4

Computing Solution Concepts of Normal-Form Games

The discussion of strategies and solution concepts in Chapter 3 largely ignored issues of computation. We start by asking the most basic question: How hard is it to compute the Nash equilibria of a game? The answer turns out to be quite subtle, and to depend on the class of games being considered.

We have already seen how to compute the Nash equilibria of simple games. These calculations were deceptively easy, partly because there were only two players and partly because each player had only two actions. In this chapter we discuss several different classes of games, starting with the simple two-player, zero-sum normal-form game. Dropping only the zero-sum restriction yields a problem of different complexity—while it is generally believed that any algorithm that guarantees a solution must have an exponential worst case complexity, it is also believed that a proof to this effect may not emerge for some time. We also consider procedures for n-player games. In each case, we describe how to formulate the problem, the algorithm (or algorithms) commonly used to solve them, and the complexity of the problem. While we focus on the problem of finding a sample Nash equilibrium, we will briefly discuss the problem of finding all Nash equilibria and finding equilibria with specific properties. Along the way we also discuss the computation of other game-theoretic solution concepts: maxmin and minmax strategies, strategies that survive iterated removal of dominated strategies, and correlated equilibria.

4.1 Computing Nash equilibria of two-player, zero-sum games

The class of two-player, zero-sum games is the easiest to solve. The Nash equilibrium problem for such games can be expressed as a *linear program (LP)*, which means that equilibria can be computed in polynomial time.[1] Consider a two-player, zero-sum game $G = (\{1, 2\}, A_1 \times A_2, (u_1, u_2))$. Let U_i^* be the expected utility for player i in equilibrium (the value of the game); since the game is zero-sum, $U_1^* = -U_2^*$. The minmax theorem (see Section 3.4.1 and Theorem 3.4.4)

1. Appendix B reviews the basics of linear programming.

tells us that U_1^* holds constant in all equilibria and that it is the same as the value that player 1 achieves under a minmax strategy by player 2. Using this result, we can construct the linear program that follows.

$$\begin{aligned}
&\text{minimize} && U_1^* && && (4.1)\\
&\text{subject to} && \sum_{k \in A_2} u_1\left(a_1^j, a_2^k\right) \cdot s_2^k \leq U_1^* && \forall j \in A_1 && (4.2)\\
& && \sum_{k \in A_2} s_2^k = 1 && && (4.3)\\
& && s_2^k \geq 0 && \forall k \in A_2 && (4.4)
\end{aligned}$$

Note first of all that the utility terms $u_1(\cdot)$ are constants in the linear program, while the mixed strategy terms $s_2^{\cdot}$ and U_1^* are variables. Let us start by looking at constraint (4.2). This states that for every pure strategy j of player 1, his expected utility for playing any action $j \in A_1$ given player 2's mixed strategy s_2 is at most U_1^*. Those pure strategies for which the expected utility is exactly U_1^* will be in player 1's best response set, while those pure strategies leading to lower expected utility will not. Of course, as mentioned earlier U_1^* is a variable; the linear program will choose player 2's mixed strategy in order to minimize U_1^* subject to the constraint just discussed. Thus, lines (4.1) and (4.2) state that player 2 plays the mixed strategy that minimizes the utility player 1 can gain by playing his best response. This is almost exactly what we want. All that is left is to ensure that the values of the variables s_2^k are consistent with their interpretation as probabilities. Thus, the linear program also expresses the constraints that these variables must sum to one (4.3) and must each be nonnegative (4.4).

This linear program gives us player 2's mixed strategy in equilibrium. In the same fashion, we can construct a linear program to give us player 1's mixed strategies. This program reverses the roles of player 1 and player 2 in the constraints; the objective is to *maximize* U_1^*, as player 1 wants to maximize his own payoffs. This corresponds to the *dual* of player 2's program.

$$\begin{aligned}
&\text{maximize} && U_1^* && && (4.5)\\
&\text{subject to} && \sum_{j \in A_1} u_1(a_1^j, a_2^k) \cdot s_1^j \geq U_1^* && \forall k \in A_2 && (4.6)\\
& && \sum_{j \in A_1} s_1^j = 1 && && (4.7)\\
& && s_1^j \geq 0 && \forall j \in A_1 && (4.8)
\end{aligned}$$

Finally, we give a formulation equivalent to our first linear program from Equations (4.1)–(4.4), which will be useful in the next section. This program works by introducing *slack variables* r_1^j for every $j \in A_1$ and then replacing the

slack variable

inequality constraints with equality constraints. This LP formulation follows.

$$\text{minimize} \quad U_1^* \tag{4.9}$$

$$\text{subject to} \quad \sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k + r_1^j = U_1^* \qquad \forall j \in A_1 \tag{4.10}$$

$$\sum_{k \in A_2} s_2^k = 1 \tag{4.11}$$

$$s_2^k \geq 0 \qquad \forall k \in A_2 \tag{4.12}$$

$$r_1^j \geq 0 \qquad \forall j \in A_1 \tag{4.13}$$

Comparing the LP formulation given in Equations (4.9)–(4.12) with our first formulation given in Equations (4.1)–(4.4), observe that constraint (4.2) changed to constraint (4.10) and that a new constraint (4.13) was introduced. To see why the two formulations are equivalent, note that since constraint (4.13) requires only that each slack variable must be positive, the requirement of equality in constraint (4.10) is equivalent to the inequality in constraint (4.2).

4.2 Computing Nash equilibria of two-player, general-sum games

Unfortunately, the problem of finding a Nash equilibrium of a two-player, general-sum game cannot be formulated as a linear program. Essentially, this is because the two players' interests are no longer diametrically opposed. Thus, we cannot state our problem as an optimization problem: one player is not trying to minimize the other's utility.

4.2.1 *Complexity of computing a sample Nash equilibrium*

The issue of characterizing the complexity of computing a sample Nash equilibrium is tricky. No known reduction exists from our problem to a decision problem that is NP-complete, nor has our problem been shown to be easier. An intuitive stumbling block is that *every* game has at least one Nash equilibrium, whereas known NP-complete problems are expressible in terms of decision problems that do not always have solutions.

Current knowledge about the complexity of computing a sample Nash equilibrium thus relies on another, less familiar complexity class that describes the problem of finding a solution which always exists. This class is called PPAD, which stands for "polynomial parity argument, directed version." To describe this class we must first define a family of directed graphs which we will denote $\mathcal{G}(n)$. Let each graph in this family be defined on a set N of 2^n nodes. Although each graph in $\mathcal{G}(n)$ thus contains a number of nodes that is exponential in n, we want to restrict our attention to graphs that can be described in polynomial space. There is no need to encode the set of nodes explicitly; we encode the set of edges in a given graph as follows. Let $Parent : N \mapsto N$ and $Child : N \mapsto N$

PPAD

be two functions that can be encoded as arithmetic circuits with sizes polynomial in n.[2] Let there be one graph $G \in \mathcal{G}(n)$ for every such pair of *Parent* and *Child* functions, as long as G satisfies one additional restriction that is described later. Given such a graph G, an edge exists from a node j to a node k iff $Parent(k) = j$ and $Child(j) = k$. Thus, each node has either zero parents or one parent and either zero children or one child. The additional restriction is that there must exist one distinguished node $0 \in N$ with exactly zero parents.

The aforementioned constraints on the in- and out-degrees of the nodes in graphs $G \in \mathcal{G}(n)$ imply that every node is either part of a cycle or part of a path from a source (a parentless node) to a sink (a childless node). The computational task of problems in the class PPAD is finding either a sink or a source other than 0 for a given graph $G \in \mathcal{G}(n)$. Such a solution always exists: because the node 0 is a source, there must be some sink which is either a descendent of 0 or 0 itself.

We can now state the main complexity result.

Theorem 4.2.1 *The problem of finding a sample Nash equilibrium of a general-sum finite game with two or more players is PPAD-complete.*

Of course, this proof is achieved by showing that the problem is in PPAD and that any other problem in PPAD can be reduced to it in polynomial time. To show that the problem is in PPAD, a reduction is given, which expresses the problem of finding a Nash equilibrium as the problem of finding source or sink nodes in a graph as described earlier. This reduction proceeds quite directly from the proof that every game has a Nash equilibrium that appeals to Sperner's lemma. The harder part is the other half of the puzzle: showing that Nash equilibrium computation is PPAD-hard, or in other words that every problem in PPAD can be reduced to finding a Nash equilibrium of some game with size polynomial in the size of the original problem. This result, obtained in 2005, is a culmination of a series of intermediate results obtained over more than a decade. The initial results relied in part on the concept of *graphical games* (see Section 6.5.2) which, in equilibrium, simulate the behavior of the arithmetic circuits *Parent* and *Child* used in the definition of PPAD. More details are given in the notes at the end of the chapter.

graphical game

What are the practical implications of the result that the problem of finding a sample Nash equilibrium is PPAD-complete? As is the case with other complexity classes such as NP, it is not known whether or not P = PPAD. However, it is generally believed (e.g., due to oracle arguments) that the two classes are not equivalent. Thus, the common belief is that in the worst case, computing a sample Nash equilibrium will take time that is exponential in the size of the game. We do know for sure that finding a Nash equilibrium of a two-player game is no easier than finding an equilibrium of an n-player game—a result that may be surprising, given that in practice different algorithms are used for the two-player case than for the n-player case—and that finding a Nash equilibrium is no easier than finding an arbitrary Brouwer fixed point.

