pure strategies T and B.

Given any particular choice of y, Player I should choose to play only those pure strategies that provide the highest payoff. This is a restatement of the best response condition. The geometric realization of the best response condition is the $n - 1$ dimensional "upper envelope," which in a 2×2 game is the union of all line segments that are above all other line segments. In Figure 1, the upper envelope consists of the solid light gray and dark gray line segments; the points on the best response curve for B from $p = 0$ to $p = \frac{1}{2}$, and of the points on the best response curve for T from $p = \frac{1}{2}$ to $p = 1$ along with the points corresponding to $p = 0$ and $p = 1$.

Since the upper envelope is the maximal payoff for any pure strategy x given a strategy y of Player II, it is the value of α, the dual variable in the linear programming solution for best responses. Given a specific y, using the simplex method to find the value of α is equivalent to finding the point on the upper envelope corresponding to

y. This is perhaps a more intuitive explanation of how using the dual variable as the objective function helps solve for a player's best response given the strategy of the other player.

By the best response condition, NE only exist on the upper envelope, as any point on a best response curve below the upper envelope is suboptimal. One implication is that any mixed equilibrium is a mix of two strategies that intersect to make a point on the upper envelope. In Figure 1, a mixed strategy is only possible (but not necessarily present) in a NE where $p = \frac{1}{2}$. At any other value of p, one strategy is better than the other, and therefore that strategy must be played with probability 1. The right plot in Figure 1 is the best response diagram for Player II's choice of l or r as Player I's probability of playing B changes from $q = 0$ to $q = 1$. The only possible mixed strategy for x in equilibrium is at the point where $x = (\frac{3}{4}, \frac{1}{4})$.

Nash Equilibria, in the geometric approach, are pairs of points at which every strategy $i \in M$ or $j \in N$ is either a best response to the strategy of the other player or is played with zero probability. A strategy is a best response where the point chosen on the player's best response diagram is on the portion of the upper envelope comprised by that strategy. A strategy is played with zero probability where the point chosen on the *other* player's best response diagram is on the polytope where that strategy's probability is zero; in these 2×2 games these are the $p = 0$, $p = 1$, $q = 0$, and $q = 1$ line segments parallel to the y-axis. Thus, for a pair of points to be a NE, every strategy $i \in M$ and $j \in N$ must fulfill the best response condition in one diagram or the other. In Figure 2, the best response condition is fulfilled for $i = 2$ and $j = 1$ in the left plot as the point is on the segment $p(y_1) = 0$ and the portion of the response curve for x_2 on the upper envelope. In the right plot, the best response condition is fulfilled for $i = 1$ and $j = 2$ where $p(x_1) = 0$ and y_2 is a best response to x. Since all 4 best response conditions are true for this set of points, it is a Nash Equilibrium.

The conventional way, as discussed in von Stengel (2002) referencing the work of Shapley, to represent such diagrams is by labeling each polytope with a strategy number. A certain point is labeled with $i \in M$ if that point is either on the upper envelope of the best response curve comprised by i in Player I's best response diagram, or if i is played with zero probability in Player II's best response diagram. The strategies are often labeled continuously, so Player I's strategies are labeled ① and ② while Player II's are labeled ③ and ④ .

A pair of points is a NE if it is completely labeled, that is it has all labels $i \in M$ and $j \in N$. This is just a restatement of the linear complementarity condition; a completely labeled pair of points has either $x_i = 0 / y_j = 0$ or $\alpha = (Ay)_i / \beta = (x^\top B)_j$ for all $i \in M$ and $j \in N$. This allows identification of Nash Equilibrium points by inspection for 2×2 games. In Figure 3, the three pairs of same-colored points are Nash Equilibria.

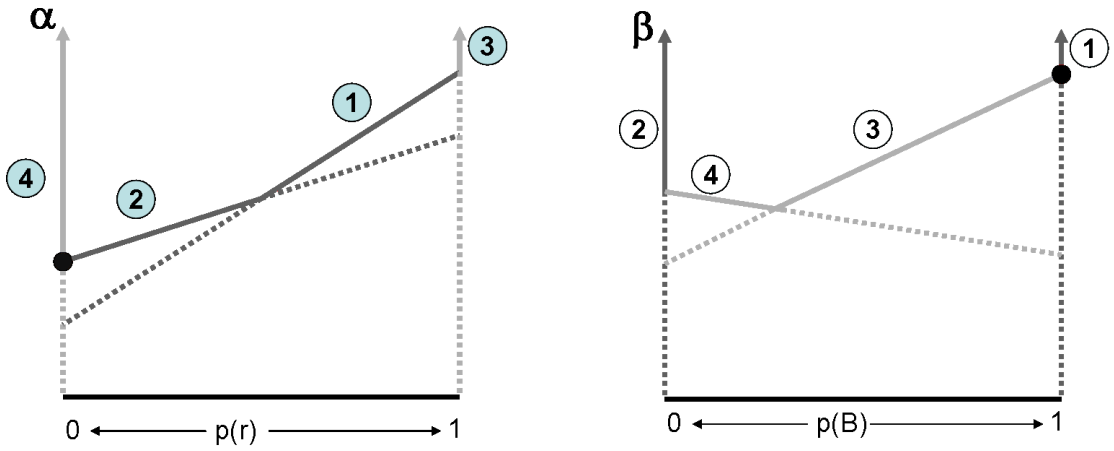


Figure 2: A Nash Equilibrium

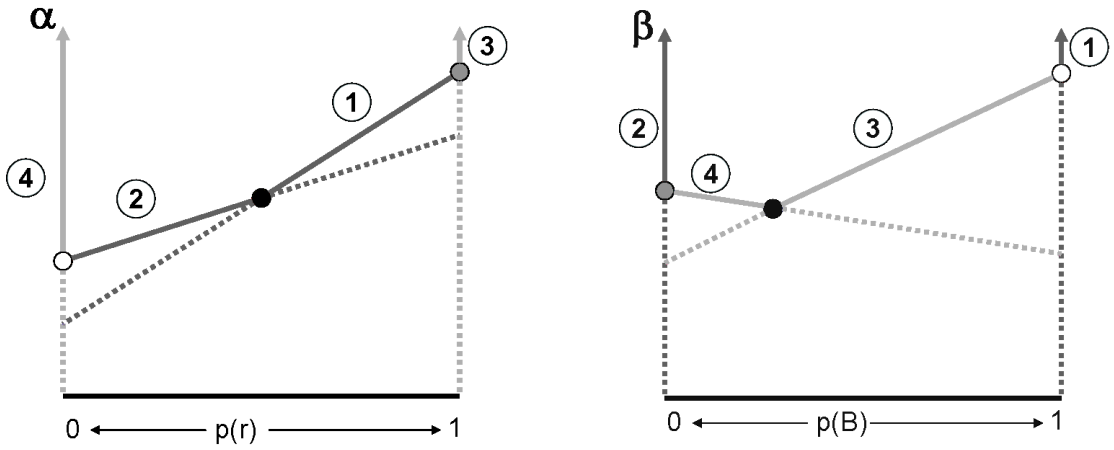


Figure 3: A labeled best response diagram and its three Nash Equilibria

2 The EEE Algorithm

The best response condition and its geometric analog provide an easy way of determining whether a given pair of strategies is a Nash Equilibrium of a bimatrix game. Finding NE given the game remains a more difficult problem. A variety of algorithms have been developed to solve problems in this vein. To find one NE of a given bimatrix game, the Lemke-Howson algorithm is often employed. This algorithm relaxes the best response condition for one pure strategy and searches the feasible set of points until that condition is once again satisfied. In its geometric form, the algorithm begins at the completely labeled (but non equilibrium) point where all strategies are played with zero probability. Allowing one strategy's label to be missing allows movement along the best-response polytope, and at each vertex some label is doubly introduced. Moving away from the double label provides a path that provably ends with the originally dropped label being reintroduced, thus landing at a Nash Equilibrium. The survey by von Stengel (2002) provides a more thorough introduction to Lemke-Howson.

The Lemke-Howson algorithm only finds one NE of a bimatrix game; however in many cases the question of enumerating all NE is relevant. Several algorithms have been proposed for solving this problem; it remains very much an active area of computational game theory. In particular, these algorithms attempt to find all extreme equilibria, which are all Nash Equilibria that are non-degenerate and the endpoints of degenerate equilibria. With such a set of extreme equilibria, all equilibria are described explicitly in the set or as a convex combination of interchangable extreme equilibria (see von Stengel (2002), Theorem 2.14). The equilibrium pair (x, y) and (x', y') are interchangable if and only if (x, y') and (x', y) are also equilibria; that is the equilibrium strategies within the set can be interchanged to form equilibria.

One enumeration algorithm uses polytope theory to enumerate all vertices of the best response polyhedra and then matches up vertices that together form the full set of best response labels. This algorithm has been implemented by Savani and a more recent version by Avis using Avis' (lexicographic reverse search) algorithm (2005) and will here be referred to as the LRS algorithm, though it should be clear that the LRS algorithm here refers to Avis' implementation incorporating both the lexicographic reverse search program and the equilibrium enumeration. Its efficiency in solving for all Nash Equilibria will be discussed later.

A 2001 paper by Audet, Hansen, Jaumard, and Savard provides a different approach to the problem of enumerating all NE in a bimatrix game. Their algorithm employs the best response condition of Nash Equilibria to restrict the set of possible $x \in X$ and $y \in Y$ until the only ones remaining are NE. Recall that for every $i \in M$, the best response condition states that $x_i(\alpha - (Ay)_i) = 0$, and therefore either $x_i = 0$ or

$\alpha = (Ay)_i$. Similarly, for every $j \in N$ either $y_j = 0$ or $\beta = (x^\top B)_j$. At a NE, both players play only best responses ($\alpha = (Ay)_i$ or $\beta = (x^\top B)_j$) to the others' move with positive probability. If a pure strategy is not a best response, it is played with zero probability ($x_i = 0$ or $y_j = 0$).

The EEE algorithm chooses some $i \in M$ or $j \in N$ and explores the implications of setting that to be a best response ($\alpha = (Ay)_i$ or $\beta = (x^\top B)_j$) or forcing it to be an unplayed strategy ($x_i = 0$ or $y_j = 0$). If the action of forcing these conditions does not exclude all possible solutions, another i or j is chosen and the process is repeated. Once all $m + n$ best response conditions are fulfilled, any point (x, y) that is still feasible is a NE. In essence, the algorithm chooses a given strategy and explores the hypothetical possibility that the strategy is a best response and separately the hypothetical possibility that the strategy is played with zero probability. Impossibilities are removed by checking for feasibility.

Treating the search for NE as a set of linear programs, the algorithm checks whether imposing certain strict equalities makes the linear program infeasible. For example, if x_1 is set to 0 to ensure that $x_1(\alpha - (Ay)_1) = 0$, a check must be performed to determine whether there exists a feasible solution to a linear program described by the constraints $\sum_{i \in M} x_i = 1, \alpha \geq (Ay)_i, x_1 = 0$. In a 2×2 game, forcing both x_1 and x_2 to equal 0 leaves no feasible solution, since the probability constraint $x_1 + x_2 = 1$ cannot be satisfied when $x_1 = x_2 = 0$.

The Audet et al. paper employs two linear programs, each of which incorporates aspects of both Player I's strategy and Player II's strategy. In particular let the two linear programs P and Q be defined by

$$
\begin{aligned}
P(x|y) \equiv \max_{x,\beta}\ & x^\top Ay - \beta \\
\text{s.t.}\ & \sum_{i \in M} x_i = 1 \\
& \beta \geq (x^\top B)_j,\ \forall j \in N \\
& x_i \geq 0,\ \forall i \in M
\end{aligned}
\qquad \text{and} \qquad
\begin{aligned}
Q(y|x) \equiv \max_{y,\alpha}\ & x^\top By - \alpha \\
\text{s.t.}\ & \sum_{j \in N} y_j = 1 \\
& \alpha \geq (Ay)_i,\ \forall i \in M \\
& y_j \geq 0,\ \forall j \in N.
\end{aligned}
$$

Each $i \in M$ and $j \in N$ corresponds to one constraint in $P(x|y)$ and one constraint in $Q(y|x)$. For $i \in M$ the P constraint is a constraint on the probability ($x_i \geq 0$) while for $j \in N$ it is a payoff constraint ($\alpha \geq (Ay)_i$). Similarly, the Q constraint for $i \in M$ is a constraint on the payoff ($\beta \geq (x^\top B)_j$) while for $j \in N$ it is a probability constraint ($y_j \geq 0$). It follows from the earlier discussion of best response constraints that an NE results from searching the feasible domains of these linear programs only if for each $i \in M$ either $P(x|y)$ is constrained to force $x_i = 0$ or $Q(y|x)$ is constrained to force $\alpha = (Ay)_i$, and correspondingly for all $j \in N$. The algorithm chooses, for each feasible

node, a pure strategy i or j which has not yet been forced to satisfy the best response constraint. The node then creates two children, one in which the inequality in P is made into an equality, and one in which the inequality in Q is made into an equality.

The algorithm starts with the linear programs P and Q from above each being constrained solely by their $m+n$ inequalities. For each node at depth $d \leq m+n$, there are $m+n-d$ best response conditions that are not yet forced. Any one of these $m+n-d$ unforced strategies can be tightened in a child node. The authors define a vector of size $m+n$ where all already forced components have value -1 and the unforced components have value $x_i(\alpha - (Ay)_i)$ for $i \in M$ and $y_j(\beta - (xB)_j)$ for $j \in N$. These values are simply the complementary slackness, a measure of how far the product is from the value it must take, 0. The variable corresponding to the largest such value is chosen and the tree branches off in two directions; in one, the variable is forced via its probability constraint and in the other by its best response constraint. Once the depth has reached $m+n$, feasibility implies that the branch of the tree being followed (strategy 1 is a best response, strategy 2 should be played with probability zero, etc.) defines a NE, since all best response conditions are fulfilled.

If a node is not feasible, it and its potential children cannot be NE and therefore it can be abandoned. This allows the algorithm to avoid searching all 2^{m+n} possible combinations of best-response/zero-probability pairs in most cases. Once the depth has reached $m+n$, all remaining nodes at that level are feasible values of x and y for which the best response conditions are all satisfied - therefore they are all NE.

In addition, there are a number of subproblems that may also be NE. By level $d = m+n$, all strategies are forced to be either a best response or played with zero probability. However, it is possible that a strategy is both a best response and played with zero probability in the case of degeneracies. In a Nash Equilibrium, a strategy must be a best response or played with zero probability, but those that are best responses also can be played with zero probability, provided there exists more than one. It is particularly all the inequalities that can still be tightened after the first $m+n$ steps. Every one of these that is found to be feasible is a NE, and its children are derived in the same way until there are no more nodes to look at in the queue. This "polychotomous branching" follows the "dichotomous branching" of the earlier steps (Audet et al. terminology).

Figure 4 shows a subset of a possible tree for a generic 2×2 game. The node numbered v is obtained by setting both x_1 and y_1 to 0. The algorithm chooses $i = 2$ as the next forced variable, and creates node v + 1. In this particular tree, that node is feasible. Its first child, created by forcing $y_2 = 0$ is not; if $y_1 = y_2 = 0$ then $y_1 + y_2 \neq 1$. The right child of v + 1, node v + 3, introduces the best response constraint for $j = 2$ by setting $\beta = (x^\top B)_2$. For this particular tree, the node is a Nash Equilibrium. The four

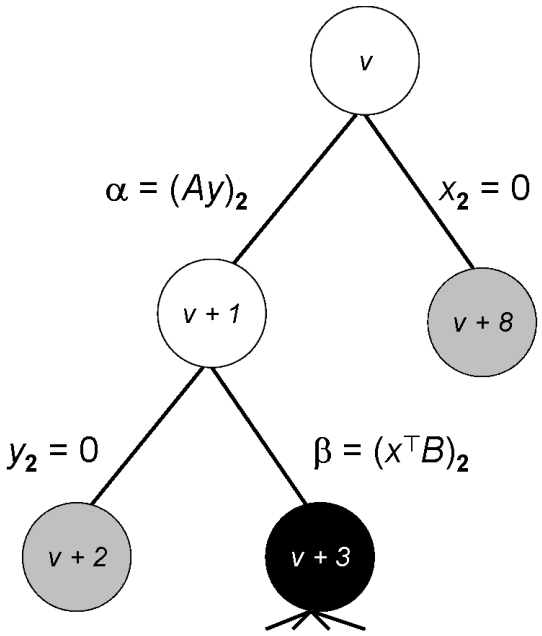


Figure 4: A possible subtree of a 2×2 game with $x_1 = y_1 = 0$

branches emanating from v + 3 represent the polychotomous branching step. After all polychotomous branches reach infeasibility (there are no new Nash Equilibria since this is a generic game) the algorithm backtracks to the right child of v, node v + 8. Since $x_1 = x_2 = 0$ implies $x_1 + x_2 \neq 1$, node v + 8 is infeasible and the subtree is finished.

The feasible set does not increase as the inequalities are forced. Either the set remains the same size, as in the case where a variable already equal to 0 is forced to be 0, or it decreases in size. The linear program is infeasible, at the very latest, at the point where all strategies for one player are forced to be played with zero probability. It is clear that all non-degenerate equilibria are found since all feasible combinations of played strategies (supports) are considered. Finding all extreme points of degenerate equilibria is trickier, and will be discussed more thoroughly in Section 6.

3 EEE Geometrically

While the authors describe the EEE algorithm using the algebraic equations of the best response conditions, the geometric discussion of Nash Equilibria presented earlier is enlightening. By requiring a variable $i \in M$ to satisfy $x_i = 0$ or $\alpha = (Ay)_i$, the EEE algorithm introduces the a new label each time the tree splits. The label is the same in both branches but is derived in the opposite best response diagram. In one branch, the label is added by restricting the space of possible equilibria to the polytope $x_i = 0$ in $P(x|y)$. In the other, it is added by restricting the space of possible equilibria to the polytope $\alpha = (Ay)_i$ in $Q(y|x)$, which is the portion of the larger dimensional best-

response polytope where the payoff to i is greater or equal to the payoff from any other pure strategy. The reverse holds for all $j \in N$. As a result, when $m+n$ distinct labels have been added, the remaining polytope is an extreme NE as it is guaranteed to have all $m+n$ labels. Infeasible solutions occur for a given linear program P or Q when the restrictions on the polytopes are such that no points fulfill all the requirements.

