Department of Computer Science
George Mason University
Technical Report Series

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/
703-993-1530

Algorithms for Leximin-Optimal Fair Policies in Repeated Games *

Gabriel Balan gbalan@cs.gmu.edu
Dana Richards richards@cs.gmu.edu
Sean Luke sean@cs.gmu.edu

Technical Report GMU-CS-TR-2008-1

Abstract

Solutions to non-cooperative multiagent systems often require achieving a joint policy which is as fair to all parties as possible. There are a variety of methods for determining the fairest such joint policy. One approach, *min fairness*, finds the policy which maximizes the minimum average reward given to any agent. We focus on an extension, *leximin fairness*, which breaks ties among candidate policies by choosing the one which maximizes the second-to-minimum average reward, then the third-to-minimum average reward, and so on. This method has a number of advantages over others in the literature, but has so far been little-used because of the computational cost in employing it to find the fairest policy. In this paper we propose a linear programming based algorithm for computing leximin fairness in repeated games which has a polynomial time complexity given certain reasonable assumptions.

1 Introduction

Consider the following resource/task allocation problem. A department of some university offers a number of classes taught by its professors. Each class can be taught by some professor and each professor can teach some class. Teaching a class can be demanding at different levels on different professors, so each valid class assignment is characterized by a vector of payoffs for the professors in the department.

At the beginning of each semester, the department head must assign professors to classes. Choosing a single assignment and using it over and over may result in uneven or inefficient workloads for the parties involved. Instead, we define a solution as a probability distribution over all possible assignments. In a solution, sometimes professors will be assigned light workloads and sometimes heavy ones, but on average the expected utilities are fair and efficient.

The motivation for this paper comes from multiagent systems, which often involve collectively deciding on a joint action which is *fairest* to all involved. This process is complicated by the fact that multiple agents (in our example, the professors) are making decisions (leaving the process vulnerable to miscoordinations), and by possible dependencies between successive joint decisions. As a first step towards analyzing this problem, we have decided to step back from these complicating factors, simplifying the problem to a single decision-making agent (the "department chair") and no dependencies between successive joint decisions. Because we have a single decision-maker, each previously joint decision may be converted to a single "action" the decision-maker may choose to make. In some sense this reduces the degree to which the problem is a multiagent problem — multiple agents are affected by the results but there is only one decision-maker. As it turns out even this simplified version is nontrivial!

The problem may be described as follows. A single decision-making agent must repeatedly select an action from a finite set of possible actions. Every such action will result a set of rewards (outcomes) for some group of individuals: different individuals may be rewarded differently from a given action, but an action will always

*This technical report is the text of an article prepared for a conference, but which was then abandoned after we discovered that the proposed algorithms had been previously discovered [10, 11].

reward a given individual by the same amount each time. Actions early on do not affect the rewards offered by actions later on.

The objective of the decision-making agent is to find a probability distribution over all actions such that, if the next action was selected from this distribution at each time step, for an infinite number of steps, the average rewards (that is, the *utilities*) among the individuals would be as fair as possible, without being wasteful. This probability distribution will be termed a *policy*.

What does it mean for a policy to be "fair"? One might define fairness as minimal variance among the utilities of individuals; or require that all individuals have identical utility. We, like others in the literature, have chosen to define fairness in a maximin sense: a policy is fairest if it is the one which provides the highest utility to its minimum-utility individual. This notion of fairness coincides with the social contracts espoused by the philosopher John Rawls: that the measure of a society is how well its worst-off are treated.

We will call the optimally fair policy, using this metric, to be a MIN-policy. But it is possible, even probable, that there will be many MIN-policy solutions to a given problem. Which should then be chosen? One approach is as follows. From among the set of equivalent MIN policies, we will prefer policies for which the second-to-worst-off individual's utility is maximized. If there is more than one such policy, we then prefer from among them policies for which the third-to-worst individual's utility is maximized, and so on. We call a maximally fair policy using this revised lexicographic-ordering metric a LEXIMIN-policy. This metric is especially attractive because LEXIMIN optimality implies pareto efficiency, thus achieving both our goals (fairness and low wastefulness).

In this paper we will show that discovering optimally LEXIMIN fair policies can be done in polynomial time with the partial help of linear programming. We will begin with a discussion of related work, then a formal description of the problem, followed by a presentation of our LEXIMIN algorithm, including an overview, complexity discussion and proof of correctness.