2. We warn the reader that some technical details are glossed over here.

4.2.2 *An LCP formulation and the Lemke–Howson algorithm*

Lemke–Howson algorithm

We now turn to algorithms for computing sample Nash equilibria, notwithstanding the discouraging computational complexity of this problem. We start with the *Lemke–Howson algorithm*, for two reasons. First, it is the best known algorithm for the two-player, general-sum case (however, it must be said, not the fastest algorithm, experimentally speaking). Second, it provides insight into the structure of Nash equilibria, and indeed constitutes an independent, constructive proof of Nash's theorem (Theorem 3.3.22).

The LCP formulation

linear complementarity problem (LCP)

Unlike in the special zero-sum case, the problem of finding a sample Nash equilibrium cannot be formulated as a linear program. However, the problem of finding a Nash equilibrium of a two-player, general-sum game can be formulated as a *linear complementarity problem* (LCP). In this section we show how to construct this formulation by starting with the slack variable formulation given in Equations (4.9)–(4.12). After giving the formulation, we present the Lemke–Howson algorithm, which can be used to solve this LCP.

feasibility program

As it turns out, our LCP will have no objective function at all, and is thus a constraint satisfaction problem, or a *feasibility program*, rather than an optimization problem. Also, we can no longer determine one player's equilibrium strategy by only considering the other player's payoff; instead, we will need to discuss both players explicitly. The LCP for computing the Nash equilibrium of a general-sum two-player game follows.

$$\sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k + r_1^j = U_1^* \qquad \forall j \in A_1 \tag{4.14}$$

$$\sum_{j \in A_1} u_2(a_1^j, a_2^k) \cdot s_1^j + r_2^k = U_2^* \qquad \forall k \in A_2 \tag{4.15}$$

$$\sum_{j \in A_1} s_1^j = 1, \quad \sum_{k \in A_2} s_2^k = 1 \tag{4.16}$$

$$s_1^j \geq 0, \quad s_2^k \geq 0 \qquad \forall j \in A_1, \forall k \in A_2 \tag{4.17}$$

$$r_1^j \geq 0, \quad r_2^k \geq 0 \qquad \forall j \in A_1, \forall k \in A_2 \tag{4.18}$$

$$r_1^j \cdot s_1^j = 0, \quad r_2^k \cdot s_2^k = 0 \qquad \forall j \in A_1, \forall k \in A_2 \tag{4.19}$$

Observe that this formulation bears a strong resemblance to the LP formulation with slack variables given earlier in Equations (4.9)–(4.12). Let us go through the differences. First, as discussed earlier the LCP has no objective function. Second, constraint (4.14) is the same as constraint (4.10) in our LP formulation; however, here we also include constraint (4.15) which constrains player 2's actions in the same way. We also give the standard constraints that probabilities sum to one (4.16), that probabilities are nonnegative (4.17) and that slack variables are nonnegative (4.18), but now state these constraints for both players rather than only for player 1.

0, 1	6, 0
2, 0	5, 2
3, 4	3, 3

Figure 4.1 A game for the exposition of the Lemke–Howson algorithm.

If we included only constraints (4.14)–(4.18)), we would still have a linear program. However, we would also have a flaw in our formulation: the variables U_1^* and U_2^* would be insufficiently constrained. We want these values to express the expected utility that each player would achieve by playing his best response to the other player's chosen mixed strategy. However, with the constraints we have described so far, U_1^* and U_2^* would be allowed to take unboundedly large values, because all of these constraints remain satisfied when both U_i^* and r_i^j are increased by the same constant, for any given i and j. We solve this problem by adding the nonlinear constraint (4.19), called the *complementarity condition*. The addition of this constraint means that we no longer have a linear program; instead, we have a linear complementarity problem.

complementarity condition

Why does the complementarity condition fix our problem formulation? This constraint requires that whenever an action is played by a given player with positive probability (i.e., whenever an action is in the support of a given player's mixed strategy) then the corresponding slack variable must be zero. Under this requirement, each slack variable can be viewed as the player's incentive to deviate from the corresponding action. Thus, the complementarity condition captures the fact that, in equilibrium, all strategies that are played with positive probability must yield the same expected payoff, while all strategies that lead to lower expected payoffs are not played. Taking all of our constraints together, we are left with the requirement that each player plays a best response to the other player's mixed strategy: the definition of a Nash equilibrium.

The Lemke–Howson algorithm: a graphical exposition

Lemke–Howson algorithm

The best-known algorithm designed to solve this LCP formulation is the *Lemke–Howson algorithm*. We will explain it initially through a graphical exposition. Consider the game in Figure 4.1. Figure 4.2 shows a graphical representation of the two players' mixed-strategy spaces in this game. Each player's strategy space is shown in a separate graph. Within a graph, each axis corresponds to one of the corresponding player's pure strategies and the region spanned by these axes represents all the mixed strategies (as discussed in Section 3.3.4, with $k + 1$ axes, the region forms a k-dimensional simplex). For example, in the right-hand side of the figure, the two dots show player 2's two pure strategies and the line connecting them (a one-dimensional simplex) represents all his possible mixed strategies.

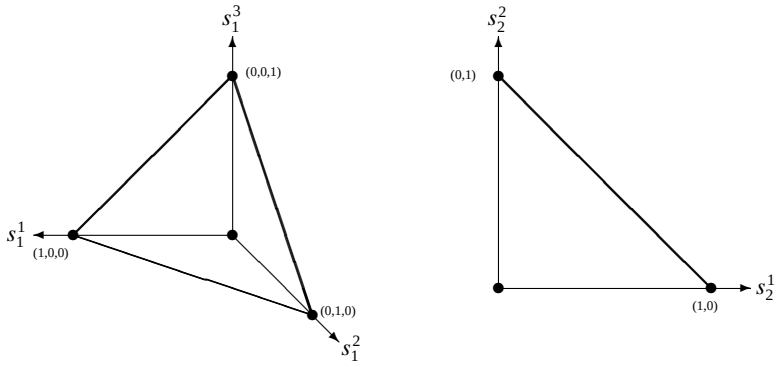

Figure 4.2 Strategy spaces for player 1 (left) and player 2 (right) in the game from Figure 4.1.

Similarly, player 1's three pure strategies are represented by the points (0, 0, 1), (0, 1, 0), and (1, 0, 0), while the set of his mixed strategies (a two-dimensional simplex) is represented by the region bounded by the triangle having these three points as its vertices. (Can you identify the point corresponding to the strategy that randomizes equally among the three pure strategies?)

Our next step in defining the Lemke–Howson algorithm is to define a labeling on the strategies. Every possible mixed strategy s_i is given a set of labels $L(s_i^j) \subseteq A_1 \cup A_2$ drawn from the set of available actions for both players. Denoting a given player as i and the other player as $-i$, mixed strategy s_i for player i is labeled as follows:

- with each of player i's actions a_i^j that is *not* in the support of s_i; and
- with each of player $-i$'s actions a_{-i}^j that *is* a best response by player $-i$ to s_i.

This labeling is useful because a pair of strategies (s_1, s_2) is a Nash equilibrium if and only if it is completely labeled (i.e., $L(s_1) \cup L(s_2) = A_1 \cup A_2$). For a pair to be completely labeled, each action a_i^j must either played by player i with zero probability, or be a best response by player i to the mixed strategy of player $-i$.[3,4]

The requirement that a pair of mixed strategies must be completely labeled can be understood as a restatement of the complementarity condition given in constraint (4.19) in the LCP for computing the Nash equilibrium of a general-sum two-player game, because the slack variable r_i^j is zero exactly when its corresponding action a_i^j is a best response to the mixed strategy s_{-i}.

3. We must introduce a certain caveat here. In general, it is possible that some actions will satisfy both of these conditions and thus belong to both $L(s_1)$ and $L(s_2)$; however, this will not occur when a game is nondegenerate. Full discussion of degenericity lies beyond the scope of the book, but for the record, one definition is as follows: A two-player game is *degenerate* if there exists some mixed strategy for either player such that the number of pure strategy best responses of the other player is greater than the size of the support of the mixed strategy. Here we will assume that the game is nondegenerate.

4. Some readers may be reminded of the labeling of simplex vertices in the proof of Sperner's Lemma in Section 3.3.4. These readers should note that these are rather different kinds of labeling, which should not be confused with each other.