The algorithm works with two sets of points, $\hat{P}$ and $\hat{Q}$, which are the feasible solutions to linear programs P and Q respectively. At each step, a restriction is placed on the elements of $\hat{P}$ or $\hat{Q}$ and the algorithm terminates if that set becomes empty. The sets $\hat{P}$ and $\hat{Q}$, at depth d, are of the form

$$\hat{P} = \bigcap_{k=0}^{g} L_x(f(k)) \quad \text{and} \quad \hat{Q} = \bigcap_{p=0}^{h} L_y(f(p))$$

where $g + h = d$, and the set $L_x(f(i))$ is the set of all points with label $f(k)$. The initial constraints $L_x(0)$ and $L_y(0)$ maintain that the sum of the probabilities must be 1 and that the point must be on the upper envelope. The latter of these requirements is enforced by the linear programming objective functions, which for this example will be $P(x|y) = \max_{x,\beta} -\beta$ and $Q(y|x) = \max_{y,\alpha} -\alpha$. These objective functions find the lowest point on the feasible portion of the upper envelope.

Figure 5 shows the algorithm initialization for the 2×2 game from Section 1.2

$$A = \begin{pmatrix} 1 & 5 \\ 2 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ 5 & 2 \end{pmatrix}.$$

Q is solved first. The vector x is set to $(1,0)$, and y looks for the lowest point on the upper envelope (because of the form of the objective function), arriving at $y = (1,0)$. When P is solved, $x = (\frac{3}{4}, \frac{1}{4})$. This completes the linear program processing of the first node.

The second node (Node 1) is obtained by requiring that $\alpha = (Ay)_1$, meaning that Player I's first strategy is a best response. This, in effect, reduces the possibilities for Player II's strategies to those in which Player I's first pure strategy has a payoff greater than or equal to that of his second pure strategy. Graphically, the points in $\hat{Q}$ must be on the portion of the first pure strategy response curve where that curve constitutes the upper envelope. In Figure 6, the feasible region is the bolded line. Alternatively, the feasible region consists of all points in $\hat{Q}$ that have label ① . After linear programming pivoting, $y = (\frac{1}{2}, \frac{1}{2})$ as this is the lowest point on the upper envelope for x. Player I's response remains unchanged, as $x = (\frac{3}{4}, \frac{1}{4})$ remains the minimization point on the upper envelope of Player II.

The first node at depth 2 (Figure 7) results from adding the requirement that $\alpha =$

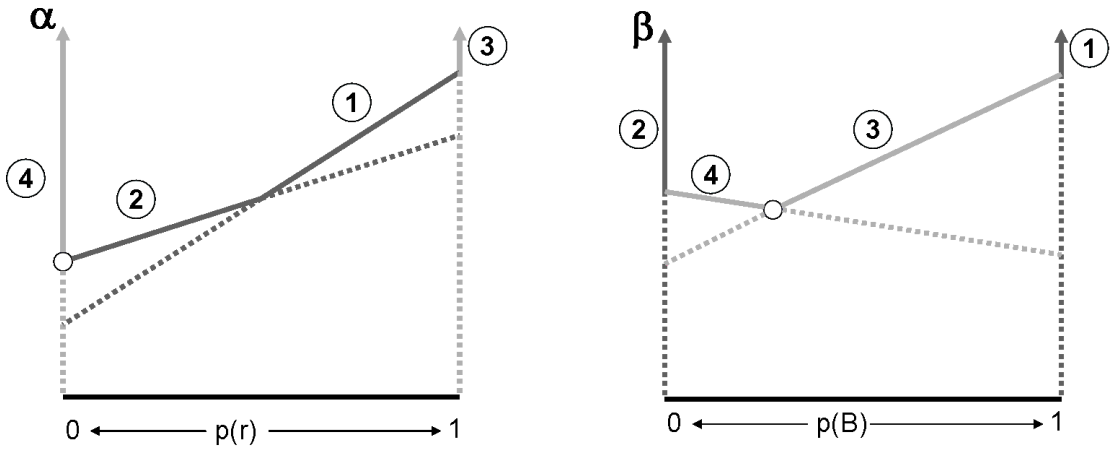


Figure 5: Node 0, Depth = 0

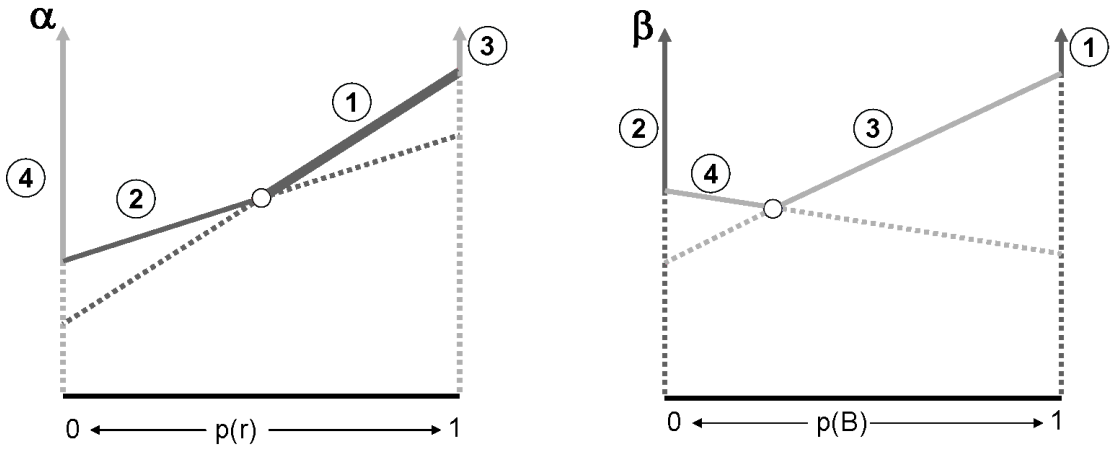


Figure 6: Node 1, Depth = 1

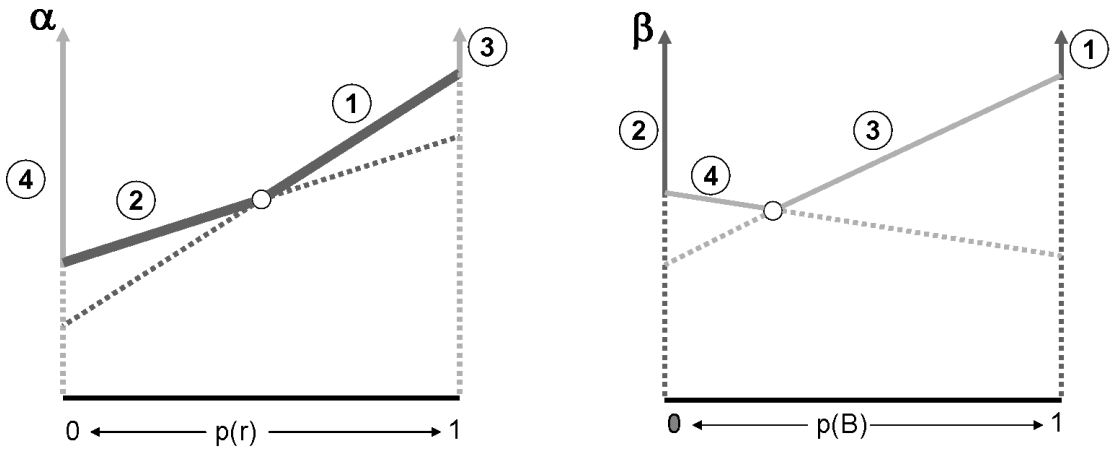


Figure 7: Node 2, Depth = 2

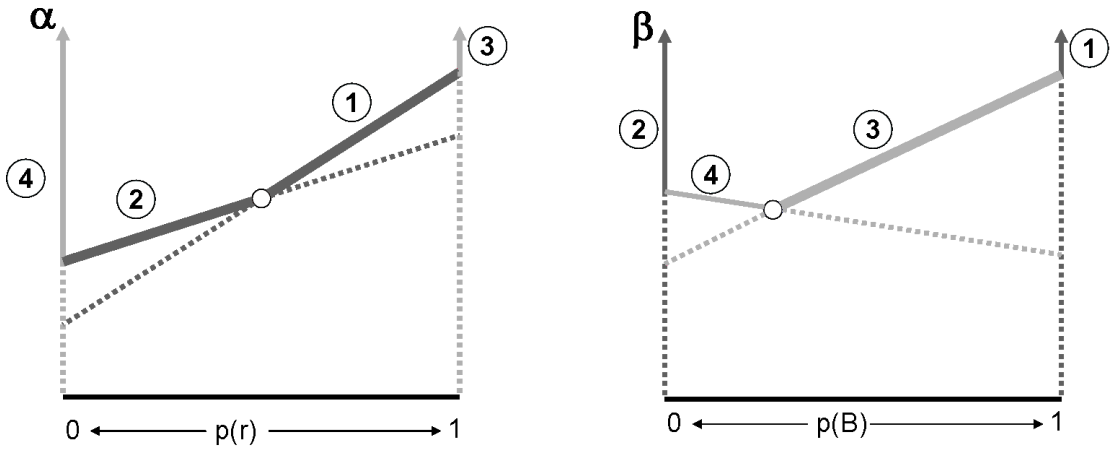


Figure 8: Node 3, Depth = 3

$(Ay)_2$. All points in the set $\hat{Q}$ must be on both best response curves for Player I as well as on the upper envelope. Only one point, $y = (\frac{1}{2}, \frac{1}{2})$ satisfies these criteria. I's response remains unchanged; $x = (\frac{3}{4}, \frac{1}{4})$. Label ② has now been introduced.

The set $\hat{P}$ is first restricted in the Node 3. Requiring that $\beta = (x^\top B)_1$ means that all points in the feasible set must be on the best response curve for Player II's left move. Figure 8 incorporates this restriction. The lowest point on I's best response curve remains $x = (\frac{3}{4}, \frac{1}{4})$. As $\hat{Q}$ is unchanged and consists of only one point, $y = (\frac{1}{2}, \frac{1}{2})$. $\hat{P}$ now consists of all best response points with label ③ .

The fifth node (Node 4) is the first in which every variable $i \in M$ and $j \in N$ is forced to obey the best response condition, meaning that introducing the label ④ means that all 4 labels are present, and any feasible point is an extreme Nash Equilibrium. The final restriction is that $\beta = (x^\top B)_2$. Only one point has both labels ③ and ④ , $x = (\frac{3}{4}, \frac{1}{4})$. The resulting pair $x = (\frac{3}{4}, \frac{1}{4})$, $y = (\frac{1}{2}, \frac{1}{2})$ is guaranteed to be a Nash Equilibrium because it is a completely labeled set of points, one that results from a forced best response

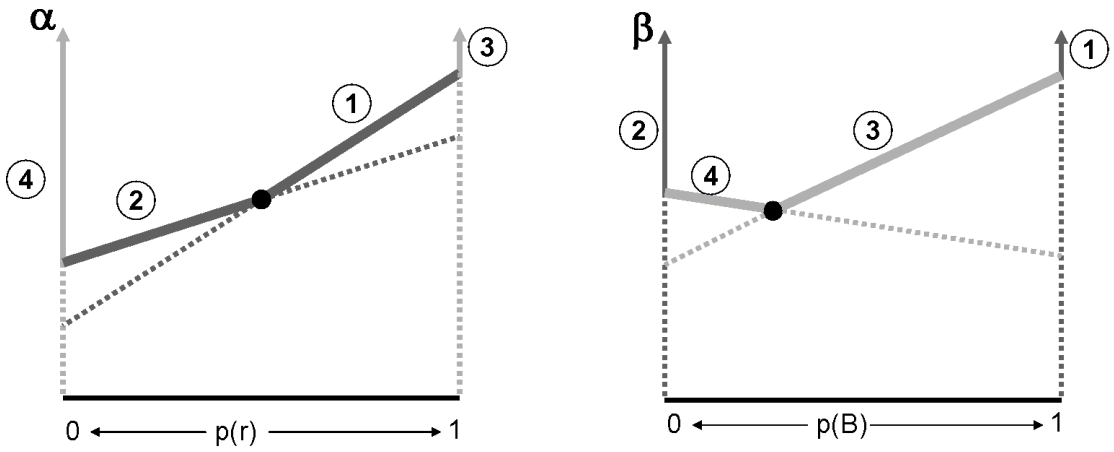


Figure 9: Node 4, Depth = 4

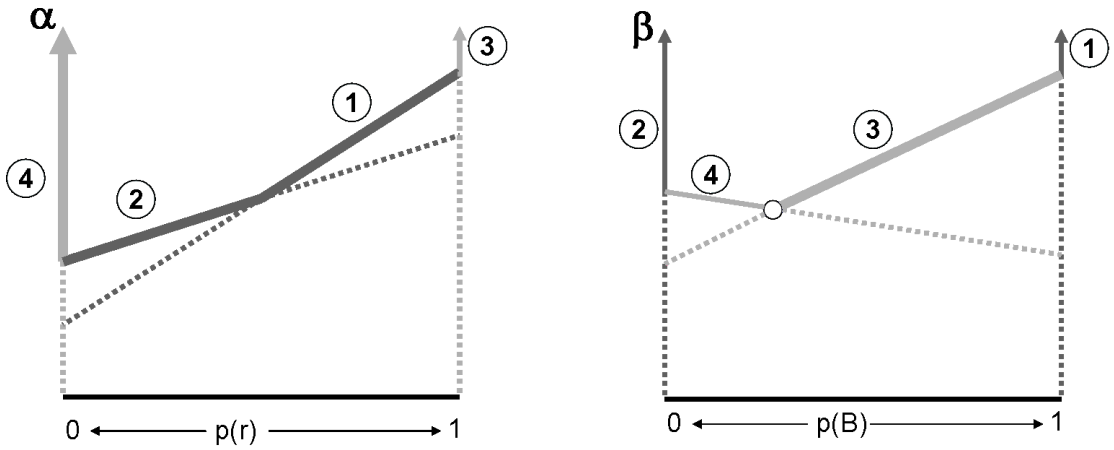


Figure 10: Node 9: An infeasible node.

situation for all $i \in M$ and $j \in N$.

Since all the branching steps followed to achieve this particular Nash Equilibrium employed best responses rather than unplayed strategies, the polychotomous branching steps will only force variables to be played with zero probability. Since any single such restriction makes either $\hat{P}$ or $\hat{Q}$ empty, this particular branch of the tree terminates at the next level.

As the algorithm employs a depth first search, the next node to check is that which is a sibling of the NE node. As the NE node was obtained by forcing $\alpha = (Ay)_2$, its sibling is obtained by confirming the label ④ in the other manner, namely setting $y_2 = 0$. Figure 10 results. Note that there is no point in the left plot that satisfies all three criteria; the intersection of the best response curves for top and bottom, along with the line segment $y_2 = 0$ is empty.

Node 9 is therefore a leaf; the sequence terminates. It is finding these infeasible points that make the algorithm terminate. Finding infeasible points high in the tree

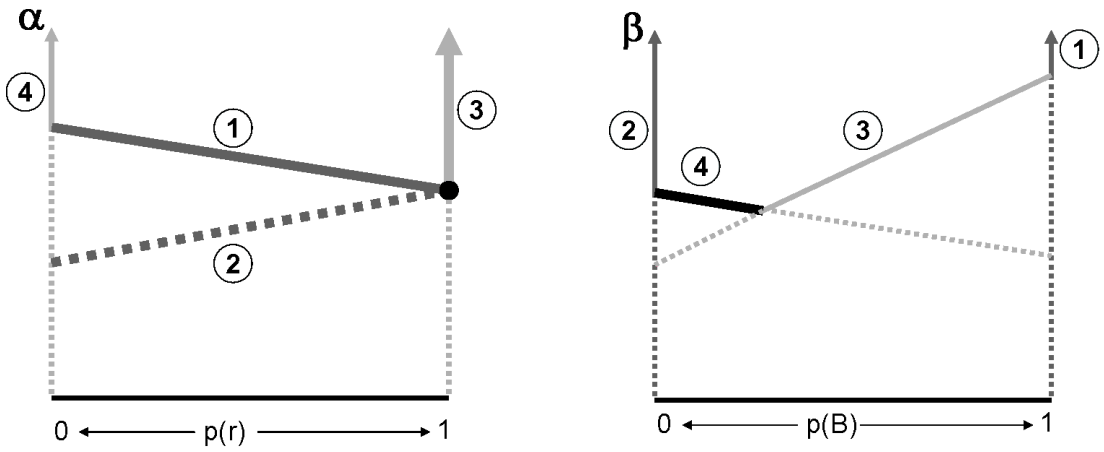


Figure 11: Best response curves for a degenerate game

decreases the number of nodes to explore and thus a good method of choosing which conditions to impose on $\hat{P}$ and $\hat{Q}$ is important for the speed of the algorithm.

Degeneracies occur when points on the best response curves for a specific player have more labels than the number of dimensions in which it is plotted (number of strategies for the other player). The extra label provides an extra degree of freedom on the other best response diagram; rather than requiring some number l of labels, only $l-1$ are required. Since there can be a continuum of points with a specific label, all points on this continuum are NE. Consider Figure 11. Since labels ① , ② , and ③ are acquired by the point in the left plot, Player I in the right plot can choose any point along the black line segment consisting of the points labeled ④ that are on the upper envelope. As the linear programming procedure finds the lowest point on the best response curve, the extreme equilibrium in Figure 12 is discovered. At another point, the degeneracy in the left diagram is not used and label ② is acquired by setting $x_2 = 0$. This constrains the set $\hat{P}$ to the single equilibrium point in 13. As a result, both extreme equilibria are found. Since y is the same in both cases, any linear combination of the two points is a Nash Equilibrium. Thus, all points on the yellow line segment in 11 are identified by the algorithm.

It is possible that, given certain degenerate games, not all extreme equilibria are found in the first $m+n$ levels of the search tree. An example where this is the case will be provided in Section 6.