2 Related Work

Minimum-bound fairness has long been of interest in "one-shot" game algorithms where risk aversion is strong. Some of the most prominent such algorithms, cake-cutting methods, compute a one-shot allocation of an infinitesimally divisible resource (the cake) [13]. There also exist cake-cutting algorithms for one-shot allocation of multiple indivisible resources, both allowing side-payments among the players [13], and disallowing them [2]. These algorithms variously allow for heterogeneous degree and measure of reward among the agents, but guarantee that each of the N agents will feel he has at least $1/N$ utility using his own measure. When all agents in such a cake-cutting method use the same utility measure, many such methods will reduce to *absolute fairness*, a simplification of MIN fairness where every agent receives *exactly* the same fraction of resource.

MIN fairness can be wasteful in the sense that once the worst-off agent is as well-off as possible, the others have no way of signaling preference over outcomes. The refinement of MIN fairness we focus on in this paper, LEXIMIN, has also been proposed in context of one-shot resource allocation, in applications such as splitting a network's capacity among multiple commodity flows [9] or splitting the usage of a satellite [1].

A stronger notion of LEXIMIN, *max-min*, is widely used in internet packet routing literature [12, 18]. For situations where giving absolute priority to the worst-off is too strong, a number of extensions and variations have been proposed. For example, *stratified egalitarianism* uses a "poverty threshold" to prefer outcomes that improve all those under the poverty line, even at the expense of those over the line [8]. Another example is the parameterized family of aggregation functions proposed in [3], which optimizes efficiency at one extreme and LEXIMIN fairness at the other, with a continuum of behaviors in between; these functions are based on the *ordered weighted averaging* framework from [17].

Repeated games have been less-well studied in this context. Our inspiration for this paper is derived from previous literature in achieving fairness through reinforcement learning on repeated games [15, 16]. Here, each player makes an action at each timestep. The joint actions of the players result in different rewards for each player, which the players use in their learning algorithm. Players occasionally meet and consult with one another, trading reward information, and the player doing best backs off to allow other players to achieve higher rewards. Unfortunately these algorithms do not discover optimal solutions, primarily because they make myopic decisions employing the smallest immediate variation in reward among the agents; and because they have limited information about the reward function.[1]

[1] For example, consider a two-player repeated game with three actions per player. The reward matrix looks like this:

		Player 1		
		A	B	C
Player 2	A	(100,0)	(0,0)	(0,0)
	B	(0,0)	(1,1)	(0,0)
	C	(0,0)	(0,0)	(0,100)

Of the three Nash equilibria (down the diagonal), the myopic decision-makers in [15] would settle on (B, B) as it contains the least variance among players' rewards and is the fairest for a one-shot game. But over the long-run, playing (A, A) half the time and (C, C) the other half would result in an average of 50 per player per timestep, compared

We ask: if one knew the entire reward function and permitted a single decision-maker to determine a joint action for the agents each timestep, how difficult would it be to identify LEXIMIN fair solutions? These assumptions move the problem to the very edge of what we would still consider a "multi-agent" problem; but they provide us with a lower bound from which we can then begin adding in multi-agent problem complications in future work.

One such LEXIMIN optimization algorithm was proposed in [1] for the constrained 1-shot resource allocation problem. This algorithm can also solve the continuous problem, but the running time is exponential, while our algorithm achieves a polynomial bound. The difference is that we are able to direct our search process to eliminate any branching. The abstract algorithm for maximin presented in [12] identifies the same theoretical principle we use to avoid branching, but the authors fail to provide any practical clues on how their set algebra operations should be implemented in the multidimensional continuous case.

3 Problem Formulation

Our repeated game is played forever. At each timestep the decision-maker chooses one of m actions, and that action rewards n *players*[2], possibly with a different reward for each player. The game has no state; a given action always rewards its players in the same fashion each time. For a given player P_i and policy f, the utility function $U(P_i, f)$ provides the expected utility that P_i receives from a game in which a decision-maker is following f.