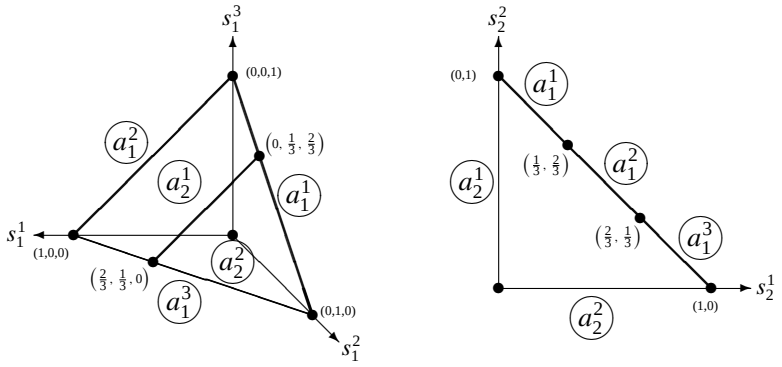

Figure 4.3 Labeled strategy spaces for player 1 (left) and player 2 (right) in the game from Figure 4.1.

It turns out that it is convenient to add one fictitious point in the strategy space of each agent, the origin; that is, $(0, 0, 0)$ for player 1 and $(0, 0)$ for player 2. Thus, we want to be able to consider these points as belonging to the players' strategy spaces. While discussing this algorithm, therefore, we redefine the players' strategy spaces to be the convex hull of their true strategy spaces and the origin of the graph. (This can be understood as replacing the constraint that $\sum_j s_i^j = 1$ with the constraint that $\sum_j s_i^j \leq 1$.) Thus, player 2's strategy space is a triangle with vertices $(0, 0)$, $(1, 0)$, and $(0, 1)$, while player 1's strategy space is a pyramid with vertices $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$.

Returning to our running example, the labeled version of the strategy spaces is given in Figure 4.3. Consider first the right side of Figure 4.3, which describes player 2's strategy space, and examine the two regions labeled with player 2's actions. The line from $(0, 0)$ to $(0, 1)$ is labeled with a_2^1, because none of these mixed strategies assign any probability to playing action a_2^1. In the same way, the line from $(0, 0)$ to $(1, 0)$ is labeled with a_2^2. Now consider the three regions labeled with player 1's actions. Examining the payoff matrix in Figure 4.1, you can verify that, for example, the action a_1^1 is a best response by player 1 to any of the mixed strategies represented by the line from $(0, 1)$ to $(\frac{1}{3}, \frac{2}{3})$. Notice that the point $(\frac{1}{3}, \frac{2}{3})$ is labeled by both a_1^1 and a_1^2, because both of these actions are best responses by player 1 to the mixed strategy $(\frac{1}{3}, \frac{2}{3})$ by player 2.[5]

Similarly, consider now the left side of Figure 4.3, representing player 1's strategy space. There is a region labeled with each action a_1^j of player 1, which is the triangle having a vertex at the origin and running orthogonal to the axis s_1^j. (Can you see why these are the only mixed strategies for player 1 that do not involve the action a_1^j?) The two regions for the labels corresponding to actions of player 2 (a_2^1 and a_2^2) divide the outer triangle. As earlier, note that some mixed strategies are multiply labeled: for example, the point $\left(0, \frac{1}{3}, \frac{2}{3}\right)$ is labeled with a_2^1, a_2^2, and a_1^1.

5. The reader may note a subtlety here. Since we added the point $(0, 0)$ and are considering the entire triangle and not just the line $(1, 0) - (0, 1)$, it might be expected that we would attach best-response labels also to interior points within the triangle. However, it turns out that the Lemke–Howson algorithm traverses only the edges of the polygon containing the simplexes and has no use for interior points, and so we ignore them.

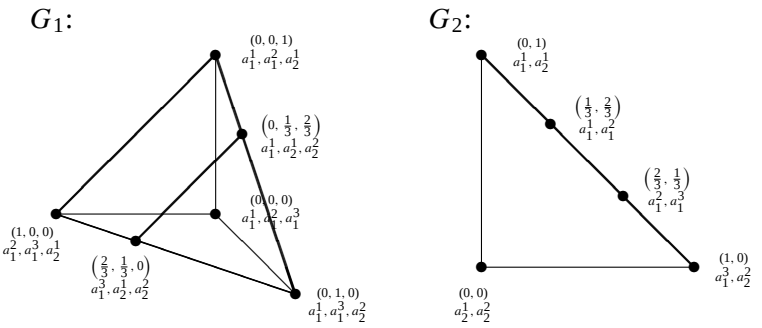

Figure 4.4 Graph of triply labeled strategies for player 1 (left) and doubly labeled strategies for player 2 (right) derived from the game in Figure 4.1.

The Lemke–Howson algorithm can be understood as searching these pairs of labeled spaces for a completely labeled pair of points. Define G_1 and G_2 to be graphs, for players 1 and 2 respectively. The nodes in the graph are fully labeled points in the labeled space, that is, triply labeled points in G_1 and doubly labeled points in G_2. An edge exists between pairs of points that differ in exactly one label. These graphs for our example are shown in Figure 4.4; each node is annotated with the mixed strategy to which it corresponds as well as the actions with which it is labeled.

When the game is nondegenerate, there are no points with more labels than the given player has actions, which implies that a completely labeled pair of strategies must consist of two points that have no labels in common. In our example it is easy to find the three Nash equilibria of the game by inspection: $($(0,0,1),(1,0)$)$, $\left(\left(0, \frac{1}{3}, \frac{2}{3}\right), \left(\frac{2}{3}, \frac{1}{3}\right)\right)$, and $\left(\left(\frac{2}{3}, \frac{1}{3}, 0\right), \left(\frac{1}{3}, \frac{2}{3}\right)\right)$.

The Lemke–Howson algorithm finds an equilibrium by following a path through pairs $(s_1, s_2) \in G_1 \times G_2$ in the cross product of the two graphs. Alternating between the two graphs, each iteration changes one of the two points to a new point that is connected by an edge to the original point. Starting from $(\mathbf{0}, \mathbf{0})$, which is completely labeled, the algorithm picks one of the two graphs and moves from $\mathbf{0}$ in that graph to some adjacent node x. The node x, together with the $\mathbf{0}$ from the other graph, together form an almost completely labeled pair, in that between them they miss exactly one label. The algorithm then moves from the remaining $\mathbf{0}$ to a neighboring node that picks up that missing label, but in the process loses a different label. The process thus proceeds, alternating between the two graphs, until an equilibrium (i.e., a totally labeled pair) is reached.

In our running example, a possible execution of the algorithm starts at $(\mathbf{0}, \mathbf{0})$ and then changes s_1 to $(0, 1, 0)$. Now, our pair $\big((0, 1, 0), (0, 0)\big)$ is a_2^1-almost completely labeled, and the duplicate label is a_2^2. For its next step in G_2 the algorithm moves to $(0, 1)$ because the other possible choice, $(1, 0)$, has the label a_2^2. Returning to G_1 for the next iteration, we move to $\left(\frac{2}{3}, \frac{1}{3}, 0\right)$ because it is the point adjacent to $(0, 1, 0)$ that does not have the duplicate label a_1^1. The final step is to change s_2 to $\left(\frac{1}{3}, \frac{2}{3}\right)$ in order to move away from the label a_2^1. We have now reached the completely labeled pair $\left(\left(\frac{2}{3}, \frac{1}{3}, 0\right), \left(\frac{1}{3}, \frac{2}{3}\right)\right)$, and the algorithm terminates. This execution trace can be summarized by the path $((0, 0, 0), (0, 0)) \rightarrow ((0, 1, 0), (0, 0)) \rightarrow ((0, 1, 0), (0, 1)) \rightarrow ((\frac{2}{3}, \frac{1}{3}, 0), (0, 1)) \rightarrow ((\frac{2}{3}, \frac{1}{3}, 0), (\frac{1}{3}, \frac{2}{3}))$.

The Lemke–Howson algorithm: A deeper look at pivoting

The graphical description of the Lemke–Howson algorithm in the previous section provides good intuition but glosses over elements that only a close look at the algebraic formulation reveals. Specifically, in abstracting away to the graphical exposition we did not specify how to compute the graph nodes from the game description. This is the role of this section. The two sections complement each other: This one provides a clear recipe for implementing the algorithm, but on its own would provide little intuition. The previous section did the opposite.