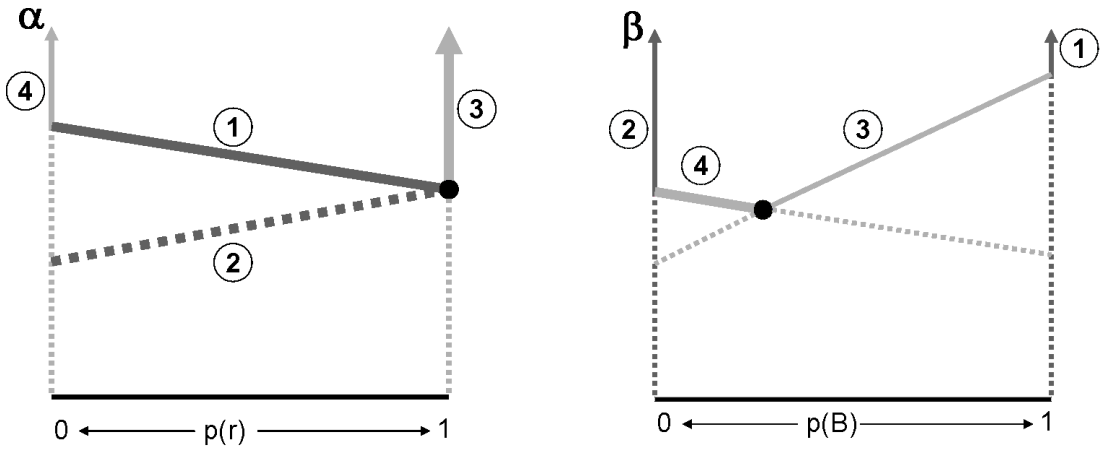


Figure 12: First endpoint of the degenerate equilibrium

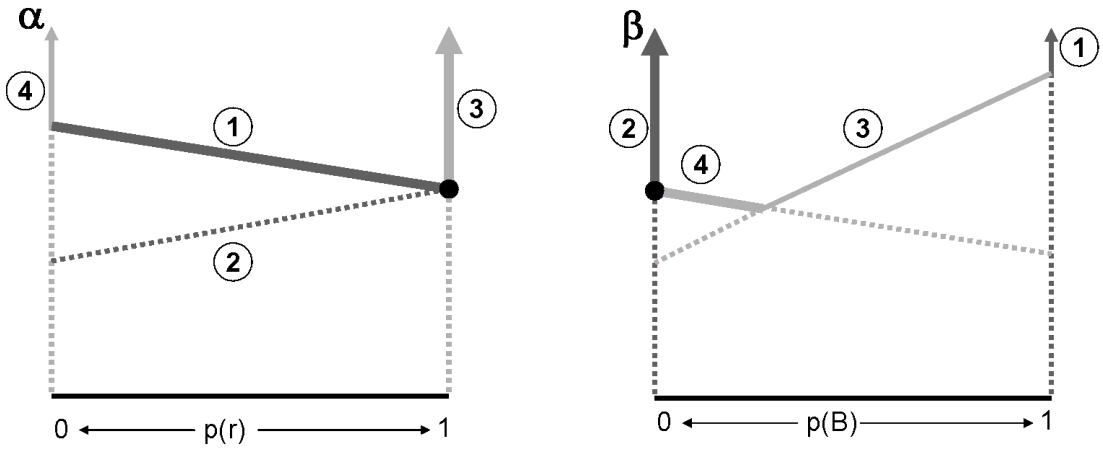


Figure 13: Second endpoint of the degenerate equilibrium

4 Integer Pivoting

The previously available implementation of the EEE algorithm uses a commercially available library (CPLEX) to solve the linear programs present in each step of the algorithm with the standard simplex method. This employs division at each step in the algorithm, with no guarantee that the results will be integers. As a result, floating-point numbers must be used in the implementation. In many cases, performing this division as part of the pivot step is sufficient.

The storage of floating-point numbers by a computer necessitates some rounding; regardless of the base used to store a number, there exist some rational numbers that cannot be expressed as a single number in a finite number of digits. Representing $\frac{2}{3}$ as a decimal point followed by 100 sixes may be close to $\frac{2}{3}$ and may suit most computations well, but it is not exact.

Since all strategies that provide a player with a less than maximal payoff are played with probability zero, it is crucial that an algorithm be able to distinguish between even minuscule differences in payoffs. In large bimatrix games, small rounding errors due to the use of floating-point numbers may well add up over the course of the large number of pivot steps needed to solve a large linear program. As a result, a more exact method of solving the linear programs for these problems is necessary.

One known solution to the above problem is avoiding division resulting in non-integral values. This practice, known as integer pivoting, is a variation on the standard simplex algorithm, where it is generally attributed to Edmonds. Azulay and Pique (2001) describe a version of the technique and provide several historical references.

Recall the linear program from 1.1

$$\begin{aligned} \max_{x_1,x_2} \; & z = 7x_1 + 3x_2 \\ \text{s.t.} \; & 3x_1 + x_2 \leq 11 \\ & x_1 + 2x_2 \leq 7 \\ & x_1, x_2 \geq 0 \end{aligned}$$

and its tableau representation

$$\begin{pmatrix} 3 & 1 & 1 & 0 & 11 \\ 1 & 2 & 0 & 1 & 7 \\ -7 & -3 & 0 & 0 & 0 \end{pmatrix}.$$

The first step of the integer pivoting procedure selects the variables to enter and exit the basis in the same way as the original pivot procedure. The variable with largest (in absolute value) negative coefficient in the objective function, here x_1, is chosen to enter

the basis. There are two bounds on x_1 to ensure non-negativity of the variables

$$\begin{aligned} \text{Constraint 1} \rightarrow \quad & 3x_1 \leq 11 \rightarrow \quad x_1 \leq \tfrac{11}{3} \\ \text{Constraint 2} \rightarrow \quad & x_1 \leq 7 \rightarrow \quad x_1 \leq 7. \end{aligned}$$

As the first bound is more restrictive, r_1 (the basis variable corresponding to the first row) exits the basis while x_1 enters.

The first step in the pivot procedure is multiplication by the pivot element, the intersection of the pivot row and the pivot column, of all non-pivot rows. This is the coefficient of the variable entering the basis in the row where it will enter. Here this corresponds to the first element of the tableau; multiplying yields

$$\begin{pmatrix} 3 & 1 & 1 & 0 & 11 \\ 3 & 6 & 0 & 3 & 21 \\ -21 & -9 & 0 & 0 & 0 \end{pmatrix}.$$

As x_1 is meant to enter the tableau, the column corresponding to x_1 must take the form of a column of the identity matrix, namely all elements except the pivot element must be zero. To accomplish this, a multiple of the pivot row is subtracted from each other row such that the element in the pivot column is zero. Subtracting the first row from the second row and 7 times the first row from the objective (third) row yields

$$\begin{pmatrix} 3 & 1 & 1 & 0 & 11 \\ 0 & 5 & -1 & 3 & 10 \\ 0 & -2 & 7 & 0 & 77 \end{pmatrix}.$$

At this point the basis columns constitute a multiple of the identity matrix. x_1 and r_2 now constitute the basis. The entire tableau is thus triple the value it would take in the normal simplex algorithm. x_1 now takes the value $\frac{11}{3}$, and the vector (x_1, x_2, r_1, r_2) is $(\frac{11}{3}, 0, 0, \frac{10}{3})$. The value of the objective function has increased from 0 to $\frac{77}{3}$.

The key to the integer pivoting algorithm is that the basis is allowed to remain a multiple of the identity matrix rather than the identity matrix itself. The multiple, which results from the multiplication pivot step, is a single value by which all of the elements of the b vector must be ultimately divided. Delaying this division, however, allows all values to remain integral.

In the next pivot step, x_2 enters the basis while r_2 exits. Multiplying by the pivot element yields

$$\begin{pmatrix} 15 & 5 & 5 & 0 & 55 \\ 0 & 5 & -1 & 3 & 10 \\ 0 & -10 & 35 & 0 & 385 \end{pmatrix}$$

and subtraction to ensure the basis is a multiple of the identity matrix results in

$$\begin{pmatrix} 15 & 0 & 6 & -3 & 45 \\ 0 & 5 & -1 & 3 & 10 \\ 0 & 0 & 45 & 0 & 405 \end{pmatrix}.$$

Note that the non-pivot rows, the first and third, are both divisible by 3, which happens to be the pivot element of the previous step. This is not accidental; it occurs at each step in the pivoting procedure. Division by the previous multiple allows the elements of the tableau to stay of reasonable size. Division by 3 of these two rows yields

$$\begin{pmatrix} 5 & 0 & 2 & -1 & 15 \\ 0 & 5 & -1 & 3 & 10 \\ 0 & 0 & 15 & 0 & 135 \end{pmatrix}.$$

There remain no positive coefficients in the objective function. The function thus takes its maximum value at $\frac{135}{5} = 27$ with the vector (x_1, x_2, r_1, r_2) at $(3, 2, 0, 0)$. While these values happen to be integral, non-integral values at this point can be represented in fractional form, i.e. if the tableau had read

$$\begin{pmatrix} 7 & 0 & 2 & -1 & 15 \\ 0 & 7 & -1 & 3 & 10 \\ 0 & 0 & 15 & 0 & 135 \end{pmatrix}$$

the vector (x_1, x_2, r_1, r_2) would read $(\frac{15}{7}, \frac{10}{7}, 0, 0)$. Storing in such a way allows computation of the solutions for the linear program exactly.

To recap, given a linear program in a tableau with a feasible basic solution, integer pivoting is accomplished by

1. Determining the element to enter the basis
2. Determining the element to leave the basis (using the minimum ratio test)
3. Multiplying all rows other than the pivot row by the pivot element
4. Subtracting a multiple of the pivot row from each other row so that the non-pivot rows equal 0 in the pivot column
5. Dividing all rows other than the pivot row by the multiple of the identity matrix from the previous step
6. Repeating steps 1-5 while at least one coefficient in the objective function is negative

The result will be the optimal solution, multiplied by an integer number which is the multiple of the identity matrix used in the last step. Dividing by this multiple yields the appropriate result.

4.1 Proof of Divisibility

The claim that each non-pivot row is divisible by the pivot element of the previous step is not immediately obvious. It is necessary to prove that in each iteration of the integer pivoting simplex algorithm, all elements in non-pivot rows are divisible by the previous multiple after the subtraction of pivot row multiples step. In addition, the value resulting from this division must be the same as the value of the current pivot element, as to make a multiple of the identity matrix. What follows is a description of how this works; readers wishing a more formal view involving determinants should see Azulay and Pique (2001).

Let the basis start as an identity matrix and the first pivot element be a_{rs}. The pivot row is r and the pivot column is s. The first step in the integer pivoting procedure is to multiply each row other than the pivot row by the pivot element. This corresponds to the rule

$$\begin{aligned} a_{ij} &\rightarrow a_{rs}a_{ij} \quad i \neq r \\ a_{ij} &\rightarrow a_{rj} \quad\quad i = r \end{aligned}$$

Then, a multiple of the pivot row is subtracted from each other row to make the element in the pivot column zero. The appropriate multiple is $\frac{a_{is}a_{rs}}{a_{rs}} = a_{is}$ since, in column s for rows $i \neq r$, this yields $a_{ij}a_{rs} - a_{rj}a_{is} = a_{is}a_{rs} - a_{is}a_{rs} = 0$. This corresponds to the rule

$$\begin{aligned} a_{ij} &\rightarrow a_{rs}a_{ij} - a_{is}a_{rj} \quad i \neq r \\ a_{ij} &\rightarrow a_{rj} \quad\quad\quad\quad\quad\quad\quad\quad i = r \end{aligned}$$

This completes the first pivot step. The value of the multiple M is now a_{rs}, which is true both for the pivot element itself and the basis elements of the tableau since they begin with value 1 and all other values are in the column are zero (by definition of the identity matrix assumption) yielding $a_{rs}a_{ij} - a_{is}a_{rj} = a_{rs} - 0(a_{is}) = a_{rs}$.

For the second step, first assume that the new pivot element comes from the same row as the last pivot element, in particular let the new row $g = r$. Again, each element is multiplied by the pivot element; as the pivot element is from the old pivot row, it must be of the form a_{rh} where h represents whichever column happens to have the new pivot element. The matrix becomes

$$\begin{aligned} a_{ij} &\rightarrow a_{rh}(a_{rs}a_{ij} - a_{is}a_{rj}) \quad i \neq r \\ a_{ij} &\rightarrow a_{rj} \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad i = r = g. \end{aligned}$$

The elements in column h but not a_{gh} must be made zero so that they can serve as the zeroes in the identity matrix. This corresponds to subtracting a multiple of the pivot row from each row. The appropriate multiple in this case is

$$\frac{a_{rh}(a_{rs}a_{ih} - a_{is}a_{rh})}{a_{rh}} = a_{rs}a_{ih} - a_{is}a_{rh}.$$

The matrix becomes

$$\begin{aligned} a_{ij} &\rightarrow a_{rh}(a_{rs}a_{ij} - a_{is}a_{rj}) - a_{rj}(a_{rs}a_{ih} - a_{is}a_{rh}) && i \neq r \\ a_{ij} &\rightarrow a_{rj} && i = r = g \end{aligned}$$

Expanding the expression yields

$$\begin{aligned} a_{ij} &\rightarrow a_{rh}a_{rs}a_{ij} - a_{rh}a_{is}a_{rj} - a_{rj}a_{rs}a_{ih} + a_{rh}a_{is}a_{rj} && i \neq r \\ &= a_{rh}a_{rs}a_{ij} - a_{rj}a_{rs}a_{ih} \\ &= a_{rs}(a_{rh}a_{ij} - a_{rj}a_{ih}) \\ a_{ij} &\rightarrow a_{rj} && i = r = g \end{aligned}$$

All elements of the non-pivot row $r = g$ are clearly divisible by a_{rs} when the pivot row is the same for both steps.

The columns corresponding to the basis, a_{rs} in the previous step, are multiplied by a_{rh} in the new step to yield $a_{rs}a_{rh}$. Again, nothing is subtracted since all elements in other rows, by definition of the basis matrix, are zero. Hence the resultant $a_{rs}a_{rh}$, when divided by a_{rs} is simply a_{rh}, and all elements of the basis matrix have the same value.

If $r \neq g$, the pivot element is multiplies each element in the rows other than g. The pivot element itself is of the form $a_{rs}a_{gh} - a_{gs}a_{rh}$ since it was in the non-pivot row previously. Therefore, the multiplication yields

$$\begin{aligned} a_{ij} &\rightarrow (a_{rs}a_{gh} - a_{gs}a_{rh})(a_{rs}a_{ij} - a_{is}a_{rj}) && i \neq r, g \\ a_{ij} &\rightarrow (a_{rs}a_{gh} - a_{gs}a_{rh})a_{rj} && i = r \\ a_{ij} &\rightarrow a_{rs}a_{gj} - a_{gs}a_{rj} && i = g \end{aligned}$$

Again, the old element in the pivot row is subtracted (times a multiple) from all other rows. The element in column j of row g is $a_{rs}a_{gj} - a_{gs}a_{rj}$. For elements in rows $i \neq r, g$ the multiple of this number subtracted from each element in row i is $b = a_{rs}a_{ih} - a_{is}a_{rh}$ while for elements in row $i = r$ it is $b = a_{rh}$. This step yields

$$\begin{aligned} a_{ij} &\rightarrow (a_{rs}a_{gh} - a_{gs}a_{rh})(a_{rs}a_{ij} - a_{is}a_{rj}) - (a_{rs}a_{ih} - a_{is}a_{rh})(a_{rs}a_{gj} - a_{gs}a_{rj}) && i \neq r, g \\ a_{ij} &\rightarrow a_{rj}(a_{rs}a_{gh} - a_{gs}a_{rh}) - a_{rh}(a_{rs}a_{gj} - a_{gs}a_{rj}) && i = r \\ a_{ij} &\rightarrow a_{rs}a_{gj} - a_{gs}a_{rj} && i = g \end{aligned}$$

Expanding these expressions and factoring yields

$$\begin{aligned}
a_{ij} &\to a_{rs}[a_{ij}(a_{rs}a_{gh} - a_{gs}a_{rh}) - a_{gs}(a_{rh}a_{ij} - a_{ih}a_{rj})] && i \neq r, g \\
a_{ij} &\to a_{rs}(a_{rj}a_{gh} - a_{rs}a_{gj}) && i = r \\
a_{ij} &\to a_{rs}a_{gj} - a_{gs}a_{rj} && i = g
\end{aligned}$$

Both rows that need to be divisible by a_{rs}, namely those rows for which $i \neq g$, are clearly divisible by that factor.

Recall that after the first pivot step, all basis elements are a_{rs}. Multiplying by the pivot element $a_{rs}a_{gh} - a_{gs}a_{rh}$ only in the new non-pivot rows yields

$$a_{ij} \to a_{rs}(a_{rs}a_{gh} - a_{gs}a_{rh}) \quad i \neq g.$$

Dividing by a_{rs} gives $a_{rs}a_{gh} - a_{gs}a_{rh}$, which is the same value as the new pivot element. Therefore, all elements of the basis submatrix have the same value.

The above argument has shown that, given any tableau, all non-pivot rows after the second pivot operation are divisible by the pivot element from the first pivot operation and result in a multiple of the identity matrix in the basis submatrix. In a similar manner, the elements of the tableau after the first pivot can be considered a new "original" tableau, and thus all non-pivot rows are divisible by the second pivot element after the third pivot step with the appropriate result for the identity submatrix. The initial tableau, with multiple 1, provides the base case as all elements are trivially divisible by 1. It has already been shown that, in this case, all elements of the basis submatrix have value a_{rs}. This illustrates how the the integral division works for the first few steps of the algorithm.

5 Algorithm Implementation

The integer pivoting EEE algorithm (EEE-I) is a Java-encoded stand-alone program that reads the payoff matrices A and B from text files and, using only integer operations, outputs all extreme Nash Equilibria of the bimatrix game. As the integers involved can easily exceed the standard integer size, the built-in Java BigInteger class is used for all operations in the linear program. This allows integers constrained in size only by available memory, and thus removes any question of overflow in large games or those with large payoffs.

The implementation consists of 4 major classes and several ancilliary classes. EEE.java, the main class, controls the flow of the algorithm at the highest level by referencing the other classes. Bimatrix.java performs the read-in procedures and returns a data object containing two BigInteger arrays. Each node of the search tree is simulated in the Node.java class. Each instance of Node holds the current linear programs P and Q, the current state of both x and y, a boolean vector describing the forced constraints, the current value of α and β, and a number of tracking variables. In addition, Node.java contains the implementation of all methods needed on the level of the node, including the choice of variable to force and the creation of new child nodes. LPType.java defines the linear program tableau LPType, vectors labeling the basis and cobasis elements of the linear program, and all necessary operations on the linear program including solution by integer pivoting and adaptation of the linear program to new constraints. Other classes, including NElist.java, NENode.java, and Tools.java provide support to these major classes. These classes are all available in Appendix A.

Rather than create an explicit tree, the EEE-I algorithm implementation uses recursion and stack properties to explore potential nodes through a simulated depth-first search. Child nodes, through recursion, are added to the top of the stack and are removed only when no further children exist; in this way a depth first search is simulated on a tree that is built as it is searched. While the explicit tree structure is helpful for understanding the EEE algorithm, it provides no added benefit in the implementation. Rather, the existence of such a tree structure would have all nodes present in memory, drastically increasing the demand on the machine. Given the depth first method only those nodes created but not solved, along with the current node being solved, are in memory.

As the implementation relies on calls between classes and data types (particular as relating to the recursive calls), a description class-by-class is complex and relatively uninformative. Therefore, what follows is a description of the implementation as it runs, which provides a more intuitive connection between the implementation and the theoretical aspects of the algorithm itself.

Upon invocation, the EEE.java class (the runnable class) calls Bimatrix.java to create a new Bimatrix object. The Bimatrix class reads in to this new object the BigInteger matrices A and B from external text files. User input dictates the size of the game. The game parameters, which are the number of strategies of each player, and the matrices themselves comprise the Bimatrix object, which is a complete description of the bimatrix game in question.

5.1 Initialization

The first node contains the initial linear programs. By convention of the original EEE paper, the linear program $Q(y|x)$ is solved first and its solution used as an initial feasible point for $P(x|y)$. Note that x is not involved in any constraints, it is simply involved in the objective function. As a result, the value of x is irrelevant to all but one line of the tableau, and will be ignored here until it is necessary.