Assume that the policy is to choose actions by drawing them from a probability distribution. We seek to discover one or more such policies such that the game is maximally fair to the players. Our measures for fairness are MIN, where the fairest policy maximizes the utility given to the minimum-utility player; and LEXIMIN, which breaks ties among MIN policies by maximizing the utility given to the second-to-minimum-utility player, then breaks further ties using the third-to-minimum player, and so on.

There are certain important observations to draw from this formulation:

- The sequence of actions is infinite: otherwise the optimal solution might depend on the length of the sequence.
- Since we are concerned with players' utilities rather than rewards, the order of actions is not important. All that matters is the relative proportions of actions that are played over the long-term.

to an average of 1 for (B, B).

[2]We realize that as these agents are not making *plays*, it's odd to call them players, but we do so for consistency with the literature.

4 LEXIMIN fairness algorithm

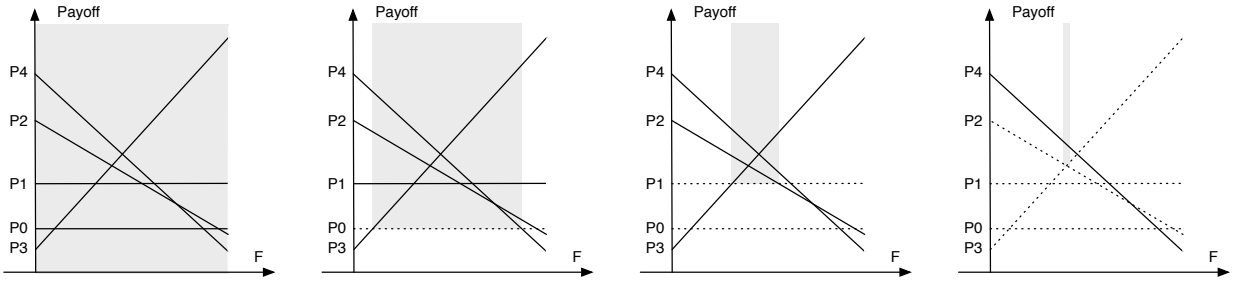
Figure 1 shows a game in which there are only two possible actions the decision-maker may make. This figure plots the utility received by various players as a function of the probability that one of those two actions is made (as opposed to the other action). Thus the horizontal axis represents the policy. These plots are lines for the following reason. Player P_3 (for example) has a high reward whenever one action is played and a low reward whenever the alternative is played. For any probability value between those two extremes (always playing an action; always playing the alternative), the average reward — the utility — is a linear function, namely, the probability-weighted sum of the two rewards.

Broadly, we will find the optimal LEXIMIN fair value(s) of f by determining the poorest player under the optimal policy, then the second poorest player, and so on until all players are "ranked." The gray regions in Figure 1 represent the shrinking of the search space as this process unfolds. In Subfigure 1(a) there are no restrictions on the policy yet. P_0 is ranked as the worst-off and the search space is restricted to the set of policies where P_0 gets its maximal utility (see Subfigure 1(b)). We refer to this space restriction as P_0's domain, or D_0. The next worse-off is P_1, and the search space is again shrunk in Subfigure 1(c). P_2 and P_3 tie for the next two positions and the search space (in this case a single point) as shown in Subfigure 1(d). P_4 is the last to be ranked.

As we stated in the previous paragraph, the algorithm (see Figure 2) starts with no ranked players and the maximal domain for the m-dimensional decision variable. It determines the poorest player and then moves recursively (Figure 2, line 31) to solve the smaller problem of ranking the remaining players within the domain where the newly-ranked player yields its ranking value.

Deciding which player to rank next is done in two phases. The first phase (the GET-BEST-VALUE-CANDIDATES call on line 12) finds the optimal value for the next-to-be-ranked and a list of candidate players that can achieve that value if ranked next. In our example, the first set of candidates consists of P_0, P_3 and P_4. If either P_3 or P_4 is chosen first, the search space is restricted to a single point, which is obviously inferior to the solution shown in Subfigure 1(d)). In the second phase (lines 14–30), we find this candidate: it can be shown that this candidate's domain encompasses all the others.

We prove later that the optimal solution is always found if one chooses to rank next the player that restricts the search space the least. In our example ranking P_0



(a) The initial problem.

(b) The second call, after P_0 is chosen for the "poorest player" position.

(c) The third call, after P_1 is ranked the second "poorest."