In fact, we do not compute the nodes in advance at all. Instead, we compute them incrementally along the path being explored. At each step, we find the missing label to be added (called the *entering variable*), add it, find out which label has been lost (it is called the *leaving variable*), and the process repeats until no variable is lost in which case a solution has been obtained. This procedure is called *pivoting*, and also underlies the *simplex algorithm* for solving linear programming problems. The high-level description of the Lemke–Howson algorithm is given in Figure 4.5.

pivot algorithms

simplex algorithm

initialize *the two systems of equations at the origin*
arbitrarily pick *one dependent variable from one of the two systems. This variable* enters *the basis.*
repeat
  identify *one of the previous basis variables which must* leave, *according to the minimum ratio test. The result is a new basis.*
  if *this basis is completely labeled* **then**
    return *the basis* // we have found an equilibrium.
  else
    the variable dual to the variable that last left enters *the basis.*

Figure 4.5 Pseudocode for the Lemke–Howson algorithm.

As can be seen from the pseudocode, identifying the entering variable follows immediately from the current labeling (except in the first step, in which the choice is arbitrary). The only nontrivial step is identifying the leaving variable. We explain it by tracing the operation of the algorithm on our example.

We start with a reformulation of the first two constraints (4.14) and (4.15) from our LCP formulation.[6]

$$\begin{array}{rcccc} r_1 & = & 1 & & -6y'_5 \\ r_2 & = & 1 & -2y'_4 & -5y'_5 \\ r_3 & = & 1 & -3y'_4 & -3y'_5 \end{array} \tag{4.20}$$

$$\begin{array}{rcccc} s_4 & = & 1 & -x'_1 & -4x'_3 \\ s_5 & = & 1 & -2x'_2 & -3x'_3 \end{array} \tag{4.21}$$

6. Beside the minor rearrangement of terms and slight notational change, the reader will note that we have lost the different U values and replaced them by the unit values 1; this turns out to be convenient computationally and does not alter the solutions.

This system admits the trivial solution of assigning 0 to all variables on the right-hand side, which is our fictitious starting point. At this point, r_1, r_2, r_3, s_4, s_5 form the basis of our system of equations, and the other variables (the y's and the x's) are the dependent variables.[7] Note that each basis variable has a dual dependent one; the dual pairs are (r_1, x'_1), (r_2, x'_2), (r_3, x'_3), (s_4, y'_4), and (s_5, y'_5). We will now iteratively remove some of the variables from the basis and replace them with what were previously dependent variables to get a new basis. The rule for which variable enters is simple; initially the choice is arbitrary, and thereafter it is the dual to the variable that previously left. The rule for which variable leaves is more complicated and is called the *minimum ratio test*. When a variable enters, the candidates to leave are all the "clashing variables"; these are all the current basis variables in whose equation the entering variable appears. If there is only one such equation we are done, but otherwise we choose as follows. Each such equation has the form $v = c + qu + T$, where v is the clashing variable, c is a constant (initially they are all 1), u is the entering variable, q is a constant coefficient, and T is a linear combination of variables other than v or u. The clashing variable to leave is the one in whose equation the q/c ratio is smallest.

minimum ratio test

We illustrate the procedure on our example. Let us arbitrarily pick x'_2 as the first entering variable. In this case we see immediately that s_5 must leave, since it is the only clashing variable. (x'_2 does not appear in the equation of any other basis variable.) With x'_2 in the basis the equations much be updated to remove any occurrence of x'_2 on the right-hand side, which in this case is achieved simply by rearranging the terms of the second equation in (4.21). This gives us the following.

$$\begin{array}{lcllll} s_4 &=& 1 & -x'_1 & -4x'_3 & \\ x'_2 &=& \frac{1}{2} & & -\frac{3}{2}x'_3 & -\frac{1}{2}s_5 \end{array} \tag{4.22}$$

The next variable that must enter the basis y'_5, s_5's dual. Now the choice for which variable should leave the basis is less obvious; all three variables r_1, r_2, r_3 clash with y'_5. The variable we choose is r_1, since it has the lowest ratio: $\frac{1}{6}$, versus $\frac{1}{5}$ for r_2 and $\frac{1}{3}$ for r_3. Equation (4.20) is now replaced by the following.

$$\begin{array}{lcllll} y'_5 &=& \frac{1}{6} & & & -\frac{1}{6}r_1 \\ r_2 &=& \frac{1}{6} & -2y'_4 & & +\frac{5}{6}r_1 \\ r_3 &=& \frac{1}{2} & -3y'_4 & & +\frac{1}{2}r_1 \end{array} \tag{4.23}$$

In this case the first equation is rearranged as above, and then, in the second two equations, the occurrences of y'_5 are replaced by $\frac{1}{6} - \frac{1}{6}r_1$.

With r_1 having left x'_1 must enter. This entails that s_4 must leave (in this case again, the only clashing variable). Equation (4.22) now changes as follows.

$$\begin{array}{lcllll} x'_1 &=& 1 & -4x'_3 & -s_4 & \\ x'_2 &=& \frac{1}{2} & -\frac{3}{2}x'_3 & & -\frac{1}{2}s_5 \end{array} \tag{4.24}$$

7. From the definitions of matrix theory, in our particular system the basis variables are independent of each other (i.e., their values can be chosen independently), but together they determine the values of all other variables.

With y'_4 entering, either r_2 or r_3 must leave, and it is r_2 leaves since its ratio of $\frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{12}$ is lower than r_3's ratio of $\frac{\frac{1}{2}}{3} = \frac{1}{6}$. Equation (4.23) changes as follows.

$$\begin{array}{rcccc} y'_5 & = & \frac{1}{6} & -\frac{1}{6}r_1 & \\ y'_4 & = & \frac{1}{12} & +\frac{5}{12}r_1 & -\frac{1}{2}r_2 \\ r_3 & = & \frac{1}{4} & -\frac{3}{4}r_1 & -\frac{3}{2}r_2 \end{array} \tag{4.25}$$

At this point the algorithm terminates since, between them, Equations (4.25) and (4.24) contain all the labels. Renormalizing the vectors x' and y' to be proper probabilities, one gets the solution $((\frac{2}{3}, \frac{1}{3}, 0), (\frac{1}{3}, \frac{2}{3}))$ with payoffs 4 and $\frac{2}{3}$ to the row and column players, respectively.

Properties of the Lemke–Howson algorithm

The Lemke–Howson algorithm has some good properties. First, it is guaranteed to find a sample Nash equilibrium. Indeed, its constructive nature constitutes an alternative proof of the existence of a Nash equilibrium (Theorem 3.3.22). Also, note the following interesting fact: Since the algorithm repeatedly seeks to cover a missing label, after choosing the initial move away from $(\mathbf{0}, \mathbf{0})$, the path through almost completely labeled pairs to an equilibrium is unique. So while the algorithm is nondeterministic, all the nondeterminism is concentrated in its first move. Finally, it can be used to find more than one Nash equilibrium. The reason the algorithm is initialized to start at the origin is that this is the only pair that is known *a priori* to be completely labeled. However, once we have found another completely labeled pair, we can use *it* as the starting point, allowing us to reach additional equilibria. For example, starting at the equilibrium we just found and making an appropriate first choice, we can quickly find another equilibrium by the path $\left(\left(\frac{2}{3}, \frac{1}{3}, 0\right), \left(\frac{1}{3}, \frac{2}{3}\right)\right) \to \left(\left(0, \frac{1}{3}, \frac{2}{3}\right), \left(\frac{1}{3}, \frac{2}{3}\right)\right) \to \left(\left(0, \frac{1}{3}, \frac{2}{3}\right), \left(\frac{2}{3}, \frac{1}{3}\right)\right)$. The remaining equilibrium can be found using the following path from the origin: $((0, 0, 0), (0, 0)) \to ((0, 0, 1), (0, 0)) \to ((0, 0, 1), (1, 0))$.

However, the algorithm is not without its limitations. While we were able to use the algorithm to find all equilibria in our running example, in general we are not guaranteed to be able to do so. As we have seen, the Lemke–Howson algorithm can be thought of as exploring a graph of all completely and almost completely labeled pairs. The bad news is that this graph can be disconnected, and the algorithm is only able to find the equilibria in the connected component that contains the origin (although luckily, there is guaranteed to be at least one such equilibrium). Not only are we unable to guarantee that we will find all equilibria—there is not even an efficient way to determine whether or not all equilibria have been found.

Even with respect to finding a single equilibrium we are not trouble free. First, there is still indeterminacy in the first move, and the algorithm provides no guidance on how to make a good first choice, one that will lead to a relatively short path to the equilibrium, if one exists. And one may not exist—there are cases in which *all* paths are of exponential length (and thus the time complexity of the Lemke–Howson algorithm is provably exponential). Finally, even if one

gives up on worst-case guarantees and hopes for good heuristics, the fact that the algorithm has no objective function means that it provides no obvious guideline to assess how close it is to a solution before actually finding one.

Nevertheless, despite all these limitations, the Lemke–Howson algorithm remains a key element in understanding the algorithmic structure of Nash equilibria in general two-person games.