Initialization requires that a given vector be found that satisfies all the initial constraints. In addition, it is necessary to use this vector to calculate the tableau. It is easy to choose a vector y that satisfies the first constraint $\sum_{j=0}^{n} y_j = 1$; in particular, the vector with a 1 in the first position and 0 elsewhere. The remaining constraints are of the form $\alpha \geq (Ay)_i, \quad \forall i \in M$. In this section, the constraint $\alpha \geq (Ay)_i$ will be written in the equivalent form $\alpha \geq \sum_{j=1}^{n} A_{ij} y_j, \quad \forall i \in M$ for reasons that will become clear. There are m of these constraints. Introducing a slack variable for each of these constraints, labeled s_t for constraint t, yields $s_t = \alpha - \sum_{j=1}^{n} A_{tj} y_j$, where s must be nonnegative. The smallest α that can be chosen to satisfy the non-negativity constraint is the largest subtracted term $\max_q A_{q1}$ since $y_1 = 1$. Substituting in this value of α leaves

$$s_t = \max_q A_{q1} y_1 - A_{t1} y_1$$

for $t \neq \arg\max_q A_{q1} y$ and

$$s_t = \max_q A_{q1} y_1 - \max_t A_{t1} y_1 = 0$$

for $t = \arg\max_q A_{q1} y$. For future notational purposes, let $\hat{q} = \arg\max_q A_{q1} y_1$.

Given the chosen vector y, the constraint equation set simplifies to

$$\begin{array}{rcl} y_1 + \sum_{k=2}^{n} y_k & = & 1 \\ s_1 + \sum_{j=1}^{n} A_{1j} y_j - \alpha & = & 0 \\ s_2 + \sum_{j=1}^{n} A_{2j} y_j - \alpha & = & 0 \\ & \vdots & \\ s_{\hat{q}} + \sum_{j=1}^{n} A_{\hat{q}j} y_j - \alpha & = & 0 \\ & \vdots & \\ s_m + \sum_{j=1}^{n} A_{mj} y_j - \alpha & = & 0 \end{array}$$

There already exists an expression for y_1 in cobasic variables, $y_1 = 1 - \sum_{k=2}^{n} y_k$. Substituting yields

$$\begin{array}{rcl} y_1 + \sum_{k=2}^{n} y_k & = & 1 \\ s_1 + A_{11} + \sum_{j=2}^{n} (A_{1j} - A_{11}) y_j - \alpha & = & 0 \\ s_2 + A_{21} + \sum_{j=2}^{n} (A_{2j} - A_{21}) y_j - \alpha & = & 0 \\ & \vdots & \\ s_{\hat{q}} + A_{\hat{q}1} + \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1}) y_j - \alpha & = & 0 \\ & \vdots & \\ s_m + A_{m1} + \sum_{j=2}^{n} (A_{mj} - A_{m1}) y_j - \alpha & = & 0 \end{array}$$

in which y_1 has been removed from all except the first equation.

By construction, $s_{\hat{q}}$ is 0 (since α at first set to that value) and therefore α can be expressed entirely in cobasic variables as

$$\alpha = s_{\hat{q}} + A_{\hat{q}1} + \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1}) y_j.$$

Substituting into the constraint set and simplifying gives

$$\begin{array}{rcl} y_1 + \sum_{k=2}^{n} y_k & = & 1 \\ s_1 - s_{\hat{q}} - \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1} - A_{1j} + A_{11}) y_j & = & A_{\hat{q}1} - A_{11} \\ s_2 - s_{\hat{q}} - \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1} - A_{2j} + A_{21}) y_j & = & A_{\hat{q}1} - A_{21} \\ & \vdots & \\ \alpha - s_{\hat{q}} - \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1}) y_j & = & A_{\hat{q}1} \\ & \vdots & \\ s_m - s_{\hat{q}} - \sum_{j=2}^{n} (A_{\hat{q}j} - A_{\hat{q}1} - A_{mj} + A_{m1}) y_j & = & A_{\hat{q}1} - A_{m1} \end{array}$$

This set of equations describes a feasible solution to the first linear program. In particular, there are $m + 1$ basic variables (y_1, α, $s_i \ \forall i \neq \hat{q} \in M$) and n basic variables ($y_i \ \forall i \neq 1$, $s_{\hat{q}}$).

These equations can be put into the tableau form described earlier. The same computations, switching the appropriate variables, sets up the initial feasible solution for

$P(x|y)$. The only difference is that a $y = y^*$ has already been found by the LP, which is used in the objective function.

Since the objective function line of the tableau depends on the other player's strategy, it must be updated before each integer pivoting procedure is started. Recall that the objective function $P(x|y)$ is $x^\top Ay - \beta$, consisting of a linear combination of each of the x variables along with the dual variable β. Each of the m variables x_i for $i \in M$ can be in the basis, the cobasis, or previously removed from the linear program. In this last case the variable has value 0 and, as a result, nothing need be added to the objective row of the LP. If x_i is in the cobasis, adding $x_i(Ay)_i$ to the objective of the LP requires only adding $(Ay)_i$ to the objective row in the column representing x_i. If x_i is in the basis, the row corresponding x_i is a representation of x_i in cobasic variables; to add $x_i(Ay)_i$ to the objective row requires adding $x_j(Ay)_i$ or $s_j(Ay)_i$ to the objective row for each j in the cobasis. The result, after subtracting the β row, is an objective row that contains the up-to-date representation of the objective function in cobasic variables.

5.2 Introducing Constraints, Determining Feasibility, and Integer Pivoting

At each node other than the root, a decision or slack variable is forced to have value 0. Since these variables are all constrained to be non-negative, any change in the value of the variable is a decrease from positive to zero. In the EEE-I implementation, it is necessary to incorporate this additional constraint into the tableau and, in addition, determine whether the tableau remains feasible. Fortunately, it is possible to accomplish both steps simultaneously.

Instead of thinking of removing a variable x_1 from the tableau, consider the linear program defined by minimizing the value of x_1. As the goal is to determine whether there exists a feasible solution when $x_1 = 0$, the set of constraints is exactly the same as that defined in the tableau. If the minimized value of x_1 subject to those constraints is non-zero, there exists no feasible solution to the tableau with $x_1 = 0$ and, thus, forcing $x_1 = 0$ renders the linear program infeasible. This defines the end of a path from the root and the node in question is a leaf of the algorithmic tree.

However, if x_1's minimized value is 0, it is possible to pivot x_1 into the cobasis. Since x_1 will no longer deviate from 0, its column in the cobasis can be dropped. At each step, one column is removed from the cobasis, decreasing the dimension of the linear program. The one exception is when all cobasic values are 0, in which case the basic variable can be dropped immediately (assuming its b value is 0) as there is no interaction between it and any other variable and it has already been minimized to have value 0. Applying this method allows the new tableau to be directly computed

from the old tableau, incorporating the additional constraint without requiring a new initialization procedure at each step.

In the bimatrix game discussed in Section 1.2 (and using the same objective functions), the tableau for $Q(y|x)$ upon entering the second node (Node 1) is

$$Q(y|x) = \begin{pmatrix} -2 & -1 & 2 \\ 1 & 0 & 1 \\ 2 & -1 & 1 \\ 2 & 1 & / \end{pmatrix}$$

where the basis, in order of rows, is (α, y_1, s_1) and the cobasis, in order of columns, is (y_2, s_2). (Note that here and throughout this paper, a / is placed in the z position to replace the number there. This is because the algorithm is not set up to compute the value of z - the value is irrelevant to the algorithm - and thus having the number that is there in the implementation may lead to confusion.) This node explores the possibility that Player II's first pure strategy is a best response, which corresponds to $s_1 = 0$. As s_1's representation is the third row of the matrix, that row (temporarily) becomes the objective function of the new linear program to minimize s_1.

Minimization proceeds along the same lines as maximization, with the sole difference that positive coefficients, rather than negative, are chosen in the objective line to determine the incoming variable. By the normal selection process, s_1 leaves the basis while y_2 enters the basis. The resulting tableau reads

$$Q(y|x) = \begin{pmatrix} 2 & -4 & 6 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ -2 & 4 & / \end{pmatrix}$$

with basis (α, y_1, y_2) and cobasis (s_1, s_2) and $M = 2$. s_1's value, as it is in the cobasis, has been reduced to 0. This provides a feasible solution with the additional constraint: $(\alpha, y_1, y_2, s_1, s_2) = (3, \frac{1}{2}, \frac{1}{2}, 0, 0)$. The column corresponding to s_1 can be dropped as it will not change from 0, leaving

$$Q(y|x) = \begin{pmatrix} 4 & 6 \\ -1 & 1 \\ 1 & 1 \\ -4 & / \end{pmatrix}.$$

s_2 has now been permanently removed from the tableau, relegating y_2 to be a best response in all subsequent nodes as is prescribed by the algorithm.

A later node of the same bimatrix game algorithm tree provides an example of what happens in the case of infeasibility. In this step, $P(x|y)$ is constrained. On entering the node,

$$P(x|y) = \begin{pmatrix} -1 & 5 \\ -1 & 3 \\ 0 & 1 \\ 1 & / \end{pmatrix}$$

with basis (β, r_2, x_2) and cobasis (r_1). x_1's absence from the basis and cobasis shows that it has been removed in an earlier step, namely $x_1 = 0$ has been forced. This node attempts to set $x_2 = 0$. It should be clear that such an assignment is infeasible; by the probability constraint x_1 and x_2 cannot both be 0. Examining the tableau, the third row (corresponding to x_2) cannot be decreased from its value of 1 as there is no positive coefficient in its cobasic representation. As a result, there is no feasible solution with $x_2 = 0$ given the previous constraints, and this path from the root can be abandoned.

The above description of the EEE-I methodology for incorporating constraints and determining feasibility is a prime example of how solving the linear programming step internally rather than with calls to an external program clarifies the algorithm. While external programs likely use more complex procedures to enforce tight constraints, the concept of decreasing the dimension of one of the linear programs in each step has an intuitive feel as there is one less of the $m + n$ best response criteria that need to be solved after any particular node. In addition, the geometric view of the algorithm provided above ties in nicely; each constraint means a decrease of one dimension in the geometric space in question, being on a polytope corresponding to $x_i = 0$ or $r_i = 0$. This is encoded, as is the depth of the node, in the size of the linear program tableau. The geometric representation of infeasibility, the inability to find a point in the decreased dimension corresponding to all constraints, is mimicked in the inability to decrease the size of the tableau. Therefore, this particular step of the implementation nicely connects a number of aspects in this work.

Feasible linear programs, once the constraint variable is removed, are solved through the integer pivoting steps described earlier. Given the implementation up to this point, the actual pivot step is easy to implement and requires only basic operations using BigIntegers. The algorithm output includes the bimatrix game, all extreme Nash Equilibria in rational-number form, and a list of the connected equilibria. In addition, the total number of nodes visited and the total number of pivots is listed, as this is a machine and platform format that may be more reasonable for cross-algorithm testing that time. Figure 14 provides the output for a 5×5 game, modeled after Savani's implementation of the LRS enumeration algorithm available at `http://banach.lse.ac.uk`.

```
The matrix is of size 5 by 5.
Matrix A:
8  3 8 2 2
5  5 7 3 2
9 10 8 5 7
8  6 9 4 9
2  9 9 1 2
Matrix B:
1 10  2  5 7
6  6  9 10 8
5  4  4  8 1
5  9 10  7 2
6  7  5 10 4
Solving.....
5 extreme equilibria.
EE #1 x = {1} ( 0 0 1/13 8/13 4/13 )    y = {1} ( 0 1/7 5/7 1/7 0 )
EE #2 x = {2} ( 0 0  3/7  4/7    0 )    y = {2} ( 0   0 1/2 1/2 0 )
EE #3 x = {3} ( 0 0    0  2/3  1/3 )    y = {3} ( 0   0   1   0 0 )
EE #4 x = {4} ( 0 0    0    1    0 )    y = {3} ( 0   0   1   0 0 )
EE #5 x = {5} ( 0 0    1    0    0 )    y = {4} ( 0   0   0   1 0 )

A total of 94 nodes.
A total of 81 pivots.
```

Figure 14: A 5×5 game and its output.

6 Degeneracy

The binary tree obtained by the dichotomous branching on the first $m + n$ best response conditions ensure that all pure strategy equilibria are found. The procedure enumerates all 2^{m+n} possible divisions of pure strategies into those that are best responses and those that are played with zero probability. However, in degenerate games there exist equilibria in which pure strategies which are best responses are played with zero probability. This can occur as long as there exist other pure strategies that have the same payoff and at least one is played with positive probability. The question of how to deal with such potential degeneracies in the EEE algorithm is the topic of this section.

The original EEE algorithm addresses the degeneracy problem by branching off into $m + n$ nodes at each discovered Nash Equilibrium, such that there is one node for each pure strategy of the game. At the node corresponding to pure strategy i, the half of the best response condition that is not fixed is then forced to be an equality, thereby making the pure strategy be both a best response and played with zero probability. If $x_i = 0$, then the node adds the constraint that $\alpha = (Ay)_i$ and vice versa. The authors refer to this as polychotomous branching. If the node is feasible, it is a Nash Equilibrium and the polychotomous branching procedure continues for all non-doubly fixed pure strategies.

Such a method will find all degeneracies as it explores all $2^{2(m+n)}$ combinatorial possibilities of fixing variables and slack variables, but involves large computational overhead. Consider a 20×20 non-degenerate bimatrix game with 51 Nash Equilibria. At each of these 51 equilibria, each of the 40 pure strategies requires its own node to be checked for feasibility. As these are all non-degenerate equilibria, all are infeasible. The result is $40 \times 51 = 2040$ unnecessary nodes. As Nash Equilibria are found on the $m + n$ level, the degeneracy check increases the tree size dramatically, particularly for games in which the search tree is relatively full or in which there are many equilibria. As will be described in the next section, tree size is very important in determining the running time of the algorithm.

In a non-degenerate game, applying m constraints onto the strategy space of Player I and n constraints onto the strategy space of Player II results in either infeasiblity or a unique pair of feasible points. The EEE algorithm attempts to minimize the search time by finding sets of constraints that yield no feasible points well before the $m + n$ level of the search tree. In a degenerate game, however, the introduction of $m + n$ labels leaves a continuum of feasible points in at least one player's strategy set. Finding when such a continuum exists is thus a restatement of the degeneracy check problem.

In understanding degeneracy, the geometric interpretation of the EEE algorithm is

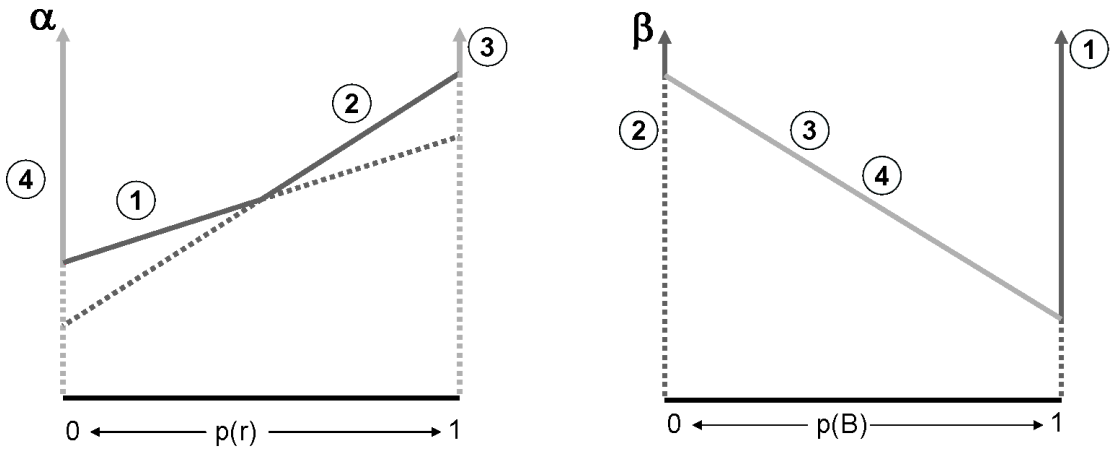


Figure 15: Best response diagrams for Player I (left) and Player II (right)

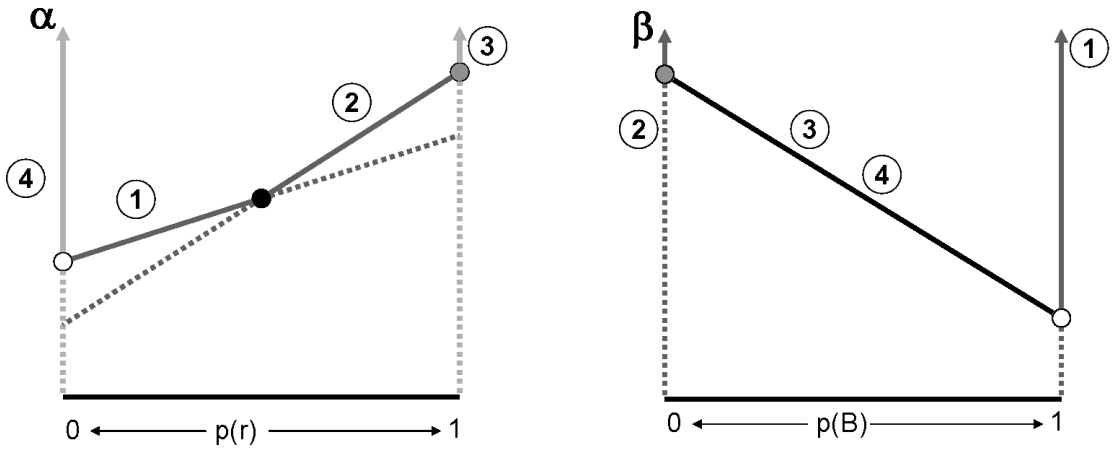


Figure 16: The Nash Equilibria of the degenerate game

helpful. Consider the 2×2 game

$$A = \begin{pmatrix} 2 & 4 \\ 1 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 5 & 5 \\ 1 & 1 \end{pmatrix}$$

whose best response curves are plotted in Figure 15. The bimatrix game has 4 extreme Nash Equilibria (Figure 16), of which two are the pure strategy pairs represented by the white and gray points. If Player I plays with positive probability both of his pure strategies (as represented by the black point), any point on the second player's best response diagram which contributes both labels ③ and ④ is a Nash Equilibrium. Any strategy of Player II provides both labels, and thus $x = (\frac{1}{2}, \frac{1}{2})$, $y = (t, 1-t)$ is a Nash Equilibrium for all $t \in [0,1]$. The extreme equilibria are the endpoints of the best response line segment, $x = (\frac{1}{2}, \frac{1}{2}), y = (0,1)$ and $x = (\frac{1}{2}, \frac{1}{2}), y = (1,0)$.