(d) After P_2 and P_3 tie for third and fourth "poorest," the last call ranks P_4 as fifth "poorest."

Figure 1: Applying our LEXIMIN algorithm to a 2-action, 5-player ($P_0 \dots P_4$) problem. On the horizontal axis we show f_0, the proportion of times action 0 will be chosen (no need to show $f_1 = 1 - f_0$). On the vertical axis are the average payoffs players get when actions are chosen in different proportions. The algorithm calls itself recursively on smaller and smaller versions of the problem. The grey areas show the Domain (width of the grey area) and the LowerBound height of the bottom edge of the grey area) for each call. Once a player has been ranked it is depicted with a dotted line.

first restricts the space to an interval (the projection of the grey rectangle in Subfigure 1(b) on the horizontal axis), while the other two candidates would have restricted the space to each of the two end points of the said interval. We will show later that it is always possible to chose in such a way that no branching is needed.

We draw attention to the third recursive call, corresponding to Figure 1(c). At this point P_2 and P_3 are equally good with respect to both criteria (optimal value and search space restriction), so the order they are ranked in is irrelevant: whichever is ranked first, the other will be ranked next without further space shrinkage. For this reason we rank them both in the same step (line 31).

See the Appendix for a proof that this algorithm always terminates and produces the correct result.

4.1 Implementation with Linear Programming

In this section we provide a implementation of the LEXIMIN algorithm using linear programming to solve MAXIMIZE calls in Figures 3 and 4. The applicability of linear programming (LP) to a wide range of problems in many different fields attracts a great deal of interest, resulting in the discovery of many algorithms for solving problems of this form. Casting our algorithm in terms of linear programming allows the use of many of these algorithms drawn from this literature.

4.1.1 Linear Programming

In the standard form, an LP instance is the problem of maximizing $c^T x$, subject to $Ax = b$ and $x \geq \overrightarrow{0}$. All coefficients are real numbers ($c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$) and so are the decision variables ($x \in \mathbb{R}^n$). It is easy to include inequalities in the list of constraints: for each inequality constraint one introduces a new variable, usually referred to as "slack" (for $\leq$ constraints) or "surplus" (for $\geq$ constraints). For example, if one wishes to optimize $f(x,y) = x + 3y$ subject to $x + y \leq 8$ and $2x - 3y \geq 2$, then one should optimize $f(x,y) = x + 3y + 0v_1 + 0v_2$ subject to the constraints $\langle x + y + v_1 = 8 \rangle$ and $\langle x - 3y - v_2 = 2 \rangle$.

An important observation about our LEXIMIN algorithm is that the domains can be represented as a set of linear constraints. Therefore the intersection of two domains could be regarded as the union of their constraint sets. It turns out that there is an even simpler solution which skips merging constraint lists and constraint redundancy detection: the constraints for each optimization subproblem can be generated on the spot, using only the value of LowerBound and a player-indexed array stating which players have been ranked and at which value. Since the targets of the MAXIMIZE subproblems are players' utility functions, which are linear combinations of a policy's components, these subproblems fit the LP restrictions. We will show how to cast all MAXIMIZE calls in the LP framework.