4.2.3 *Searching the space of supports*

One can identify a spectrum of approaches to the design of algorithms. At one end of the spectrum one can develop deep insight into the structure of the problem, and craft a highly specialized algorithm based on this insight. The Lemke–Howson algorithm lies close to this end of the spectrum. At the other end of the spectrum, one identifies relatively shallow heuristics and hopes that these, coupled with ever-increasing computing power, will do the job. Of course, in order to be effective, even these heuristics must embody some insight into the problem. However, this insight tends to be limited and local, yielding rules of thumb that aid in guiding the search through the space of possible solutions, but that do not directly yield a solution. One of the lessons from computer science is that sometimes heuristic approaches can outperform more sophisticated algorithms in practice. In this section we discuss such a heuristic algorithm.

The basic idea behind the algorithm is straightforward. We first note that while the general problem of computing a Nash equilibrium (NE) is a complementarity problem, computing whether there exists a NE with a *particular support*[8] for each player is a relatively simple feasibility program. So the problem is reduced to searching the space of supports. Of course the size of this space is exponential in the number of actions, and this is where the heuristics come in.

We start with the feasibility program. Given a support profile $\sigma = (\sigma_1, \sigma_2)$ as input (where each $\sigma_i \subseteq A_i$), feasibility program TGS (for "test given supports") finds a NE p consistent with σ or proves that no such strategy profile exists. In this program, v_i corresponds to the expected utility of player i in an equilibrium, and the subscript $-i$ indicates the player other than i as usual. The complete program follows.

$$\sum_{a_{-i} \in \sigma_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) = v_i \qquad \forall i \in \{1,2\}, a_i \in \sigma_i \tag{4.26}$$

$$\sum_{a_{-i} \in \sigma_{-i}} p(a_{-i}) u_i(a_i, a_{-i}) \leq v_i \qquad \forall i \in \{1,2\}, a_i \notin \sigma_i \tag{4.27}$$

$$p_i(a_i) \geq 0 \qquad \forall i \in \{1,2\}, a_i \in \sigma_i \tag{4.28}$$

$$p_i(a_i) = 0 \qquad \forall i \in \{1,2\}, a_i \notin \sigma_i \tag{4.29}$$

$$\sum_{a_i \in \sigma_i} p_i(a_i) = 1 \qquad \forall i \in \{1,2\} \tag{4.30}$$

8. Recall that the support specifies the pure strategies played with nonzero probability (see Definition 3.2.6).

Constraints (4.26) and (4.27) require that each player must be indifferent between all actions within his support and must not strictly prefer an action outside of his support. These imply that neither player can deviate to a pure strategy that improves his expected utility, which is exactly the condition for the strategy profile to be a NE. Constraints (4.28) and (4.29) ensure that each S_i can be interpreted as the support of player i's mixed strategy: the pure strategies in S_i must be played with zero or positive probability, and the pure strategies not in S_i must be played with zero probability.[9] Finally, constraint (4.30) ensures that each p_i can be interpreted as a probability distribution. A solution will be returned only when there exists an equilibrium with support S (subject to the caveat in footnote 9).

With this feasibility program in our arsenal, we can proceed to search the space of supports. There are three keys to the efficiency of the following algorithm, called SEM (for *support-enumeration method*). The first two are the factors used to order the search space. Specifically, SEM considers every possible support size profile separately, favoring support sizes that are balanced and small. The third key to SEM is that it separately instantiates each player's support, making use of what we will call *conditional strict dominance* to prune the search space.

support-enumeration method

conditional strict dominance

Definition 4.2.2 (Conditionally strictly dominated action) *An action $a_i \in A_i$ is* conditionally strictly dominated, *given a profile of sets of available actions $R_{-i} \subseteq A_{-i}$ for the remaining agents, if the following condition holds: $\exists a'_i \in A_i \ \forall a_{-i} \in R_{-i} : \ u_i(a_i, a_{-i}) < u_i(a'_i, a_{-i})$.*

Observe that this definition is strict because, in a Nash equilibrium, no action that is played with positive probability can be conditionally dominated given the actions in the support of the opponents' strategies. The problem of checking whether an action is conditionally strictly dominated is equivalent to the problem of checking whether the action is strictly dominated by a pure strategy in a reduced version of the original game. As we show in Section 4.5.1, this problem can be solved in time linear in the size of the game.

The preference for small support sizes amplifies the advantages of checking for conditional dominance. For example, after instantiating a support of size two for the first player, it will often be the case that many of the second player's actions are pruned, because only two inequalities must hold for one action to conditionally dominate another.

Pseudocode for SEM is given in Figure 4.6.

Note that SEM is complete, because it considers all support size profiles and because it prunes only those actions that are *strictly* dominated. As mentioned earlier, the number of supports is exponential in the number of actions and hence this algorithm has an exponential worst-case running time.

9. Note that constraint (4.28) allows an action $a_i \in S_i$ to be played with zero probability, and so the feasibility program may sometimes find a solution even when some S_i includes actions that are not in the support. However, player i must still be indifferent between action a_i and each other action $a'_i \in S_i$. Thus, simply substituting in $S_i = A_i$ would not necessarily yield a Nash equilibrium as a solution.

forall *support size profiles* $x = (x_1, x_2)$*, sorted in increasing order of, first,* $|x_1 - x_2|$ *and, second,* $(x_1 + x_2)$ **do**
  forall $\sigma_1 \subseteq A_1$ *s.t.* $|\sigma_1| = x_1$ **do**
    $A_2' \leftarrow \{a_2 \in A_2$ not conditionally dominated, given $\sigma_1\}$
    if $\nexists a_1 \in \sigma_1$ *conditionally dominated, given* A_2' **then**
      forall $\sigma_2 \subseteq A_2'$ *s.t.* $|\sigma_2| = x_2$ **do**
        if $\nexists a_1 \in \sigma_1$ *conditionally dominated, given* σ_2 **and** *TGS is satisfiable for* $\sigma = (\sigma_1, \sigma_2)$ **then**
          return *the solution found; it is a NE*

Figure 4.6 The SEM algorithm

Of course, any enumeration order would yield a solution; the particular ordering here has simply been shown to yield solutions quickly in practice. In fact, extensive testing on a wide variety of games encountered throughout the literature has shown SEM to perform *better* than the more sophisticated algorithms. Of course, this result tells us as much about the games in the literature (e.g., they tend to have small-support equilibria) as it tells us about the algorithms.

4.2.4 *Beyond sample equilibrium computation*

In this section we consider two problems related to the computation of Nash equilibria in two-player, general-sum games that go beyond simply identifying a sample equilibrium.

First, instead of just searching for a sample equilibrium, we might want to find an equilibrium with a specific property. Listed below are several different questions we could ask about the existence of such an equilibrium.

1. **(Uniqueness)** Given a game G, does there exist a unique equilibrium in G?
2. **(Pareto optimality)** Given a game G, does there exist a strictly Pareto efficient equilibrium in G?
3. **(Guaranteed payoff)** Given a game G and a value v, does there exist an equilibrium in G in which some player i obtains an expected payoff of at least v?
4. **(Guaranteed social welfare)** Given a game G, does there exist an equilibrium in which the sum of agents' utilities is at least k?
5. **(Action inclusion)** Given a game G and an action $a_i \in A_i$ for some player $i \in N$, does there exist an equilibrium of G in which player i plays action a_i with strictly positive probability?
6. **(Action exclusion)** Given a game G and an action $a_i \in A_i$ for some player $i \in N$, does there exist an equilibrium of G in which player i plays action a_i with zero probability?

The answers to these questions are more useful that they might appear at first glance. For example, the ability to answer the *guaranteed payoff* question

in polynomial time could be used to find, in polynomial time, the maximum expected payoff that can be guaranteed in a Nash equilibrium. Unfortunately, all of these questions are hard in the worst case.

Theorem 4.2.3 *The following problems are NP-hard when applied to Nash equilibria:* uniqueness, Pareto optimality, guaranteed payoff, guaranteed social welfare, action inclusion, *and* action exclusion.

This result holds even for two-player games. Further, it is possible to show that the *guaranteed payoff* and *guaranteed social welfare* properties cannot even be approximated to any constant factor by a polynomial-time algorithm.

A second problem is to determine *all* equilibria of a game.

Theorem 4.2.4 *Computing all of the equilibria of a two-player, general-sum game requires worst-case time that is exponential in the number of actions for each player.*

This result follows straightforwardly from the observation that a game with k actions can have $2^k - 1$ Nash equilibria, even if the game is nondegenerate (when the game is degenerate, it can have an infinite number of equilibria). Consider a two-player Coordination game in which both players have k actions and a utility function given by the identity matrix possesses $2^k - 1$ Nash equilibria: one for each nonempty subset of the k actions. The equilibrium for each subset is for both players to randomize uniformly over each action in the subset. Any algorithm that finds all of these equilibria must have a running time that is at least exponential in k.