There exists some node in the tree where Player II's first strategy ③ is a best response

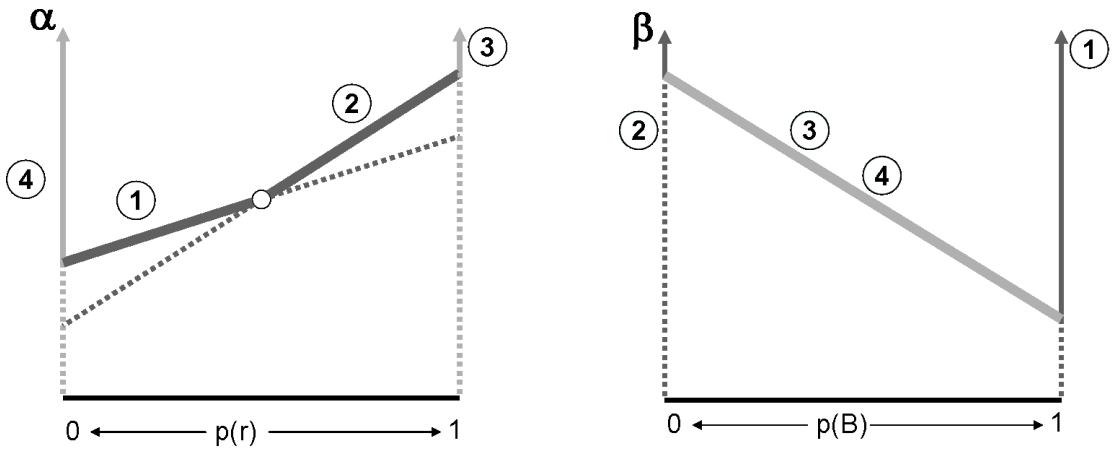


Figure 17: A node where Player I's strategies are both best responses, and one of Player II's strategies is a best response. The diagram does not change when the second of Player II's strategies is made a best response.

($\beta = (x^\top B)_1$) and her second strategy is to be added as a best response ($\beta = (x^\top B)_2$), or vice versa. In Figure 17, both of Player I's strategies have been forced to be best responses. Forcing Player II's second strategy to be a best response does not decrease the size of the feasible set - it is a repetitive constraint corresponding to a redundant equation.

Finding both endpoints of a degenerate game within the first $m + n$ levels of the search tree depends critically on the objective function used. Only if the objective function happens to find both endpoints at various nodes in the search tree will a degeneracy check remain unlikely. In this game, the objective function $P(x|y) = x^\top Ay - \beta$ and $Q(x|y) = x^\top By - \alpha$ finds only one of the two extreme points within the first $m + n$ levels of the tree. However, the objective function $P(x|y) = x^\top(A + B)y - \beta$ and $Q(y|x) = x^\top(A + B)y - \alpha$ finds both. Switching the rows of Player II's payoff matrix (which changes the game) yields the opposite result. It seems very unlikely that there exists a pair of objective functions that are guaranteed to find all extreme equilibria without a degeneracy check, and therefore some sort of degeneracy check seems necessary.

However, the insight gained from the geometric approach can be used to improve the efficacy of the degeneracy check. Recall that the EEE-I implementation discussed in this paper includes constraints in a linear program by solving the linear program of minimizing the variable (or slack) to be removed. If the minimized value of the variable is 0, it can be pivoted into the cobasis where the column corresponding to the variable can be removed. If the minimized value of the variable is non-zero, the potential linear program is infeasible.

The one exception to this rule is when the row corresponding to the variable to be

removed is all zero and there is at least one variable in the cobasis. In this case, the basic variable cannot be pivoted since all potential pivot elements are zero. However, deleting this constraint from the tableau has no effect on the feasible set since there is no interaction between it and any other variable. The basic variable's line in the tableau thus describes a redundant equation, exactly the condition described above for degenerate cases. Therefore, deletion of an all-zero row necessitates a degeneracy check; all Nash Equilibria with no row deletion can be considered terminal nodes.

Analyzing the tableau structure aids in the understanding of this fact. In non-degenerate cases, each of the $m + n$ imposed constraints corresponds to the deletion of one column in one of the tableaus. At the equilibrium level, only the constant b column remains. However, in a degenerate equilibrium, a row has been deleted at some point in the first $m + n$ levels and therefore there remains at least one cobasic variable in at least one of the tableaus. The fact that this cobasic variable remains implies that the value of the basic variables is unfixed, as it depends on the value of the cobasic variable. Therefore, it remains necessary to perform a degeneracy check only as long as there is at least one cobasic variable.

In the fourth node, both pure strategies of Player I have been forced to be best responses and the first strategy of Player II has been forced to be a best response. Therefore, this node is exactly as diagrammed in Figure 17. Upon initialization of node 4 using the objective function from the original EEE paper, the tableaus read

$$Q(y|x) = \begin{pmatrix} 6 \\ 1 \\ 1 \\ / \end{pmatrix} \qquad P(x|y) = \begin{pmatrix} -4 & 1 \\ 1 & 1 \\ 0 & 0 \\ 4 & / \end{pmatrix}$$

As expected, the two constraints on Player I's strategies has reduced one of the tableaus to only a b column. The third row in the $P(x|y)$ tableau, the all zero row, is the basis row corresponding to s_2. Were this row not all zero, introducing Player II's second strategy as a best response would correspond to pivoting the variable out of the basis. However, the fact that the row is completely zero makes this impossible, and dropping the row equates to introducing a redundant equation. Hence, in the next node, there remain two columns in the tableau (one cobasic variable).

The preceding discussion implies a change to the structure of the algorithm. The EEE algorithm searches by binary branching through the first $m + n$ levels and polychotomous branching whenever there is a feasible node at the Nash Equilibrium level. The EEE-I refinement is to branch dichotomously and then polychotmously until only the b column remains. This ensures that all extreme equilibria are found, without the drastic search increase that the original EEE algorithm requires. The result is a cleaner

and more readily understandable degeneracy check structure.

It is important to note that the refinement to the degeneracy check finds all extreme equilibria regardless of the objective function employed. Finding all extreme points of degenerate equilibria is not a matter of luck; the few branches required below depth $m + n$ explore all possibilities of variable choices until none remain and all variables are fixed. What is being removed from the original EEE degeneracy description is purely nodes that either are immediately infeasible, or equilibria that will be discovered elsewhere.

In addition, rather than branching on all strategies on each level after $m + n$, index numbering of the variables is used to ensure combinations of forced variables are not rechecked. When the degeneracy check is necessary, all variables from the unfixed linear program (or both if both are unfixed) are checked by polychotomously branching on each index; however each feasible branch then only deals with those variables with indices larger than its own. In this way, all combinations of forced variables are explored, but not multiple times.

Figure 18 shows the search tree using the original EEE method handling degeneracies in a 2×2 game for an optimized objective function that will be discussed in the next section. Some sort of degeneracy check is necessary to find all 4 extreme equilibria. Sixty-seven nodes are searched by the algorithm. Figure 19 shows the search tree for the improved EEE-I handling of degeneracies. Only 29 nodes are necessary to find all equilibria. While this example is particularly chosen to show the increase, it is clear that such a dramatic change in a game as small as 2×2 projects to even greater differences in search tree sizes for games of large size.

While the new degeneracy check clarifies the algorithm, the computational improvement is difficult to measure. The number of nodes to search decreases dramatically, however it is unclear exactly how much time this saves. The EEE-I algorithm, by virtue of the column decrease at every level in non-degenerate equilibria, requires only checking the value of the b column corresponding to the pure strategy variable is non-zero. As long as it is so, there exists no feasible solution when both parts of the best response condition are introduced. Therefore, in pure strategy equilibria it seems likely that the computational time is minimal for the degenerate case, though certainly non-zero.

However, the inner workings of a commercial linear program solver likely do not lend themselves as readily to easy identification of infeasible nodes past the $m + n$ level. It is quite possible that these nodes take as long in feasibility determination as the other nodes in the search tree. In that case, the improvement garnered from the new degeneracy check would certainly be significant. In a hypothetical 2×2 non-degenerate game with 51 equilibria, none of the 2040 unnecessary nodes (calculated at

the beginning of this section) would be searched in the new check.

$$A = \begin{pmatrix} 2 & 5 \\ 2 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & 4 \\ 5 & 4 \end{pmatrix}$$

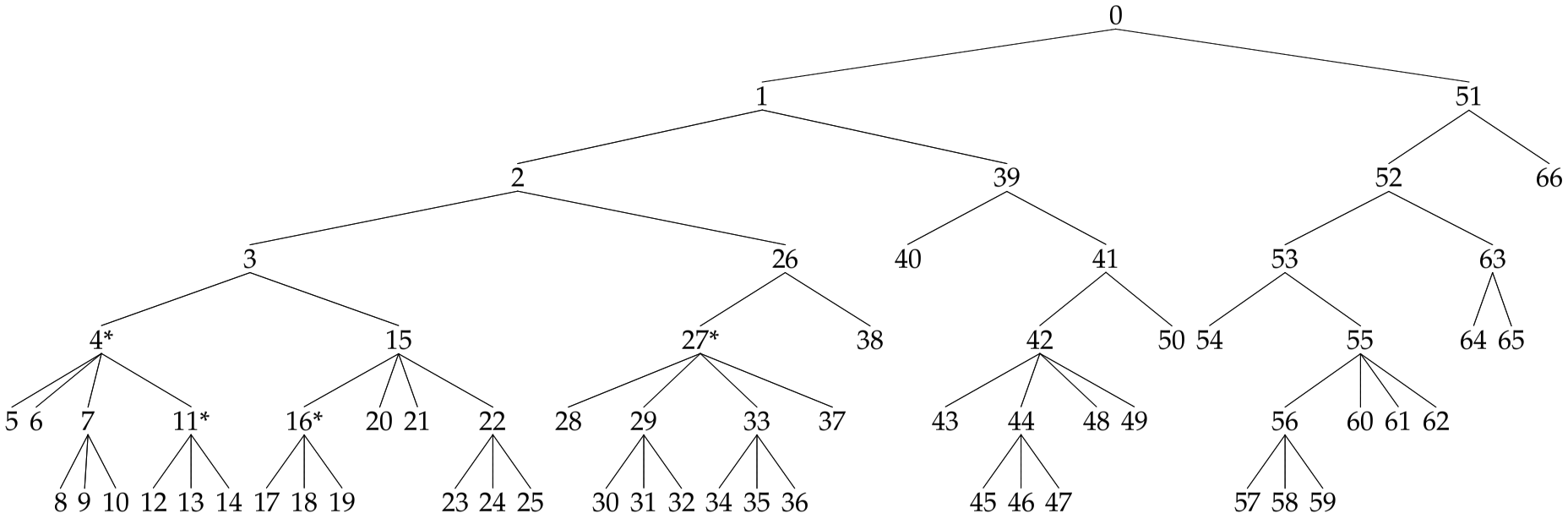


Figure 18: Search tree for original EEE degeneracy check. * represent the first time an NE is found.

$$A = \begin{pmatrix} 2 & 5 \\ 2 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & 4 \\ 5 & 4 \end{pmatrix}$$

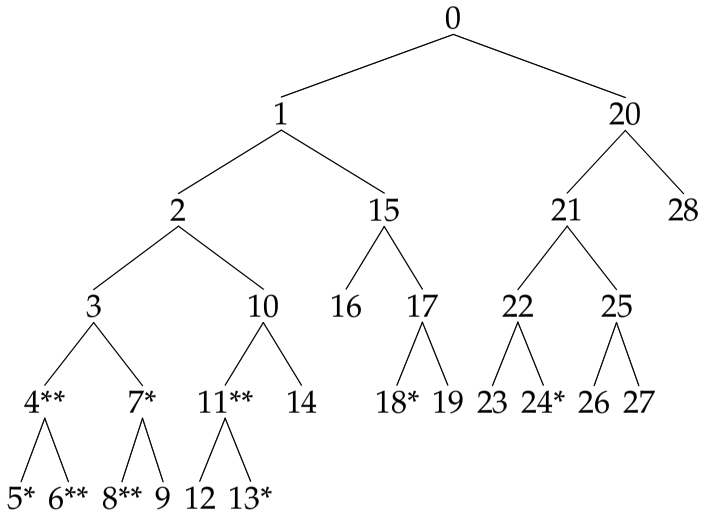


Figure 19: Search tree for improved EEE-I degeneracy check. * represent NE, ** represent the first time an NE is found.

		10×10	12×12	14×14	16×16	18×18	20×20
EEE-I	time (s)	1.13	6.27	24.03	97.13	596.60	1383.93
LRS	time (s)	0.85	4.02	18.23	155.85	1472.41	5732.65

		20×10	22×10	24×10	26×10	28×10	30×10
EEE-I	time (s)	20.37	18.72	27.87	64.89	62.18	91.22
LRS	time (s)	5.54	6.08	6.60	9.52	12.21	15.45

Figure 20: Average running time for 5 random games of each size by EEE-I and LRS. While running time for both increases with game size, LRS increases at a much faster rate.

7 Experimental Results

This section provides some brief experimental results describing EEE-I's behavior in terms of running time. The primary determinant of EEE-I's speed for a particular instance is the number of nodes searched and the number of pivots required at each node. EEE-I generally solves for equilibria faster than enumerating all 2^{m+n} possible supports because sets of constraints that render the linear program infeasible are found and their descendants ignored. For a given game, the earlier such infeasible constraints are found, the faster the algorithm will run. Finding infeasible constraint sets is the domain of the objective function and will be discussed shortly.

Games may require different search tree sizes to find all equilibria regardless of the objective function. As the number of player strategies increases, so too does the depth of the tree $m + n$ required to find a Nash Equilibrium; increased payoff matrix dimensions thus slows the algorithm. Figure 20 compares the running time of the EEE-I implementation and the LRS algorithm, discussed earlier, for random games of both square and rectangular dimension. While the running time for both algorithms increases with size, the LRS algorithm seems to be far more sensitive to the smaller of the two dimensions, making it suboptimal for games in which both dimensions are large. The LRS algorithm thus seems useful for games with large discrepancies in number of strategies while the EEE-I implementation is more promising for solution of general large games.

Empirically, the number of NE often increases as the size of the game increases. To remove the possibility that this increase in running time is wholly due to the number of equilibria, Figure 21 provides the EEE-I and LRS running times for the unique, pure-strategy equilibrium multiplication game defined by $A_{ij} = B_{ij} = (i+1)(j+1)$. By deletion of strictly dominated strategies, the only NE of this game has both agents play their last strategy. As game size increases so does running time.

Games of the same size are not all created equal; some games require a larger num-

		20×20	30×30	40×40	50×50	60×60
EEE-I	time (s)	0.32	0.58	1.13	2.00	3.23
LRS	time (s)	1.17	2.32	4.06	6.21	8.81

		20×20	30×20	40×20	50×20	60×20
EEE-I	time (s)	0.32	0.42	0.56	0.74	0.98
LRS	time (s)	1.16	1.73	2.07	2.57	3.01

Figure 21: Running time for the unique, pure-strategy multiplication game.

ber of searched nodes. Consider Figure 22, two search trees for different 2×2 games using the optimized objective function described below. On top, a highly degenerate game, large constraint sets are required before feasibility is obtained. This is separate from the issue of degeneracy discussed previously. While the improved degeneracy check decreases the size of the search tree, the game simply requires more searched nodes than the bottom tree, a non-degenerate game with 1 pure strategy NE. Search tree size is also a function of the number of NE of the instance game; since each Nash Equilibria requires at least 1 node at depth $m + n$, a game with more equilibria may well require a larger search tree.

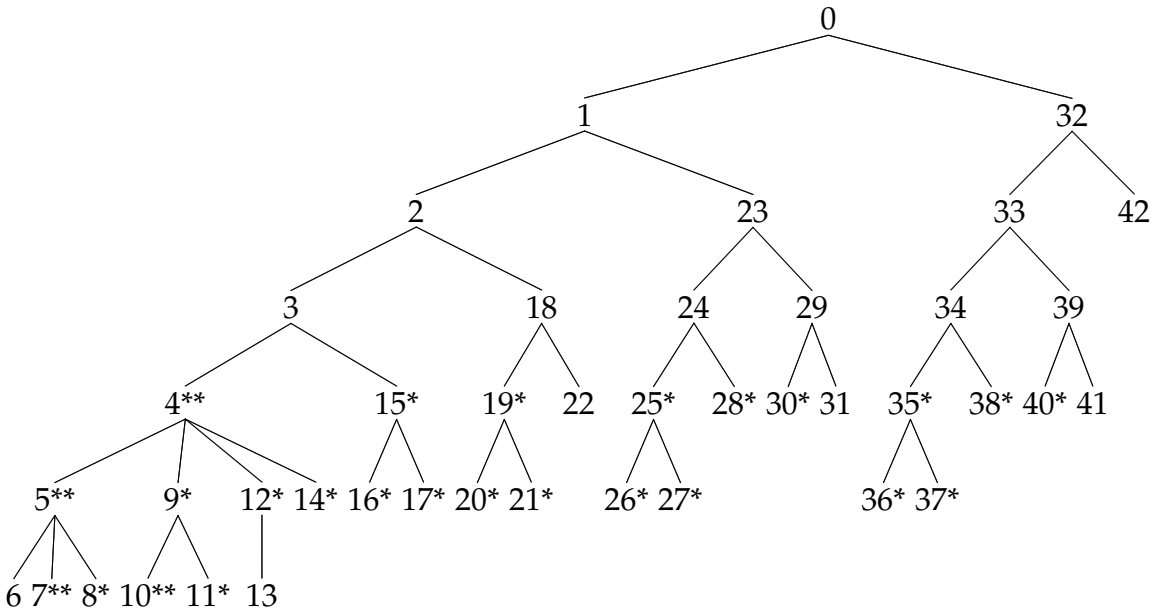
Hence, good instances for EEE-I for a given game size are non-degenerate games with a small number of equilibria. Bad instances either have a large number of equilibria or are highly degenerate. Figure 23 provides the average running time and the average number of equilibria for two 9×9 games as run by an optimized (by objective function) version of EEE-I and by the same version without the improved degeneracy check (EEE-o).

In the symmetric $r \times r$ "guessing game," each player chooses a number from 1 to r. The player choosing the lower number receives that value, while the player choosing the higher number receives the difference between the high and low choices. If both choose the same value, they receive the lowest payoff. The payoff matrices are

$$A_{ij} = \begin{cases} i+6 & i<j \\ i-j+6 & i>j \\ 1 & i=j \end{cases} \qquad B_{ij} = \begin{cases} j-i+6 & i<j \\ j+6 & i>j \\ 1 & j=i. \end{cases}$$

The "dollar game" is a combination of the Grab the Dollar and War of Attrition games as described in the GAMUT's User Guide (2004). Players surround two objects, one with utility value 100 and one with utility value 50. If both players grab for the objects at the same time, both receive nothing. If one grabs an object earlier than the other, the faster moving agent receives the object valued 100 while the slower moving player receives the object valued 50. In addition, both players value time and their

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$



$$A = \begin{pmatrix} 1 & 4 \\ 2 & 5 \end{pmatrix} \qquad B = \begin{pmatrix} 2 & 3 \\ 5 & 2 \end{pmatrix}$$

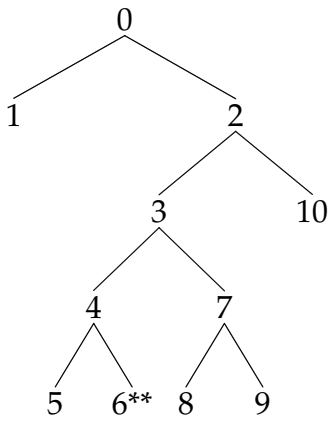


Figure 22: Search tree for a highly degenerate game (top) and a non-degenerate game (bottom). *s represent NE, ** represent the first time an NE is found.

		Guessing Game 5 NE	**Dollar Game** 73 NE
EEE-I	time	0.36	25.25
	nodes	337	141531
	pivots	530	43987
EEE-o	time	0.37	14806.06
	node	427	279020949
	pivots	530	20326989

Figure 23: Comparison of running time, nodes, and pivots for the 4 tested objective functions.

potential payoff decreases two units for each time period before a player grabs for the better object. For solution purposes, one unit is added to all payoffs. The payoff matrix is thus

$$A_{ij} = \begin{cases} 101 - 2i & i < j \\ 51 - 2j & j < i \\ 1 & i = j \end{cases} \qquad B_{ij} = \begin{cases} 51 - 2i & i < j \\ 101 - 2j & j < i \\ 1 & j = i. \end{cases}$$

The "guessing game," which has a small number of equilibria, provides an example of a good instance for EEE and one for which the improved degeneracy check makes no improvement. The "dollar game," however, provides a poor instance as it results in a large number of equilibria, many of which are degenerate. The difference is more pronounced for the old degeneracy version EEE-o than the improved EEE-I; in fact the "dollar game" provides an example of a game that benefits greatly from the improved degeneracy check. While the EEE-o algorithm takes over 4 hours to find the 73 equilibria, the EEE-I algorithm takes less than half a minute.

7.1 Objective Functions

Each node of the EEE search tree solves a linear program, whose constraint set is at the heart of the EEE algorithm but whose objective function has, to this point, been mostly ignored. The EEE algorithm proceeds by altering the constraint set and searching for feasibility, with Nash Equilibria found solely through the feasible set. While, regardless of the objective function, all NE are found by the algorithm, the form of the objective function is a major determinant of the running time of the algorithm.

The EEE algorithm's ability to find infeasibility at early tree depths depends on the order of variables chosen to force into equalities. If the combination of two particular variables renders the linear program infeasible, exploring the combination early in the search tree saves exploration of a far larger number of nodes than exploring the same combination late in the search tree. At each step, the pure strategy with largest com-

plementary slackness $x_i(\alpha - (Ay)_i) \quad \forall i \in M$ or $y_j(\beta - (x^\top B)_j) \quad \forall j \in N$ is chosen as the next pure strategy to be forced as a best response or played with zero probability. The relative values of the complementary slackness is a function of the values of A and y (for $i \in M$) or B and x (for $j \in N$) which are determined by the solution to the linear program in the last step, values determined by the objective function. As a result, choosing the proper objective function can have a large impact on EEE running time.

The original EEE algorithm uses the objective functions $P(x|y) = \max_{x,\beta} x^\top Ay - \beta$ and $Q(y|x) = \max_{y,\alpha} x^\top By - \alpha$. Several objective functions may appear more intuitive, including simply minimizing the dual variable; $P(x|y) = -\beta$ and $Q(y|x) = -\alpha$. This requires less computation than the original EEE algorithm and therefore may save time since the objective function is recalculated for each linear program at each node.

Another pair of objective functions worth exploring are the empty objective functions: $P(x|y) = \max_{x,\beta} \ 0$ and $Q(y|x) = \max_{y,\alpha} \ 0$. Since the EEE-I implementation proceeds until there exists only one feasible solution, even the empty objective function will find all Nash Equilibria as the feasible set does not depend on the objective function. The fact that this objective function does not appear to move the system towards Nash Equilibria may be compensated for in running time by the fact that it searches for the first feasible point, and then terminates. This may avoid a large number of pivot steps, which appear to be a major determinant of the time spent by the algorithm on each node.

The last pair of objective functions explored in this section is developed directly from the endpoint of the algorithm, the search for points where $x_i(\alpha - (Ay)_i) = 0 \ \ \forall i \in M$ and $y_j(\beta - (x^\top B)_j) = 0 \quad \forall j \in N$. As x, y, A, B, α, and β are all nonnegative, the algorithm searches for the point that minimizes

$$x_i(\alpha - (Ay)_i) + (\beta - (x^\top B)_j)y_j \ \ \forall i \in M, j \in N.$$

If the minimized point has objective value 0, it is a Nash Equilibrium; otherwise it is not. A logical objective function to check is therefore

$$\min_{x,y,\alpha,\beta} x_i(\alpha - (Ay)_i) + (\beta - (x^\top B)_j)y_j \ \ \forall i \in M, j \in N$$

which can be rewritten in vector form with the all 1 vector $\mathbf{1}$ to remove the $\forall i \in M, j \in N$ as

$$\min_{x,y,\alpha,\beta} x(\mathbf{1} \cdot \alpha - Ay) + (\mathbf{1} \cdot \beta - x^\top B)y.$$

	Random (average) 17×17 57 NE	**Guessing Game** 22×22 3 NE	**Dollar Game** 10×10 91 NE
$P = \max_{x,\beta} x^\top Ay - \beta$ $Q = \max_{y,\alpha} x^\top By - \alpha$	185.74	43.75	126.21
$P = \max_{x,\beta} -\beta$ $Q = \max_{y,\alpha} -\alpha$	408.27	185.27	118.99
$P = \max_{x,\beta} 0$ $Q = \max_{y,\alpha} 0$	396.80	169.10	117.23
$P = \max_{x,\beta} x^\top (A+B)y - \beta$ $Q = \max_{y,\alpha} x^\top (A+B)y - \alpha$	145.65	32.66	121.83

Figure 24: Comparison of running time, nodes, and pivots for the 4 tested objective functions.

Multiplying and replacing the probability sums $x \cdot \mathbf{1} = 1$ and $\mathbf{1} \cdot y = 1$ yields

$$\min_{x,y,\alpha,\beta} \alpha - x^\top Ay + \beta - x^\top By$$

which in maximization form is

$$\max_{x,y,\alpha,\beta} x^\top Ay + x^\top By - \alpha - \beta = \max_{x,y,\alpha,\beta} x^\top (A+B)y - \alpha - \beta.$$

Note that the sum of game matrices is now used rather than each separately. Choosing over x already determines α while choosing over y already determines β, so the resulting objective functions are $P(x|y) = \max_{x,\beta} x^\top (A+B)y - \beta$ and $Q(y|x) = \max_{y,\alpha} x^\top (A+B)y - \alpha$.

Figure 24 provides the results of a series of tests meant to determine how the objective functions act on three types of games: a set of 5 random games, the guessing game, and the dollar game. No one objective function is the best in all three cases, however the objective functions $P(x|y) = \max_{x,\beta} x^\top (A+B)y - \beta$ and $Q(y|x) = \max_{y,\alpha} x^\top (A+B)y - \alpha$ seem to be best overall as they work far better in the random and guessing cases, and very slightly worse in the dollar case. Importantly, this objective function solves all three game classes faster than the original EEE objective functions $P(x|y) = \max_{x,\beta} x^\top Ay - \beta$ and $Q(y|x) = \max_{y,\alpha} x^\top By - \alpha$.

8 Conclusions

Variations on the EEE algorithm appear promising as a method of solving for all Nash Equilibria of bimatrix games, particularly for large games. This paper provides tangible improvements to the EEE family of algorithms. The implementation presented here uses only integer arithmetic in solution of the linear programs at each node of the EEE search tree. Rather than necessitating rounding in the linear program solution, the use of integer pivoting to solve linear programs removes any doubt of error due to rounding, which becomes more of a concern as the size of games increases. Integer pivoting appears not to increase the running time so much as to make it undesirable, though an exact comparison with the original EEE implementation is difficult because of differing machinery and programming languages.

The integer pivoting implementation also provides insight into the inner working of the algorithm. Introducing new constraints and its connection to the geometric properties of the algorithm, as well as the dimensions of the LP tableau, is clarified by the new implementation. The tableau need not be recalculated (other than the objective function) for each node; altering the tableau to incorporate the new constraint simultaneously checks for feasibility.

EEE in its original form performs a degeneracy check, regardless of the information in the search tree, that requires a large number of additional nodes. A contribution of this paper is to improve and clarify the degeneracy check, checking nodes past the NE level only where necessary. Constraints are fixed not until a certain point in the tree, but until the feasible set is reduced to one point or infeasibility. This drastically decreases the number of nodes, and thereby the running time in several cases, improving one of the previously weak points of the algorithm.

In the near future, the EEE-I implementation will be made publicly accessible as a tool for the academic community on a website with the LRS equilibrium enumeration implementation by Avis. Use by the academic community will hopefully lead to more improvements on the EEE family of algorithms and the EEE-I algorithm in particular.

References

[1] Audet, C., Hansen, P., Jaumard, B., and Savard, G. (2001) "Enumeration of All Extreme Equilibria of Bimatrix Games," *SIAM Journal of Scientific Computing* **23:1** pp. 323-338

[2] Avis, D. (2000) "*lrs*: A Revised Implementation of the Reverse Search Vertex Enumeration Algorithm," in *Polytopes - Combinatorics and Computation*, ed. Kalai, G. and Ziegler, G.M., DMV-Seminars, Birkhauser-Verlag, pp. 177-198

[3] Avis, D. (2005) gnash: An implementation of Nash Equilibrium enumeration using *LRS*, information available at `http://cgm.cs.mcgill.ca/~avis/C/lrs.html`, last accessed Dec. 6, 2005

[4] Azulay, D.-O. and Pique, J.-F. (2001) "A Revised Simplex Method with Integer Q-Matrices," *ACM Transactions on Mathematical Software* **27:3** pp. 350-360

[5] Chvatal, V. (1983) *Linear Programming*. Freeman, New York

[6] GAMUT User's Guide (2004) GAMUT Version 1.0.1, available at `http://gamut.stanford.edu`, last accessed Dec. 6, 2005

[7] Savani, R. An implementation of Nash Equilibrium enumeration using *LRS*, available on `http://banach.lse.ac.uk`, last accessed Dec. 6, 2005

[8] von Stengel, B. (2002) "Computing Equilibria for Two Person Games" Chapter 45 in *Handbook of Game Theory* **3**, ed. Aumann, R.J. and Hart, S., North-Holland, Amsterdam, pp. 1725-1756