4.3 Computing Nash equilibria of *n*-player, general-sum games

nonlinear complementarity problem

For n-player games where $n \geq 3$, the problem of finding a Nash equilibrium can no longer be represented even as an LCP. While it does allow a formulation as a *nonlinear complementarity problem*, such problems are often hopelessly impractical to solve exactly. Unlike the two-player case, therefore, it is unclear how to best formulate the problem as input to an algorithm. In this section we discuss three possibilities.

Instead of solving the nonlinear complementarity problem exactly, there has been some success approximating the solution using a *sequence of linear complementarity problems (SLCP)*. Each LCP is an approximation of the problem, and its solution is used to create the next approximation in the sequence. This method can be thought of as a generalization to *Newton's method* of approximating the local maximum of a quadratic equation. Although this method is not globally convergent, in practice it is often possible to try a number of different starting points because of its relative speed.

Another approach is to formulate the problem as a minimum of a function. First, we need to define some more notation. Starting from a strategy profile s, let $c_i^j(s)$ be the change in utility to player i if he switches to playing action a_i^j as

a pure strategy. Then, define $d_i^j(s)$ as $c_i^j(s)$ bounded from below by zero.

$$c_i^j(s) = u_i(a_i^j, s_{-i}) - u_i(s)$$
$$d_i^j(s) = \max(c_i^j(s), 0)$$

Note that $d_i^j(s)$ is positive if and only if player i has an incentive to deviate to action a_i^j. Thus, strategy profile s is a Nash equilibrium if and only if $d_i^j(s) = 0$ for all players i, and all actions j for each player.

We capture this property in the objective function given in Equation (4.31); we will refer to this function as $f(s)$.

$$\text{minimize} \quad f(s) = \sum_{i \in N} \sum_{j \in A_i} \left(d_i^j(s) \right)^2 \tag{4.31}$$

$$\text{subject to} \quad \sum_{j \in A_i} s_i^j = 1 \qquad \forall i \in N \tag{4.32}$$

$$s_i^j \geq 0 \qquad \forall i \in N, \forall j \in A_i \tag{4.33}$$

This function has one or more global minima at 0, and the set of all s such that $f(s) = 0$ is exactly the set of Nash equilibria. Of course, this property holds even if we did not square each $d_i^j(s)$, but doing so makes the function differentiable everywhere. The constraints on the function are the obvious ones: each player's distribution over actions must sum to one, and all probabilities must be nonnegative. The advantage of this method is its flexibility. We can now apply any method for constrained optimization.

If we instead want to use an unconstrained optimization method, we can roll the constraints into the objective function (which we now call $g(s)$) in such a way that we still have a differentiable function that is zero if and only if s is a Nash equilibrium. This optimization problem follows.

$$\text{minimize} \quad \sum_{i \in N} \sum_{j \in A_i} \left(d_i^j(s) \right)^2 + \sum_{i \in N} \left(1 - \sum_{j \in A_i} s_i^j \right)^2 + \sum_{i \in N} \sum_{j \in A_i} \left(\min(s_i^j, 0) \right)^2$$

Observe that the first term in $g(s)$ is just $f(s)$ from Equation (4.31). The second and third terms in $g(s)$ enforce the constraints given in Equations (4.32) and (4.33) respectively.

A disadvantage in the formulations given in both Equations (4.31)–(4.33) and Equation (4.3) is that both optimization problems have local minima which do not correspond to Nash equilibria. Thus, global convergence is an issue. For example, considering the commonly-used optimization methods hill-climbing and simulated annealing, the former get stuck in local minima while the latter often converge globally only for parameter settings that yield an impractically long running time.

simplicial subdivision

When global convergence is required, a common choice is to turn to the class of *simplicial subdivision algorithms*. Before describing these algorithms we will revisit some properties of the Nash equilibrium. Recall from the Nash existence theorem (Theorem 3.3.22) that Nash equilibria are fixed points of

the best response function, f. (As defined previously, given a strategy profile $s = (s_1, s_2, \ldots, s_n)$, $f(s)$ consists of all strategy profiles $(s_1', s_2', \ldots, s_n')$ such that s_i' is a best response by player i to s_{-i}.) Since the space of mixed-strategy profiles can be viewed as a product of simplexes—a so-called *simplotope*—f is a function mapping from a simplotope to a set of simplotopes.

simplotope

Scarf's algorithm is a simplicial subdivision method for finding the fixed point of any function on a simplex or simplotope. It divides the simplotope into small regions and then searches over the regions. Unfortunately, such a search is approximate, since a continuous space is approximated by a mesh of small regions. The quality of the approximation can be controlled by refining the meshes into smaller and smaller subdivisions. One way to do this is by restarting the algorithm with a finer mesh after an initial solution has been found. Alternately, a *homotopy method* can be used. In this approach, a new variable is added that represents the fidelity of the approximation, and the variable's value is gradually adjusted until the algorithm converges.

homotopy method

An alternative approach, due to Govindan and Wilson, uses a homotopy method in a different way. (This homotopy method actually turns out to be an n-player extension of the Lemke–Howson algorithm, although this correspondence is not obvious.) Instead of varying between coarse and fine approximations, the new added variable interpolates between the given game and an easy-to-solve game. That is, we define a set of games indexed by a scalar $\lambda \in [0, 1]$ such that when $\lambda = 0$, we have our original game, and when $\lambda = 1$, we have a very simple game. (One way to do this is to change the original game by adding a "bonus" λk to each player's payoff in one outcome $a = (a_1, \ldots, a_n)$. Consider a choice of k big enough that for each player i, playing a_i is a strictly dominant strategy. Then, when $\lambda = 1$, a will be a (unique) Nash equilibrium, and when $\lambda = 0$, we will have our original game.) We begin with an equilibrium to the simple game and $\lambda = 1$ and let both the equilibrium to the game and the index vary in a continuous fashion to trace the path of game-equilibrium pairs. Along this path λ may both decrease and increase; however, if the path is followed correctly, it will necessarily pass through a point where $\lambda = 0$. This point's corresponding equilibrium is a sample Nash equilibrium of the original game.

Finally, it is possible to generalize the SEM algorithm to the n-player case. Unfortunately, the feasibility program becomes nonlinear, as follows. We call this feasibility program TGS-n.

$$\sum_{a_{-i} \in \sigma_{-i}} \left(\prod_{j \neq i} p_j(a_j) \right) u_i(a_i, a_{-i}) = v_i \qquad \forall i \in N, a_i \in \sigma_i \tag{4.34}$$

$$\sum_{a_{-i} \in \sigma_{-i}} \left(\prod_{j \neq i} p_j(a_j) \right) u_i(a_i, a_{-i}) \leq v_i \qquad \forall i \in N, a_i \notin \sigma_i \tag{4.35}$$

$$p_i(a_i) \geq 0 \qquad \forall i \in N, a_i \in \sigma_i \tag{4.36}$$

$$p_i(a_i) = 0 \qquad \forall i \in N, a_i \notin \sigma_i \tag{4.37}$$

$$\sum_{a_i \in \sigma_i} p_i(a_i) = 1 \qquad \forall i \in N \tag{4.38}$$

The expression $p(a_{-i})$ from constraints (4.26) and (4.27) is no longer a single variable, but must now be written as $\prod_{j \neq i} p_j(a_j)$ in constraints (4.34) and (4.35). The resulting feasibility problem can be solved using standard numerical techniques for nonlinear optimization. As with two-player games, in principle any enumeration method would work; the question is which search heuristic works the fastest. It turns out that a minor modification of the SEM heuristic described in Figure 4.6 is effective for the general case as well: one simply reverses the lexicographic ordering between size and balance of supports (SEM first sorts them by size, and then by a measure of balance; in the n-player case we reverse the ordering). The resulting heuristic algorithm performs very well in practice, and better than the algorithms discussed earlier. We should note that while the ordering between balance and size becomes extremely important to the efficiency of the algorithm as n increases, this reverse ordering does not perform substantially worse than SEM in the two-player case, because the smallest of the balanced support size profiles still appears very early in the ordering.

4.4 Computing maxmin and minmax strategies for two-player, general-sum games

Recall from Section 3.4.1 that in a two-player, general-sum game a maxmin strategy for player i is a strategy that maximizes his worst-case payoff, presuming that the other player j follows the strategy that will cause the greatest harm to i. A minmax strategy for j against i is such a maximum-harm strategy. Maxmin and minmax strategies can be computed in polynomial time because they correspond to Nash equilibrium strategies in related zero-sum games.

Let G be an arbitrary two-player game $G = (\{1, 2\}, A_1 \times A_2, (u_1, u_2))$. Let us consider how to compute a maxmin strategy for player 1. It will be useful to define the zero-sum game $G' = (\{1, 2\}, A_1 \times A_2, (u_1, -u_1))$, in which player 1's utility function is unchanged and player 2's utility is the negative of player 1's. By the minmax theorem (Theorem 3.4.4), since G' is zero sum every strategy for player 1 which is part of a Nash equilibrium strategy profile for G' is a maxmin strategy for player 1 in G'. Notice that by definition, player 1's maxmin strategy is independent of player 2's utility function. Thus, player 1's maxmin strategy is the same in G and in G'. Our problem of finding a maxmin strategy in G thus reduces to finding a Nash equilibrium of G', a two-player, zero-sum game. We can thus solve the problem by applying the techniques given earlier in Section 4.1.