A Java Implementation of the EEE-I Algorithm

This appendix contains the primary code written for the EEE-I implementation: EEE.java, Node.java, LPType.java, Bimatrix.java, NENode.java, NEList.java, and Tools.java. The code is included here to serve as a reference for readers interested in the implementation details, as broadly described in Section 5. Not included here is ColFormat.java, which is a simple formatting class.

```
import java.math.*;

public class EEE
// integer pivoting implementation of Audet et. al. (2001) "Enumeration of all
// extreme equilibria of bimatrix games" SIAM J. Sci. Comp. 23, 323-338
// with alterations as discussed in paper accompanying this code
{
    // constants
    static int   LCP        = 1;
    static int   CHOICE     = 0;

    // counters
    static int   pivotCount = 0;
    static int   count      = 0;

    NEList       NE         = new NEList();

    // command line arguments - see directions();
    static int[] cline      = new int[6];

    int          bound;

    public static void main(String[] args)
    {
        EEE run = new EEE();
        int r = 0, c = 0;

        // command line argument read-in
        for (int i = 0; i < cline.length && i < args.length; i++)
            cline[i] = Integer.parseInt(args[i]);

        if (args.length < 2)
        {
            directions();
            System.exit(999);
        }

        r = Integer.parseInt(args[0]);
        c = Integer.parseInt(args[1]);

        run.program(r, c);
    }

    int program(int r, int c)
    // main program class
    {
        // loads in the game and creates the queue with the initial node
        Bimatrix Game = new Bimatrix(r, c, "matrixA.txt", "matrixB.txt");

        bound = Game.row + Game.col;

        if (EEE.cline[2] == 0 || EEE.cline[2] == 1)
        {
            Game.output();
            System.out.print("Solving");
        } else if (EEE.cline[2] == 2)
            Game.latex();
```



```
        Node N = initialize(Game);

        if (EEE.cline[2] == 2)
            System.out.print("[0");
        bound = Game.row + Game.col;

        // solve the game recursively
        solveNode(N);

        if (EEE.cline[2] != 3)
            System.out.println();

        NE.print();

        if (EEE.cline[2] != 3)
            System.out.println();

        if (EEE.cline[2] == 0 || EEE.cline[2] == 3)
        {
            System.out.println("A total of " + count + " nodes.");
            System.out.println("A total of " + pivotCount + " pivots.");
        }

        return 0;
    }

    void solveNode(Node N)
    // the recursively called method for solving a node
    {
        count++;

        if (EEE.cline[2] == 1)
        {
            System.out.println("--------------------");
            System.out.println("Node #" + (count - 1) + " is node " + N);
            Tools.printVector(N.constraint);
        }

        // solves both linear programs for the given node - if either is
        // infeasible, moves to the next node in the stack
        if (!(N.doNodeLPs()))
        {
            if (EEE.cline[2] == 2)
                System.out.print("]");
            return;
        }
        // at levels < n + m creates two child nodes, each with one more
        // linear complimentarity condition satisfied
        if (N.depth < bound)
        {
            int index = N.nextTight();
            solveNode(N.addChild(index, LCP));
            solveNode(N.addChild(index, CHOICE));
        } else
        // at levels >= n + m, feasibility means a NE
        // degeneracy check if an LP has more than 1 column in general
        // or in all cases for the old version (cline[5] == 1)
```

```
        {
            NE.add(N.x, N.y);

            if (cline[5] != 1)
            {
                if (EEE.cline[2] == 2)
                    System.out.print("*");

                if (N.Q.LPCols == 1 && N.P.LPCols == 1)
                {
                    if (EEE.cline[2] == 2)
                        System.out.print("]");
                    return;
                }

                // determines the degeneracy check
                if (N.depth == bound)
                {
                    N.dindex = (N.P.LPCols > 1) ? 0 : N.x.length;
                    N.dend = (N.Q.LPCols > 1) ? N.x.length + N.y.length
                            : N.x.length;
                }
            }

            if (cline[5] == 1)
                old_degenerate(N);
            else
                degenerate(N);
        }

        if (EEE.cline[2] == 2)
            System.out.print("]");
    }

    void degenerate(Node N)
    // new degeneracy check
    {
        for (int i = N.dindex; i < N.dend; i++)
        {
            if (!N.constraint[i][LCP])
            {
                N.dindex = i + 1;
                solveNode(N.addChild(i, LCP));
            } else if (!N.constraint[i][CHOICE])
            {
                N.dindex = i + 1;
                solveNode(N.addChild(i, CHOICE));
            }
        }
    }

    void old_degenerate(Node N)
    // old degeneracy check
    {
        for (int i = 0; i < N.constraint.length; i++)
        {
            if (!N.constraint[i][LCP])
                solveNode(N.addChild(i, LCP));
```



```
            else if (!N.constraint[i][CHOICE])
                solveNode(N.addChild(i, CHOICE));

        }
    }

    LPType initQ(Bimatrix Game)
    // creates the initial LP for Q -- see initialization section of paper
    {
        BigInteger[][] A = Game.A;
        BigInteger[][] B = Game.B;

        BigInteger[][] LP = new BigInteger[Game.row + 2][Game.col + 1];

        int Qrows = LP.length;
        int Qcols = LP[0].length;

        int qhat = Tools.getLargestCol(Game.A, 0);

        for (int j = 0; j < Qcols - 2; j++)
        {
            LP[0][j] = A[qhat][0].subtract(A[qhat][j + 1]);
            LP[1][j] = BigInteger.ONE;
        }

        LP[0][Qcols - 2] = BigInteger.ONE.negate();
        LP[1][Qcols - 2] = BigInteger.ZERO;
        LP[0][Qcols - 1] = A[qhat][0].multiply(BigInteger.ONE);
        LP[1][Qcols - 1] = BigInteger.ONE;

        for (int i = 2; i < qhat + 2; i++)
        {
            for (int j = 0; j < Qcols - 2; j++)
                LP[i][j] = (A[qhat][0].subtract(A[qhat][j + 1]))
                        .add(A[i - 2][j + 1].subtract(A[i - 2][0]));
            LP[i][Qcols - 2] = BigInteger.ONE.negate();
            LP[i][Qcols - 1] = A[qhat][0].subtract(A[i - 2][0]);

        }
        for (int i = qhat + 2; i < Qrows - 1; i++)
        {
            for (int j = 0; j < Qcols - 2; j++)
                LP[i][j] = (A[qhat][0].subtract(A[qhat][j + 1]))
                        .add(A[i - 1][j + 1].subtract(A[i - 1][0]));
            LP[i][Qcols - 2] = BigInteger.ONE.negate();
            LP[i][Qcols - 1] = A[qhat][0].subtract(A[i - 1][0]);
        }

        LPType Q = new LPType();
        Q.LP = LP;
        Q.LPRows = Qrows;
        Q.LPCols = Qcols;
        Q.basis = determineBasis(Game.col, Game.row, qhat);
        Q.cobasis = determineCobasis(Game.col, Game.row, qhat);

        return Q;
    }
```

```
    int[] determineBasis(int primalVars, int dualVars, int hat)
    // fills the basis index vector
    {
        int[] basis = new int[dualVars + 1];

        basis[0] = 0;
        basis[1] = 1;
        for (int i = 0; i < hat; i++)
            basis[2 + i] = primalVars + i + 1;
        for (int i = hat + 1; i < dualVars; i++)
            basis[1 + i] = primalVars + i + 1;

        return basis;
    }

    int[] determineCobasis(int primalVars, int dualVars, int hat)
    // fills the cobasic index vector
    {
        int[] cobasis = new int[primalVars];

        for (int i = 0; i < cobasis.length - 1; i++)
            cobasis[i] = 2 + i;
        cobasis[cobasis.length - 1] = primalVars + hat + 1;

        return cobasis;
    }

    LPType initP(Bimatrix Game)
    // creates the initial LP for P -- see initialization section of paper
    {
        BigInteger[][] A = Game.A;
        BigInteger[][] B = Game.B;

        BigInteger[][] LP = new BigInteger[Game.col + 2][Game.row + 1];

        int Prows = LP.length;
        int Pcols = LP[0].length;

        int phat = Tools.getLargest(Game.B[0], 0, Game.B[0].length - 1);

        for (int j = 0; j < Pcols - 2; j++)
        {
            LP[0][j] = B[0][phat].subtract(B[j + 1][phat]);
            LP[1][j] = BigInteger.ONE;
        }
        LP[0][Pcols - 2] = BigInteger.ONE.negate();
        LP[1][Pcols - 2] = BigInteger.ZERO;
        LP[0][Pcols - 1] = B[0][phat].multiply(BigInteger.ONE);
        LP[1][Pcols - 1] = BigInteger.ONE;

        for (int i = 2; i < phat + 2; i++)
        {
            for (int j = 0; j < Pcols - 2; j++)
                LP[i][j] = (B[0][phat].subtract(B[j + 1][phat]))
                        .add(B[j + 1][i - 2].subtract(B[0][i - 2]));
            LP[i][Pcols - 1] = B[0][phat].subtract(B[0][i - 2]);
            LP[i][Pcols - 2] = BigInteger.ONE.negate();
        }
```



```
        for (int i = phat + 2; i < Prows - 1; i++)
        {
            for (int j = 0; j < Pcols - 2; j++)
                LP[i][j] = (B[0][phat].subtract(B[j + 1][phat]))
                        .add(B[j + 1][i - 1].subtract(B[0][i - 1]));
            LP[i][Pcols - 1] = B[0][phat].subtract(B[0][i - 1]);
            LP[i][Pcols - 2] = BigInteger.ONE.negate();
        }

        LPType P = new LPType();
        P.LP = LP;
        P.LPRows = Prows;
        P.LPCols = Pcols;
        P.basis = determineBasis(Game.row, Game.col, phat);
        P.cobasis = determineCobasis(Game.row, Game.col, phat);

        return P;
    }

    Node initialize(Bimatrix Game)
    // creates the initial node
    {
        LPType P = null;
        LPType Q = null;

        P = initP(Game);
        Q = initQ(Game);

        BigInteger[] x = new BigInteger[Game.row];
        x[0] = BigInteger.ONE;
        for (int i = 1; i < x.length; i++)
            x[i] = BigInteger.ZERO;
        BigInteger[] y = new BigInteger[Game.col];
        y[0] = BigInteger.ONE;

        for (int i = 1; i < y.length; i++)
            y[i] = BigInteger.ZERO;

        boolean[][] constraint = new boolean[Game.row + Game.col][2];

        Node N = new Node();
        N.constraint = Tools.copy(constraint);
        N.depth = 0;
        N.Game = Game;
        N.out = 0;
        N.P = P;
        N.Q = Q;
        N.x = x;
        N.y = y;
        N.alpha = BigInteger.ZERO;
        N.beta = BigInteger.ZERO;
        return N;
    }

    static void directions()
    {
        System.out
                .println("Invoke the program from the command line using");
```

```
        System.out.println("\"java EEE m n arg1 arg2 arg3 arg4\"");
        System.out
                .println("where m is the number of strategies for Player I");
        System.out
                .println("and n is the number of strategies for Player II.");
        System.out.println("arg1 − output type");
        System.out.println("\t 0: standard output [default]");
        System.out.println("\t 1: verbose output");
        System.out
                .println("\t 2: latex tree output (using synttree package)");
        System.out.println("arg2 − objective function");
        System.out
                .println("\t 0: P = x(A+B)y − alpha \t Q = x(A+B)y − beta [default]");
        System.out.println("\t 1: P = xAy − alpha \t \t Q = xBy − beta");
        System.out.println("\t 2: P = −alpha \t \t \t Q = −beta");
        System.out.println("\t 3: P = 0 \t \t \t Q = 0");
        System.out.println("arg3 − input type");
        System.out
                .println("\t 0: A in matrixA.txt, B in matrixB.txt [default]");
        System.out
                .println("\t 1: payoff matrices in Game Tracer form in GT.game");
        System.out.println("arg4 − degeneracy check");
        System.out.println("\t 0: new EEE−I check [default]");
        System.out.println("\t 1: old EEE check");
    }

}
```

```
import java.math.*;

public class Node
{
    int          NodeID;
    Bimatrix     Game;
    LPType       P, Q;
    BigInteger[] x, y;
    BigInteger   alpha, beta;
    boolean[][]  constraint;
    boolean      changedP;
    int          out  = 0;
    int          depth;
    static int   NONE = 0;
    BigInteger   PM, QM;
    boolean      removedRow;
    int          counter;
    int          dindex, dend;

    boolean doNodeLPs()
    // solves the LPs for the node. Determines which to solve first by checking
    // if P or Q was changed
    {
        BigInteger[][] A = (EEE.cline[3] == 1) ? Game.A : Game.sum;
        BigInteger[][] B = (EEE.cline[3] == 1) ? Game.B : Game.sum;

        if (changedP)
        {
            updateObjective(P, Tools.matrixTimesVector(A, y));
            printState("Before pivots P:", P);
            if (!P.IP(out))
                return false;
            updateVars(P, x, Game.row + 1, 'P');
            printState("After pivots P:", P);
            updateObjective(Q, Tools.vectorTimesMatrix(x, B));
            printState("Before pivots Q:", Q);
            Q.IP(NONE);
            updateVars(Q, y, Game.col + 1, 'Q');
            printState("After pivots Q:", Q);
        } else
        {
            updateObjective(Q, Tools.vectorTimesMatrix(x, B));
            printState("Before pivots Q:", Q);
            if (!Q.IP(out))
                return false;

            updateVars(Q, y, Game.col + 1, 'Q');
            printState("After pivots Q:", Q);
            updateObjective(P, Tools.matrixTimesVector(A, y));
            printState("Before pivots P:", P);
            P.IP(NONE);
            updateVars(P, x, Game.row + 1, 'P');
            printState("After pivots P:", P);
        }

        return true;
    }
```



```
    void updateVars(LPType L, BigInteger[] primal, int length, char type)
    // updates variables using the finished tableau, to be run after completed
    // LP solution
    {
        // go through each element of the basis and update it's value, which is
        // the last element of that row divided by <
        int k;

        for (int i = 0; i < primal.length; i++)
            if ((k = Tools.lookup(L.basis, i + 1)) != -1)
                primal[i] = L.LP[k][L.LPCols - 1];
            else
                primal[i] = BigInteger.ZERO;

        if (type == 'P')
        {
            PM = L.M;
            beta = L.LP[0][L.LPCols - 1];
        } else if (type == 'Q')
        {
            QM = L.M;
            alpha = L.LP[0][L.LPCols - 1];
        } else
            System.out.println("ERROR!");
    }

    Node addChild(int index, int type)
    // adds a new node to the stack derived from the parent node, with one
    // additional constraint added
    // H.out is calculated by determining if the variables is CHOICE or LCP,
    // then adding the appropriate constant
    {
        Node H = new Node();

        if (EEE.cline[2] == 1)
            System.out.println(this + " parents " + H);

        if (EEE.cline[2] == 2)
            System.out.print("[" + EEE.count);
        H.changedP = ((index < Game.row && type == 0) ||
                (index >= Game.row && type == 1));

        if (!H.changedP)
        {
            if (type == EEE.CHOICE)
                H.out = index - Game.row + 1;
            else
                H.out = index + Game.col + 1;
        } else
            H.out = index + 1;

        H.constraint = Tools.copy(constraint);
        H.constraint[index][type] = true;
        H.depth = depth + 1;
        H.Game = Game;
        H.counter = EEE.count;
        H.P = P.copy();
```

```
        H.Q = Q.copy();
        H.x = Tools.copy(x);
        H.y = Tools.copy(y);
        H.NodeID = NodeID + 1;
        H.alpha = alpha.add(BigInteger.ZERO);
        H.beta = beta.add(BigInteger.ZERO);
        H.dindex = index;
        H.dend = dend;

        return H;
    }

    int nextTight()
    // returns the index of the next variable to be made tight
    {
        BigInteger[][] val = new BigInteger[2][constraint.length];
        BigInteger[] xB = Tools.vectorTimesMatrix(x, Game.B);
        BigInteger[] Ay = Tools.matrixTimesVector(Game.A, y);

        BigInteger firstTerm;
        BigInteger secondTerm;

        for (int i = 0; i < Game.row; i++)
        {
            if (constraint[i][EEE.CHOICE] || constraint[i][EEE.LCP])
            {
                val[0][i] = BigInteger.ONE.negate();
                val[1][i] = BigInteger.ONE;

            } else
            {
                firstTerm = x[i].multiply(alpha).multiply(QM);
                secondTerm = x[i].multiply(QM).multiply(QM);
                val[0][i] = firstTerm.subtract(secondTerm);
                val[1][i] = PM.multiply(QM).multiply(QM);
            }
        }
        for (int i = 0; i < Game.col; i++)
        {
            if (constraint[i + Game.row][EEE.CHOICE]
                    || constraint[i + Game.row][EEE.LCP])
            {
                val[0][i + Game.row] = BigInteger.ONE.negate();
                val[1][i + Game.row] = BigInteger.ONE;
            } else
            {
                firstTerm = y[i].multiply(beta).multiply(PM);
                secondTerm = y[i].multiply(xB[i]).multiply(PM);
                val[0][i + Game.row] = firstTerm.subtract(secondTerm);
                val[1][i + Game.row] = QM.multiply(PM).multiply(PM);
            }
        }

        for (int i = 0; i < val[1].length; i++)
            if (val[1][i].compareTo(BigInteger.ZERO) == 0)
                val[1][i] = BigInteger.ONE;

        return Tools.getLargest(val);
```



```
    }

    void updateObjective(LPType W, BigInteger[] product)
    // recalculates the objective function in terms of the current tableau
    {
        int z = W.LPRows - 1;
        int b = W.LPCols - 1;
        int pos;
        BigInteger max = BigInteger.ONE;

        Tools.fillVector(W.LP[z], 0);

        if (EEE.cline[3] == 0 || EEE.cline[3] == 1)
        {
            for (int i = 0; i < product.length; i++)
            {
                if (product[i].compareTo(max) > 0)
                    max = product[i].multiply(BigInteger.ONE);
            }

            for (int i = 0; i < product.length; i++)
            {
                // if variable i is in the basis (corresponding to index i+1)
                // then the row corresponding to that variable must be added
                // (multiplied by the vector entry) to the last row
                if ((pos = Tools.lookup(W.basis, i + 1)) != -1)
                    for (int j = 0; j < W.LP[z].length; j++)
                        W.LP[z][j] = W.LP[z][j].add(W.LP[pos][j]
                                .multiply(product[i]));
                // if variable i is in the cobasis, the column corresponding to
                // that variable is added (times the vector entry) to the last
                // row
                else if ((pos = Tools.lookup(W.cobasis, i + 1)) != -1)
                    W.LP[z][pos] = W.LP[z][pos].subtract(product[i]);
            }

        }

        if (EEE.cline[3] != 3)
        {
            for (int j = 0; j < W.LP[z].length; j++)
                W.LP[z][j] = W.LP[z][j].subtract(W.LP[0][j]);
        }
    }

    void printState(String S, LPType LPT)
    // output routine - prints the current state
    {
        if (EEE.cline[2] == 1)
        {
            BigInteger[][] LP = LPT.LP;

            System.out.println(S);

            System.out.println("Depth = " + depth);
            System.out.println("Out = " + out);
            System.out.print("Cobasis = ");
            Tools.printVector(LPT.cobasis);
```

```
            System.out.print("Basis = ");
            Tools.printVector(LPT.basis);
            System.out.println("M = " + LPT.M);
            ColFormat.print(LP);
            System.out.print("x = (");
            for (int i = 0; i < x.length - 1; i++)
                System.out.print(x[i] + ",");

            System.out.print(x[x.length - 1] + ")");
            System.out.println();
            System.out.print("y = (");
            for (int i = 0; i < y.length - 1; i++)
                System.out.print(y[i] + ",");
            System.out.print(y[y.length - 1]);
            System.out.println(")");
        }

    }
}
```

```
import java.math.*;

public class LPType
// integer linear program solving class
{
    BigInteger[][] LP;
    int            LPRows, LPCols;
    BigInteger     M        = BigInteger.ONE;
    int[]          basis, cobasis;
    int            countpiv = 0;

    public boolean IP(int out)
    {
        int pivotRow, pivotCol;

        if (!removeVar(out))
        {
            if (EEE.cline[2] == 1)
                System.out.println("Rid Unsuccessful");
            return false;
        }

        // solve the linear program
        while ((pivotCol = Tools
                .getSmallest(LP[LPRows - 1], 0, LPCols - 2)) != -1)
        {
            pivotRow = findPivotRow(pivotCol);
            pivot(pivotRow, pivotCol);
        }

        return true;
    }

    public boolean removeVar(int out)
    // determines feasibility by attempting to remove variable out from
    // LP - removes if possible
    {
        int pivotRow = -1, pivotCol, pos;
        int remove = Tools.lookup(cobasis, out);

        if (out == 0)
            return true;

        if (remove != -1)
        {
            if (EEE.cline[2] == 1)
                System.out.println("In the cobasis, column " + remove);
            deleteCol(remove);
            return true;
        }

        remove = Tools.lookup(basis, out);
        if (EEE.cline[2] == 1)
        {
            System.out.println("In the basis, row " + remove);
            printState("Before trying to rid:");
        }
```



```
        if (LP[remove][LPCols - 1].compareTo(BigInteger.ZERO) == 0)
            return removeRowZero(remove);

        while ((pivotCol = Tools.getLargest(LP[remove], 0, LPCols - 2)) != -1)
        {
            pivotRow = findPivotRow(pivotCol);
            pivot(pivotRow, pivotCol);

            if (pivotRow == remove)
            {
                deleteCol(pivotCol);
                return true;
            } else
            {
                remove = Tools.lookup(basis, out);
                if (LP[remove][LPCols - 1].compareTo(BigInteger.ZERO) == 0)

                    return removeRowZero(remove);
            }
        }

        return false;
    }

    public boolean removeRowZero(int remove)
    // if the value of the basic variable is zero, pivots it out (if possible)
    // and removes the variable. If all elements are zero in the row, removes
    // the row
    {
        int pos = Tools.getLargestAbs(LP[remove], 0, LPCols - 2,
                BigInteger.ZERO);
        if (pos == -1)
            deleteRow(remove);
        else
        {
            if (LP[remove][pos].compareTo(BigInteger.ZERO) < 0)
                for (int j = 0; j < LPCols; j++)
                    LP[remove][j] = LP[remove][j].negate();
            pivot(remove, pos);
            deleteCol(pos);
        }
        return true;
    }

    void pivot(int pivotRow, int pivotCol)
    // the pivot procedure
    {
        BigInteger pivotPoint = LP[pivotRow][pivotCol];
        BigInteger factor;

        countpiv++;
        if (countpiv > 10000)
        {
            pivotCol = bland_rule();
            pivotRow = findPivotRow(pivotCol);
            pivotPoint = LP[pivotRow][pivotCol];
        }
```

```
        EEE.pivotCount++;

        if (EEE.cline[2] == 1)
        {
            System.out.println(cobasis[pivotCol] + " will enter.");
            System.out.println(basis[pivotRow] + " will exit.");
        }

        // multiply each row other than the pivot row by the pivot element
        for (int i = 0; i < pivotRow; i++)
            for (int j = 0; j < LPCols; j++)
                LP[i][j] = LP[i][j].multiply(pivotPoint);
        for (int i = pivotRow + 1; i < LPRows; i++)
            for (int j = 0; j < LPCols; j++)
                LP[i][j] = LP[i][j].multiply(pivotPoint);

        // determine the factor needed to make the pivot column a multiple of