The computation of minmax strategies follows the same pattern. We can again use the minmax theorem to argue that player 2's Nash equilibrium strategy in G' is a minmax strategy for him against player 1 in G. (If we wanted to compute player 1's minmax strategy, we would have to construct another game G'' where player 1's payoff is $-u_2$, the negative of player 2's payoff in G.) Thus, both maxmin and minmax strategies can be computed efficiently for two-player games.

4.5 Identifying dominated strategies

Recall that one strategy dominates another when the first strategy is always at least as good as the second, regardless of the other players' actions. (Section 3.4.3 gave the formal definitions.) In this section we discuss some computational uses for identifying dominated strategies, and consider the computational complexity of this process.

As discussed earlier, iterated removal of strictly dominated strategies is conceptually straightforward: the same set of strategies will be identified regardless of the elimination order, and all Nash equilibria of the original game will be contained in this set. Thus, this method can be used to narrow down the set of strategies to consider before attempting to identify a sample Nash equilibrium. In the worst case this procedure will have no effect—many games have *no* dominated strategies. In practice, however, it can make a big difference to iteratively remove dominated strategies before attempting to compute an equilibrium.

Things are a bit trickier with the iterated removal of *weakly* or *very weakly* dominated strategies. In this case the elimination order does make a difference: the set of strategies that survive iterated removal can differ depending on the order in which dominated strategies are removed. As a consequence, removing weakly or very weakly dominated strategies *can* eliminate some equilibria of the original game. There is still a computational benefit to this technique, however. Since no new equilibria are ever created by this elimination (and since every game has at least one equilibrium), at least one of the original equilibria always survives. This is enough if all we want to do is to identify a sample Nash equilibrium. Furthermore, iterative removal of weakly or very weakly dominated strategies can eliminate a larger set of strategies than iterative removal of strictly dominated strategies and so will often produce a smaller game.

What is the complexity of determining whether a given strategy can be removed? This depends on whether we are interested in checking the strategy for domination by a pure or mixed strategies, whether we are interested in strict, weak or very weak domination, and whether we are interested only in domination or in survival under iterated removal of dominated strategies.

4.5.1 *Domination by a pure strategy*

The simplest case is checking whether a (not necessarily pure) strategy s_i for player i is (strictly; weakly; very weakly) dominated by any pure strategy for i. For concreteness, let us consider the case of strict dominance. To solve the problem we must check every pure strategy a_i for player i and every pure-strategy profile for the other players to determine whether there exists some a_i for which it is never weakly better for i to play s_i instead of a_i. If so, s_i is strictly dominated. An algorithm for this case is given in Figure 4.7.

Observe that this algorithm works because we do not need to check every *mixed*-strategy profile of the other players, even though the definition of dominance refers to such strategies. Why can we get away with this? If it is the case (as the inner loop of our algorithm attempts to prove) that for every

```
forall pure strategies a_i ∈ A_i for player i where a_i ≠ s_i do
    dom ← true
    forall pure-strategy profiles a_{-i} ∈ A_{-i} for the players other than i do
        if u_i(s_i, a_{-i}) ≥ u_i(a_i, a_{-i}) then
            dom ← false
            break
    if dom = true then
        return true
return false
```

Figure 4.7 Algorithm for determining whether s_i is strictly dominated by any pure strategy.

pure-strategy profile $a_{-i} \in A_{-i}$, $u_i(s_i, a_{-i}) < u_i(a_i, a_{-i})$, then there cannot exist any mixed-strategy profile $s_{-i} \in S_{-i}$ for which $u_i(s_i, s_{-i}) \geq u_i(a_i, s_{-i})$. This holds because of the linearity of expectation.

The case of very weak dominance can be tested using essentially the same algorithm as in Figure 4.7, except that we must test the condition $u_i(s_i, s_{-i}) > u_i(s_i', s_{-i})$. For weak dominance we need to do a bit more book-keeping: we can test the same condition as for very weak dominance, but we must also set $dom \leftarrow false$ if there is not at least one s_{-i} for which $u_i(s_i, s_{-i}) < u_i(s_i', s_{-i})$. For all of the definitions of domination, the complexity of the procedure is $O(|A|)$, linear in the size of the normal-form game.

4.5.2 *Domination by a mixed strategy*

Recall that sometimes a strategy is not dominated by any pure strategy, but *is* dominated by some mixed strategy. (We saw an example of this in Figure 3.16.) We cannot use a simple algorithm like the one in Figure 4.7 to test whether a given strategy s_i is dominated by a mixed strategy because these strategies cannot be enumerated. However, it turns out that we can still answer the question in polynomial time by solving a linear program. In this section, we will assume that player i's utilities are strictly positive. This assumption is without loss of generality because if any player i's utilities were negative, we could add a constant to all of i's payoffs without changing the game (see Section 3.1.2).

Each flavor of domination requires a somewhat different linear program. First, let us consider strict domination by a mixed strategy. This would seem to have the following straightforward LP formulation (indeed, a mere feasibility program).

$$\sum_{j \in A_i} p_j u_i(a_j, a_{-i}) > u_i(s_i, a_{-i}) \qquad \forall a_{-i} \in A_{-i} \tag{4.39}$$

$$p_j \geq 0 \qquad \forall j \in A_i \tag{4.40}$$

$$\sum_{j \in A_i} p_j = 1 \tag{4.41}$$

While constraints (4.39)–(4.41) do indeed describe strict domination by a mixed strategy, they do not constitute a linear program. The problem is that the constraints in linear programs must be *weak* inequalities (see Appendix B), and

thus we cannot write constraint (4.39) as we have done here. Instead, we must use the LP that follows.

$$\text{minimize} \quad \sum_{j \in A_i} p_j \tag{4.42}$$

$$\text{subject to} \quad \sum_{j \in A_i} p_j u_i(a_j, a_{-i}) \geq u_i(s_i, a_{-i}) \qquad \forall a_{-i} \in A_{-i} \tag{4.43}$$

$$p_j \geq 0 \qquad \forall j \in A_i \tag{4.44}$$

This linear program simulates the strict inequality of constraint (4.39) through the objective function, as we will describe in a moment. Because no constraints restrict the p_j's from above, this LP will always be feasible. However, in the optimal solution the p_j's may not sum to 1; indeed, their sum can be greater than 1 or less than 1. In the optimal solution the p_j's will be set so that their sum cannot be reduced any further without violating constraint (4.43). Thus for at least some $a_{-i} \in A_{-i}$ we will have $\sum_{j \in A_i} p_j u_i(a_j, a_{-i}) = u_i(s_i, a_{-i})$. A strictly dominating mixed strategy therefore exists if and only if the optimal solution to the LP has objective function value strictly less than 1. In this case, we can add a positive amount to each p_j in order to cause constraint (4.43) to hold in its strict version everywhere while achieving the condition $\sum_j p_j = 1$.

Next, let us consider very weak domination. This flavor of domination does not require any strict inequalities, so things are easy here. Here we *can* construct a feasibility program—nearly identical to our earlier failed attempt from Equations (4.39)–(4.41)—which follows.

$$\sum_{j \in A_i} p_j u_i(a_j, a_{-i}) \geq u_i(s_i, a_{-i}) \qquad \forall a_{-i} \in A_{-i} \tag{4.45}$$

$$p_j \geq 0 \qquad \forall j \in A_i \tag{4.46}$$

$$\sum_{j \in A_i} p_j = 1 \tag{4.47}$$

Finally, let us consider weak domination by a mixed strategy. Again our inability to write a strict inequality will make things more complicated. However, we can derive an LP by adding an objective function to the feasibility program given in Equations (4.45)–(4.47).

$$\text{maximize} \quad \sum_{a_{-i} \in A_{-i}} \left[\left(\sum_{j \in A_i} p_j \cdot u_i(a_j, a_{-i}) \right) - u_i(s_i, a_{-i}) \right] \tag{4.48}$$

$$\text{subject to} \quad \sum_{j \in A_i} p_j u_i(a_j, a_{-i}) \geq u_i(s_i, a_{-i}) \qquad \forall a_{-i} \in A_{-i} \tag{4.49}$$

$$p_j \geq 0 \qquad \forall j \in A_i \tag{4.50}$$

$$\sum_{j \in A_i} p_j = 1 \tag{4.51}$$

Because of constraint (4.49), any feasible solution will have a nonnegative objective value. If the optimal solution has a strictly positive objective, the mixed strategy given by the p_j's achieves strictly positive expected utility for at least one $a_{-i} \in A_{-i}$, meaning that s_i is weakly dominated by this mixed strategy.