        // an identity column, then subtract off that multiple from each row.
        // replace the pivot column (move it out of the basis) with the
        // appropriate nonbasic column
        for (int i = 0; i < pivotRow; i++)
        {
            factor = LP[i][pivotCol].divide(pivotPoint);
            if (factor.compareTo(BigInteger.ZERO) == 0)
                continue;
            for (int j = 0; j < LPCols; j++)
                if (j == pivotCol)
                    LP[i][j] = factor.multiply(M).negate();
                else
                    LP[i][j] = LP[i][j].subtract(LP[pivotRow][j]
                            .multiply(factor));
        }
        LP[pivotRow][pivotCol] = M;
        for (int i = pivotRow + 1; i < LPRows; i++)
        {
            factor = LP[i][pivotCol].divide(pivotPoint);
            if (factor.compareTo(BigInteger.ZERO) == 0)
                continue;
            for (int j = 0; j < LPCols; j++)
                if (j == pivotCol)
                    LP[i][j] = factor.multiply(M).negate();
                else
                    LP[i][j] = LP[i][j].subtract(LP[pivotRow][j]
                            .multiply(factor));
        }
        // change the basis and cobasis indices to reflect the change
        int temp = cobasis[pivotCol];
        cobasis[pivotCol] = basis[pivotRow];
        basis[pivotRow] = temp;

        // divide each element of the matrix not in the pivot row by M
        for (int i = 0; i < pivotRow; i++)
            for (int j = 0; j < LPCols; j++)
                LP[i][j] = LP[i][j].divide(M);
        for (int i = pivotRow + 1; i < LPRows; i++)
            for (int j = 0; j < LPCols; j++)
                LP[i][j] = LP[i][j].divide(M);
```



```
        // replace M with the new value
        M = pivotPoint;

        printState("End of pivot step.");
    }

    int bland_rule()
    // switch to Bland's rule after a large number of pivots in the
    // same LP to avoid cycling
    {
        if (EEE.cline[2] == 3)
            System.out.println("Bland's rule being used.");

        for (int i = 0; i < LP[0].length - 1; i++)
            if (LP[LPRows - 1][i].compareTo(BigInteger.ZERO) < 0)
                return i;

        return -1;
    }

    int findPivotRow(int pivotCol)
    // the pivot row should be the row for which the ratio of the
    // b column to the pivot column has smallest absolute value.
    {
        int b = LPCols - 1;
        int min = 1;
        BigInteger min_constant = LP[min][b];
        BigInteger min_coeff = LP[min][pivotCol];

        for (int i = min + 1; i < LPRows - 1; i++)
        {
            BigInteger left = min_constant.multiply(LP[i][pivotCol]);
            BigInteger right = min_coeff.multiply(LP[i][b]);
            if (left.compareTo(right) > 0)
            {
                min_coeff = LP[i][pivotCol];
                min_constant = LP[i][b];
                min = i;
            }
        }

        return min;
    }

    void deleteRow(int row)
    // removes row from the LP
    {
        // swap the row to remove with the last
        for (int i = 0; i < LPCols; i++)
        {
            LP[row][i] = LP[LPRows - 2][i];
            LP[LPRows - 2][i] = LP[LPRows - 1][i];
        }

        BigInteger[][] to = new BigInteger[LP.length - 1][LP[0].length];
        for (int i = 0; i < to.length; i++)
            for (int j = 0; j < to[0].length; j++)
```

```
                to[i][j] = LP[i][j].add(BigInteger.ZERO);

        LP = to;

        basis[row] = basis[basis.length - 1];
        int[] b = new int[basis.length - 1];
        for (int i = 0; i < b.length; i++)
            b[i] = basis[i];
        basis = b;
        LPRows--;

    }

    void deleteCol(int col)
    // removes column col from the LP
    {
        // swap the column to remove with the last
        for (int i = 0; i < LPRows; i++)
        {
            LP[i][col] = LP[i][LPCols - 2];
            LP[i][LPCols - 2] = LP[i][LPCols - 1];
        }

        BigInteger[][] to = new BigInteger[LP.length][LP[0].length - 1];
        for (int i = 0; i < to.length; i++)
            for (int j = 0; j < to[0].length; j++)
                to[i][j] = LP[i][j].add(BigInteger.ZERO);

        LP = to;
        cobasis[col] = cobasis[cobasis.length - 1];
        int[] cob = new int[cobasis.length - 1];
        for (int i = 0; i < cob.length; i++)
            cob[i] = cobasis[i];
        cobasis = cob;
        LPCols--;
    }

    public LPType copy()
    // returns a copy of the current LP
    {
        LPType to = new LPType();

        to.LP = Tools.copy(LP);
        to.basis = Tools.copy(basis);
        to.cobasis = Tools.copy(cobasis);
        to.LPCols = LPCols;
        to.LPRows = LPRows;
        to.M = M;

        return to;
    }

    public void printState(String S)
    // output - prints the state
    {
        if (EEE.cline[2] == 1)
        {
            System.out.println(S);
```



```
            System.out.println("M is: " + M);
            System.out.print("Basis =");
            Tools.printVector(basis);
            System.out.print("Cobasis =");
            Tools.printVector(cobasis);
            ColFormat.print(LP);
        }

    }

}
```

```
import java.util.*;
import java.io.*;
import java.math.*;

public class Bimatrix
{
    int           row;
    int           col;
    BigInteger[][] A;
    BigInteger[][] B;
    BigInteger[][] sum;

    Bimatrix(int r, int c, String aname, String bname)
    // constructs an r x c Bimatrix with the matrices in the files passed
    // checks to make sure inputted matrix has no zero payoff strategies
    {
        row = r;
        col = c;
        A = new BigInteger[row][col];
        B = new BigInteger[row][col];

        if (EEE.cline[4] == 1)
            GTMatrices(A, B);
        else
        {
            readMatrix(A, aname);
            readMatrix(B, bname);
        }

        if (hasZero(A) || hasZero(B))
        {
            System.out
                    .println("Current functionality requires positive payoffs.");
            System.exit(999);
        }

        getSum();
    }

    public void getSum()
    // Game.sum = Game.A + Game.B
    {
        sum = new BigInteger[A.length][A[0].length];
        for (int i = 0; i < A.length; i++)
            for (int j = 0; j < A[0].length; j++)
                sum[i][j] = A[i][j].add(B[i][j]);

    }

    public void GTMatrices(BigInteger[][] matA, BigInteger[][] matB)
    // reads input for Game Tracer type
    // input procedure determined from Flanagan, Java In A Nutshell
    {
        try
        {
            BufferedReader GameFile = new BufferedReader(new FileReader(
                    "GT.game"));
```



```
            String s = GameFile.readLine();
            s = GameFile.readLine();
            s = GameFile.readLine();

            s = GameFile.readLine();
            java.util.StringTokenizer t = new java.util.StringTokenizer(s);
            for (int j = 0; j < col; j++)
                for (int i = 0; i < row; i++)
                {
                    matA[i][j] = new BigInteger(Integer.parseInt(t
                            .nextToken())
                            + "");
                    if (matA[i][j].compareTo(BigInteger.ZERO) < 0)
                        throw new IllegalArgumentException();
                }
            s = GameFile.readLine();
            t = new java.util.StringTokenizer(s);
            for (int j = 0; j < col; j++)
                for (int i = 0; i < row; i++)
                {
                    matB[i][j] = new BigInteger(Integer.parseInt(t
                            .nextToken())
                            + "");
                    if (matB[i][j].compareTo(BigInteger.ZERO) < 0)
                        throw new IllegalArgumentException();
                }

        } catch (NoSuchElementException a)
        {
            EEE.directions();
            System.out.println("Error: Matrix not large enough for task.");
            System.exit(999);
        } catch (IOException b)
        {
            EEE.directions();
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        } catch (NumberFormatException c)
        {
            EEE.directions();
            System.out
                    .println("Error: Matrix values are not all integral.");
            System.exit(999);
        } catch (IllegalArgumentException d)
        {
            System.out
                    .println("Error: Currently requires non-negative payoffs.");
            System.exit(999);
        } catch (IndexOutOfBoundsException e)
        {
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        } catch (NullPointerException f)
        {
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        }
```

```
        ;
    }

    public void readMatrix(BigInteger[][] mat, String filename)
    // reads in filename into the passed matrix
    // input procedure determined from Flanagan, Java In A Nutshell
    {
        try
        {
            BufferedReader size = new BufferedReader(new FileReader(
                    filename));

            for (int i = 0; i < row; i++)
            {
                String s = size.readLine();
                java.util.StringTokenizer t = new java.util.StringTokenizer(
                        s);
                for (int j = 0; j < col; j++)
                {
                    mat[i][j] = new BigInteger(Integer.parseInt(t
                            .nextToken())
                            + "");
                    if (mat[i][j].compareTo(BigInteger.ZERO) < 0)
                        throw new IllegalArgumentException();
                }
            }

        } catch (NoSuchElementException a)
        {
            System.out.println("Error: Matrix not large enough for task.");
            System.exit(999);
        } catch (IOException b)
        {
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        } catch (NumberFormatException c)
        {
            System.out
                    .println("Error: Matrix values are not all integral.");
            System.exit(999);
        } catch (IllegalArgumentException d)
        {
            System.out
                    .println("Error: Currently requires non-negative payoffs.");
            System.exit(999);
        } catch (IndexOutOfBoundsException e)
        {
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        } catch (NullPointerException f)
        {
            System.out.println("Error: Input problem. Check matrices.");
            System.exit(999);
        }
        ;
    }

    public void output()
```



```
    // output method for the entire bimatrix
    {
        System.out.println("The matrix is of size " + row + " by " + col
                + ".");
        System.out.println("Matrix A:");
        ColFormat.print(A);
        System.out.println("Matrix B:");
        ColFormat.print(B);
    }

    public void latex()
    // latex output method for the entire bimatrix
    {
        System.out.print("$$\\begin{array}{ccc}");
        System.out.print("A = \\left(\\begin{array}{");
        for (int i = 0; i < col; i++)
            System.out.print("c");
        System.out.print("}");
        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < col - 1; j++)
                System.out.print(A[i][j] + "&");
            System.out.print(A[i][col - 1]);
            if (i != row - 1)
                System.out.print("\\\\");
        }
        System.out
                .print("\\end{array}\\right) & \\; \\; \\; \\; \\; \\; & B = \\left(\\begin{array}{");
        for (int i = 0; i < col; i++)
            System.out.print("c");
        System.out.print("}");
        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < col - 1; j++)
                System.out.print(B[i][j] + "&");
            System.out.print(B[i][col - 1]);
            if (i != row - 1)
                System.out.print("\\\\");
        }
        System.out.print("\\end{array}\\right)");
        System.out.println("\\end{array}$$");
        System.out.print("\\synttree");
    }

    boolean hasZero(BigInteger[][] mat)
    {

        for (int i = 0; i < mat.length; i++)
        {
            for (int j = 0; j < mat[0].length; j++)
                if (mat[i][j].compareTo(BigInteger.ZERO) == 0)
                    return true;

        }
        return false;
    }

}
```

```
import java.math.*;

public class NENode
{
    BigInteger[][] x;
    int            xTag = -1;
    BigInteger[][] y;
    int            yTag = -1;
    NENode         next;
}
```

```
import java.math.BigInteger;

public class NEList
// Nash Equilibrium list
{
    NENode     root;
    static int maxXTag = 1;
    static int maxYTag = 1;

    void add(BigInteger[] x_, BigInteger[] y_)
    // adds a given equilibrium pair to the NEList if not already present
    {
        BigInteger xsum = BigInteger.ZERO, ysum = BigInteger.ZERO;

        BigInteger[][] x = new BigInteger[2][x_.length];
        BigInteger[][] y = new BigInteger[2][y_.length];

        for (int i = 0; i < x_.length; i++)
        {
            x[0][i] = x_[i];
            xsum = xsum.add(x_[i]);
        }
        for (int i = 0; i < y_.length; i++)
        {
            y[0][i] = y_[i];
            ysum = ysum.add(y_[i]);
        }

        Tools.fillVector(x[1], xsum);
        Tools.fillVector(y[1], ysum);

        NENode NE = new NENode();

        NE.x = normalize(x);
        NE.y = normalize(y);

        if (root == null)
        {
            root = NE;
            if (EEE.cline[2] == 0)
                System.out.print(".");
            if (EEE.cline[2] == 2)
                System.out.print("*");
            NE.xTag = 0;
            NE.yTag = 0;
            return;
        }

        if (!present(NE))
        {
            if (EEE.cline[2] == 0)
                System.out.print(".");
            if (EEE.cline[2] == 2)
                System.out.print("*");
            if (NE.xTag == -1)
            {
                NE.xTag = NEList.maxXTag;
```



```
                NEList.maxXTag++;
            }
            if (NE.yTag == -1)
            {
                NE.yTag = NEList.maxYTag;
                NEList.maxYTag++;
            }

            NENode H = root;
            while (H.next != null)
                H = H.next;
            H.next = NE;
        }
    }

    boolean isEqualY(NENode N, BigInteger[][] y)
    // checks if strategy y is already present
    {
        for (int i = 0; i < y[0].length; i++)
            if ((y[0][i].compareTo(N.y[0][i])) != 0
                    || (y[1][i].compareTo(N.y[1][i])) != 0)
                return false;
        return true;
    }

    boolean isEqualX(NENode N, BigInteger[][] x)
    //  checks if strategy x is already present
    {
        for (int i = 0; i < x[0].length; i++)
            if ((x[0][i].compareTo(N.x[0][i])) != 0
                    || (x[1][i].compareTo(N.x[1][i])) != 0)
                return false;

        return true;
    }

    boolean present(NENode NE)
    // determines whether a Nash Equilibrium pair is already present
    {
        NENode N = root;

        while (N != null)
        {
            if (isEqualX(N, NE.x) && isEqualY(N, NE.y))
                return true;
            else if (isEqualX(N, NE.x))
                NE.xTag = N.xTag;
            else if (isEqualY(N, NE.y))
                NE.yTag = N.yTag;

            N = N.next;
        }

        return false;
    }

    int print()
    // prints out the Nash Equilibrium list
```

```
    {
        int counter = 1;
        int length = length();
        NENode N = root;

        N = root;
        String[][] NEs = new String[length][N.x[0].length + N.y[0].length
                + 10];
        boolean[][] connected = new boolean[NEList.maxXTag
                + NEList.maxYTag][NEList.maxYTag + NEList.maxXTag];

        System.out.println(length + " extreme equilibria.");

        if (EEE.cline[2] != 3)
        {
            for (int i = 0; i < length; i++)
            {
                NEs[i][0] = ("EE");
                NEs[i][1] = ("#" + (i + 1));
                NEs[i][2] = "x =";
                NEs[i][3] = ("{" + (N.xTag + 1) + "}");
                NEs[i][4] = "(";
                for (int j = 0; j < N.x[0].length; j++)
                {
                    if (N.x[0][j].compareTo(BigInteger.ZERO) == 0)
                        NEs[i][j + 5] = "0";
                    else if (N.x[1][j].compareTo(BigInteger.ONE) == 0)
                        NEs[i][j + 5] = "1";
                    else
                        NEs[i][j + 5] = (N.x[0][j] + "/" + N.x[1][j]);

                }
                NEs[i][N.x[0].length + 5] = ")";
                NEs[i][N.x[0].length + 6] = "\ty =";
                NEs[i][N.x[0].length + 7] = ("{" + (N.yTag + 1) + "}");
                NEs[i][N.x[0].length + 8] = "(";
                for (int j = 0; j < N.y[0].length; j++)
                {
                    if (N.y[0][j].compareTo(BigInteger.ZERO) == 0)
                        NEs[i][N.x[0].length + j + 9] = "0";
                    else if (N.y[1][j].compareTo(BigInteger.ONE) == 0)
                        NEs[i][N.x[0].length + j + 9] = "1";
                    else
                        NEs[i][N.x[0].length + j + 9] = (N.y[0][j] + "/"
                                + N.y[1][j]);

                }
                NEs[i][N.x[0].length + N.y[0].length + 9] = ")";

                connected[N.xTag][N.yTag + NEList.maxXTag] = true;
                connected[N.yTag + NEList.maxXTag][N.xTag] = true;

                N = N.next;
            }

            ColFormat.print(NEs);
        }
```



```
        // returns number of extreme equilibria
        return length;

    }

    BigInteger[][] normalize(BigInteger vec[][])
    // normalizes the vector so that it is a probability vector
    {
        BigInteger[][] normed = new BigInteger[vec.length][vec[0].length];
        BigInteger sum = BigInteger.ZERO;
        BigInteger gcd = BigInteger.ZERO;

        for (int i = 0; i < vec[1].length; i++)
        {
            gcd = vec[0][i].gcd(vec[1][i]);
            normed[1][i] = vec[1][i].divide(gcd);
            normed[0][i] = vec[0][i].divide(gcd);
        }

        return normed;
    }

    int length()
    // determines the number of equilibria in the list
    {
        NENode N = root;
        int counter = 0;

        while (N != null)
        {
            counter++;
            N = N.next;
        }

        return counter;
    }
}
```

```
import java.math.*;

public class Tools
// tools for use in EEE-I implementation
{
    public static int lookup(int[] vector, int search)
    {
        for (int i = 0; i < vector.length; i++)
            if (vector[i] == search)
                return i;

        return -1;
    }

    public static int getLargestAbs(BigInteger[] row, int first, int last,
            BigInteger except)
    {
        BigInteger max = except;
        int max_index = -1;
        int i = first;
        while (row[i].compareTo(except) == 0 && i < last)
            i++;
        max = row[i];
        max_index = i;
        if (max.compareTo(except) == 0)
            return -1;

        while (i <= last)
        {
            if (row[i].abs().compareTo(max) > 0
                    && row[i].compareTo(except) != 0)
            {
                max = row[i].abs();
                max_index = i;
            }
            i++;
        }
        return max_index;
    }

    public static int getLargest(BigInteger[] row, int first, int last)
    {
        BigInteger max = BigInteger.ZERO;
        int max_index = -1;
        for (int i = first; i <= last; i++)
        {
            if (row[i].compareTo(max) > 0)
            {
                max = row[i];
                max_index = i;
            }
        }
        return max_index;
    }

    public static int getSmallest(BigInteger[] row, int first, int last)
    {
```



```
        BigInteger min = BigInteger.ZERO;
        int min_index = -1;
        for (int i = first; i <= last; i++)
        {
            if (row[i].compareTo(min) < 0)
            {
                min = row[i];
                min_index = i;
            }
        }
        return min_index;
    }

    static int getLargest(BigInteger[][] mat)
    {
        int max = 0;
        BigInteger max_num = mat[0][0];
        BigInteger max_den = mat[1][0];

        for (int i = 1; i < mat[0].length; i++)
        {
            BigInteger left = max_num.multiply(mat[1][i]);
            BigInteger right = max_den.multiply(mat[0][i]);
            if ((right.compareTo(left) > 0))
            {
                max_num = mat[0][i];
                max_den = mat[1][i];
                max = i;
            }
        }

        return max;
    }

    static int getLargestCol(BigInteger[][] matrix, int column)
    {
        BigInteger largest = BigInteger.ZERO;
        int largestindex = -1;

        for (int i = 0; i < matrix.length; i++)
            if (matrix[i][column].compareTo(largest) > 0)
            {
                largest = matrix[i][column];
                largestindex = i;
            }

        return largestindex;
    }

    static BigInteger[] matrixTimesVector(BigInteger[][] mat,
            BigInteger[] vec)
    {
        BigInteger[] sol = new BigInteger[mat.length];
        for (int i = 0; i < sol.length; i++)
            sol[i] = BigInteger.ZERO;
        for (int i = 0; i < mat.length; i++)
            for (int j = 0; j < mat[0].length; j++)
                sol[i] = sol[i].add(mat[i][j].multiply(vec[j]));
```

```
        return sol;
    }

    static BigInteger[] vectorTimesMatrix(BigInteger[] vec,
            BigInteger[][] mat)
    {
        BigInteger[] sol = new BigInteger[mat[0].length];
        for (int i = 0; i < sol.length; i++)
            sol[i] = BigInteger.ZERO;

        for (int i = 0; i < sol.length; i++)
            for (int j = 0; j < mat.length; j++)
                sol[i] = sol[i].add(mat[j][i].multiply(vec[j]));

        return sol;
    }

    public static void printVector(BigInteger[] vector)
    {
        System.out.print("(");
        for (int i = 0; i < vector.length; i++)
            System.out.print(vector[i] + " ");
        System.out.println(")");
    }

    public static void printVector(boolean[][] constraint)
    {
        for (int i = 0; i < constraint.length; i++)
            System.out.println(constraint[i][0] + "\t" + constraint[i][1]);
    }

    public static void printVector(int[] vector)
    {
        System.out.print("(");
        for (int i = 0; i < vector.length; i++)
            System.out.print(vector[i] + " ");
        System.out.println(")");
    }

    static int[] copy(int[] from)
    {
        int[] to = new int[from.length];

        for (int i = 0; i < to.length; i++)
            to[i] = from[i];
        return to;
    }

    static boolean[][] copy(boolean[][] from)
    {
        boolean[][] to = new boolean[from.length][from[0].length];

        for (int i = 0; i < to.length; i++)
            for (int j = 0; j < to[0].length; j++)
                to[i][j] = from[i][j];
        return to;
    }
```



```
    static BigInteger[][] copy(BigInteger[][] from)
    {
        BigInteger[][] to = new BigInteger[from.length][from[0].length];

        for (int i = 0; i < to.length; i++)
            for (int j = 0; j < to[0].length; j++)
                to[i][j] = from[i][j].add(BigInteger.ZERO);
        return to;
    }

    static BigInteger[] copy(BigInteger[] from)
    {
        BigInteger[] to = new BigInteger[from.length];

        for (int i = 0; i < to.length; i++)
            to[i] = from[i].add(BigInteger.ZERO);
        return to;
    }

    static void fillVector(BigInteger[] vector, int fill)
    {
        for (int i = 0; i < vector.length; i++)
            vector[i] = new BigInteger(fill + "");
    }

    static void fillVector(BigInteger[] vector, BigInteger fill)
    {
        for (int i = 0; i < vector.length; i++)
            vector[i] = fill.multiply(BigInteger.ONE);
    }
}
```