As a closing remark, observe that all of our linear programs can be modified to check whether a strategy s_i is strictly dominated by any mixed strategy that only places positive probability on some subset of i's actions $T \subset A_i$. This can be achieved simply by replacing all occurrences of A_i by T in the linear programs given earlier.

4.5.3 *Iterated dominance*

Finally, we consider the iterated removal of dominated strategies. We only consider pure strategies as candidates for removal; indeed, as it turns out, it never helps to remove dominated mixed strategies when performing iterated removal. It *is* important, however, that we consider the possibility that pure strategies may be dominated *by* mixed strategies, as we saw in Section 3.4.3.

For all three flavors of domination, it requires only polynomial time to iteratively remove dominated strategies until the game has been maximally reduced (i.e., no strategy is dominated for any player). A single step of this process consists of checking whether every pure strategy of every player is dominated by any other mixed strategy, which requires us to solve at worst $\sum_{i \in N} |A_i|$ linear programs. Each step removes one pure strategy for one player, so there can be at most $\sum_{i \in N}(|A_i| - 1)$ steps.

However, recall that some forms of dominance can produce different reduced games depending on the order in which dominated strategies are removed. We might therefore want to ask other computational questions, regarding which strategies remain in reduced games. Listed below are some such questions.

1. **(Strategy elimination)** Does there exist some elimination path under which the strategy s_i is eliminated?
2. **(Reduction identity)** Given action subsets $A'_i \subseteq A_i$ for each player i, does there exist a maximally reduced game where each player i has the actions A'_i?
3. **(Reduction size)** Given constants k_i for each player i, does there exist a maximally reduced game where each player i has exactly k_i actions?

It turns out that the complexity of answering these questions depends on the form of domination under consideration.

Theorem 4.5.1 *For iterated strict dominance, the* strategy elimination, reduction identity, uniqueness *and* reduction size *problems are in P. For iterated weak dominance, these problems are NP-complete.*

The first part of this result, considering iterated strict dominance, is straightforward: it follows from the fact that iterated strict dominance always arrives at the same set of strategies regardless of elimination order. The second part is tricker; indeed, our statement of this theorem sweeps under the carpet some subtleties

about whether domination by mixed strategies is considered (it is in some cases, and is not in others) and the minimum number of utility values permitted for each player. For all the details, the reader should consult the papers cited at the end of the chapter.

4.6 Computing correlated equilibria

The final solution concept that we will consider is correlated equilibrium. It turns out that correlated equilibria are (probably) easier to compute than Nash equilibria: a sample correlated equilibrium can be found in polynomial time using a linear programming formulation. It is not hard to see (e.g., from the proof of Theorem 3.4.13) that every game has at least one correlated equilibrium in which the value of the random variable can be interpreted as a recommendation to each agent of what action to play, and in equilibrium the agents all follow these recommendations. Thus, we can find a sample correlated equilibrium if we can find a probability distribution over pure action profiles with the property that each agent would prefer to play the action corresponding to a chosen outcome when told to do so, given that the other agents are doing the same.

As in Section 3.2, let $a \in A$ denote a pure-strategy profile, and let $a_i \in A_i$ denote a pure strategy for player i. The variables in our linear program are $p(a)$, the probability of realizing a given pure-strategy profile a; since there is a variable for every pure-strategy profile there are thus $|A|$ variables. Observe that as above the values $u_i(a)$ are constants. The linear program follows.

$$\sum_{a \in A | a_i \in a} p(a) u_i(a) \geq \sum_{a \in A | a_i \in a} p(a) u_i(a'_i, a_{-i}) \quad \forall i \in N, \forall a_i, a'_i \in A_i \tag{4.52}$$

$$p(a) \geq 0 \qquad \forall a \in A \tag{4.53}$$

$$\sum_{a \in A} p(a) = 1 \tag{4.54}$$

Constraints (4.53) and (4.54) ensure that p is a valid probability distribution. The interesting constraint is (4.52), which expresses the requirement that player i must be (weakly) better off playing action a when he is told to do so than playing any other action a'_i, given that other agents play their prescribed actions. This constraint effectively restates the definition of a correlated equilibrium given in Definition 3.4.12. Note that it can be rewritten as $\sum_{a \in A | a_i \in a} [u_i(a) - u_i(a'_i, a_{-i})] p(a) \geq 0$; in other words, whenever agent i is "recommended" to play action a_i with positive probability, he must get at least as much utility from doing so as he would from playing any other action a'_i.

We can select a desired correlated equilibrium by adding an objective function to the linear program. For example, we can find a correlated equilibrium that maximizes the sum of the agents' expected utilities by adding the objective function

$$\text{maximize:} \quad \sum_{a \in A} p(a) \sum_{i \in N} u_i(a). \tag{4.55}$$

Furthermore, all of the questions discussed in Section 4.2.4 can be answered about correlated equilibria in polynomial time, making them (most likely) fundamentally easier problems.

Theorem 4.6.1 *The following problems are in the complexity class P when applied to correlated equilibria:* uniqueness, Pareto optimal, guaranteed payoff, subset inclusion, *and* subset containment.

Finally, it is worthwhile to consider the reason for the computational difference between correlated equilibria and Nash equilibria. Why can we express the definition of a correlated equilibrium as a linear constraint (4.52), while we cannot do the same with the definition of a Nash equilibrium, even though both definitions are quite similar? The difference is that a correlated equilibrium involves a single randomization over action profiles, while in a Nash equilibrium agents randomize separately. Thus, the (nonlinear) version of constraint (4.52) which would instruct a feasibility program to find a Nash equilibrium would be

$$\sum_{a \in A} u_i(a) \prod_{j \in N} p_j(a_j) \geq \sum_{a \in A} u_i(a'_i, a_{-i}) \prod_{j \in N \setminus \{i\}} p_j(a_j) \quad \forall i \in N, \forall a'_i \in A_i.$$

This constraint now mimics constraint (4.52), directly expressing the definition of Nash equilibrium. It states that each player i attains at least as much expected utility from following his mixed strategy p_i as from any pure strategy deviation a'_i, given the mixed strategies of the other players. However, the constraint is nonlinear because of the product $\prod_{j \in N} p_j(a_j)$.

4.7 History and references

The complexity of finding a sample Nash equilibrium is explored in a series of articles. First came the original definition of the class TFNP [Megiddo and Papadimitriou, 1991], a super-class of PPAD followed by the definition of PPAD by Papadimitriou [1994]. Next, Goldberg and Papadimitriou [2006] showed that finding an equilibrium of a game with any constant number of players is no harder than finding the equilibrium of a four-player game, and Daskalakis et al. [2006b] showed that these computational problems are PPAD-complete. The result was almost immediately tightened to encompass two-player games by Chen and Deng [2006]. The NP-completeness results for Nash equilibria with specific properties are due to Gilboa and Zemel [1989] and Conitzer and Sandholm [2003b]; the inapproximability result appeared in Conitzer [2006].

A general survey of the classical algorithms for computing Nash equilibria in 2-person games is provided in von Stengel [2002]. Another good survey is McKelvey and McLennan [1996]. Some specific references, both to these classical algorithms and to the newer ones discussed in the chapter, are as follows. The Lemke–Howson algorithm Lemke and Howson [1964] can be understood as a a specialization of Lemke's pivoting procedure for solving linear complementarity problems [Lemke, 1978]. The graphical exposition of the Lemke–Howson algorithm appeared first in Shapley [1974], and then in a modified

version in von Stengel [2002]. Our description of the Lemke–Howson algorithm is based on the latter. An example of games for which *all* Lemke–Howson paths are of exponential length appears in Savani and von Stengel [2004]. Scarf's simplicial-subdivision-based algorithm is described in Scarf [1967]. Homotopy-based approximation methods are covered, for example, in García and Zangwill [1981]. Govindan and Wilson's homotopy method was presented in Govindan and Wilson [2003]; its path-following procedure depends on topological results due to Kohlberg and Mertens [1986]. The support-enumeration method for finding a sample Nash equilibrium is described in Porter et al. [2004a]. The complexity of iteratedly eliminating dominated strategies is described in Gilboa et al. [1989] and Conitzer and Sandholm [2005].

GAMBIT

Two online resources are of particular note. *GAMBIT* [McKelvey et al., 2006] (`http://econweb.tamu.edu/gambit`) is a library of game-theoretic algorithms for finite normal-form and extensive-form games. It includes many different algorithms for finding Nash equilibria. In addition to several algorithms that can be used on general sum, n-player games, it includes implementations of algorithms designed for special cases, including two-player games, zero-sum games, and finding all equilibria. Finally, *GAMUT* [Nudelman et al., 2004] (`http://gamut.stanford.edu`) is a suite of game generators designed for testing game-theoretic algorithms.

GAMUT