The first place we use MAXIMIZE calls is in the function GET-BEST-VALUE-CANDIDATES, called on line 12 of Figure 2 to find the list of players with the highest possible utility while being the "poorest." We have a naive implementation and an improved, but harder to prove implementation. Our naive implementation (Figure 3) is based on solving multiple optimization subproblems: for each player P_i, we simply require the maximization of the utility function $U(P_i, f)$, subject to constraints on f

```
1: call LEXIMIN(Utility function U, All players P, F = {f ∈ [0,1]^|A| | Σ_{a∈A} f_a = 1}, −∞)
2:     // A is the set of possible actions
3:     // Each f is a policy. f_a is the probability of performing action a in policy f
4:     // U(P_i, f) is the utility for player P_i ∈ P in policy f ∈ F

5: procedure LEXIMIN(U, UnrankedPlayers, Domain, LowerBound)
6:     // Domain is the domain of legal policies, expressed as a set of constraints
7:     // LowerBound is the maximum utility, constrained by Domain of the most recently ranked player
8:     returns: set of all fair policies
9:     if UnrankedPlayers = ∅ then
10:        return Domain                    // ∀f ∈ Domain are optimal solutions.
11:    end if
12:    (LowerBound, BestValueCandidates) ← GET-BEST-VALUE-CANDIDATES(U, UnrankedPlayers, Domain, LowerBound)
13:    // Focus on the region where no unranked player has a smaller utility than LowerBound.
14:    Domain ← Domain ∩ {f ∈ F | ∀P_k ∈ UnrankedPlayers : U(P_k, f) ≥ LowerBound}
15:    // Find a candidate player whose domain encompasses the other candidates' domains. There must be one.
16:    BestDomainCandidate ← an arbitrary member of BestValueCandidates
17:    for each P_i ∈ BestValueCandidates do
18:        if SUPERSET(Domain, P_i, BestDomainCandidate, LowerBound) then
19:            BestDomainCandidate ← P_i
20:        end if
21:    end for
22:    // Find all candidates with the same domain as BestDomainCandidate.
23:    BestDomainCandidates ← ∅
24:    for each P_i ∈ BestValueCandidates do
25:        if SUPERSET(Domain, P_i, BestDomainCandidate, LowerBound) then
26:            BestDomainCandidates ← BestDomainCandidates ∪ {P_i}
27:        end if
28:    end for
29:    // Restrict the domain further to the minimum-utility region of BestDomainCandidate, and recurse
30:    Domain ← Domain ∩ {f ∈ F | U(BestDomainCandidate, f) = LowerBound}
31:    return LEXIMIN(U, UnrankedPlayers \ BestDomainCandidates, Domain, LowerBound )
32: end procedure
```

Figure 2: Algorithm for optimizing the LEXIMIN function over the players' average payoffs.

that guarantee that for those policies f, P_i has the lowest utility of any player. The MAXIMIZE call provides us with a value for f and its corresponding minimum utility. All players yielding maximal value are returned, along with their associated f values. Later, we will present an improved version requiring solving a single optimization subproblem.

The constraints result from the following GET-BEST-VALUE-CANDIDATES function invariant, which is part of the proof of correctness for our LEXIMIN algorithm:

$$\begin{aligned}(1)\quad \mathsf{Domain} = \{f \mid \forall a \in A,\ f_a \geq 0 \wedge \sum_{a \in A} f_a = 1 \wedge \\ \forall P_j \in \mathsf{UnrankedPlayers}, U(P_j, f) \geq \mathsf{LowerBound} \wedge \\ \forall P_k \in \mathsf{RankedPlayers}, U(P_k, f) = P_k\text{'s ranking value}\}\end{aligned}$$

Aside from the constraints that never change ($\langle \sum_{a \in A} f_a = 1 \rangle$ and the m constraints $\langle f_a \geq 0 \rangle$, which are implied and need not be included) there is always exactly one constraint associated with each player P_k: $\langle U(P_k, f) = P_k\text{'s ranking value} \rangle$ if $P_k \in \mathsf{RankedPlayers}$ and $\langle U(P_k, f) \geq U(P_i, f) \rangle$ otherwise. P_i is the player whose utility is maximized in the current LP.

The second place we use MAXIMIZE is when we compare domains in the SUPERSET function (see Figure 4, lines 5–6) to determine if one encompasses the other. The key operation here is the predicate $D_i \setminus D_j \neq \emptyset$, whose value we compute by solving a linear program: since P_j gets a utility equal to LowerBound from every policy in D_j, if the largest utility value P_j can get while using a policy in D_i exceeds LowerBound, then it must be that $D_i \setminus D_j \neq \emptyset$. The one-to-one mapping of constraints to players is as follows: $\langle U(P_k, f) = P_k\text{'s ranking value} \rangle$ if $P_k \in \mathsf{RankedPlayers}$, $\langle U(P_k, f) = \mathsf{LowerBound} \rangle$ if $P_k = P_i$ and $\langle U(P_k, f) \geq \mathsf{LowerBound} \rangle$ otherwise.

4.2 Reducing the number of LP calls

We can reduce the number of LPs our algorithm has to solve by using an alternative implementation for the GET-BEST-VALUE-CANDIDATES function (Figure 5). We are going to use a common "trick" from convex optimization: the "epigraph form" [1, 5]. Instead of finding the highest utility for each unranked player pro-

```
1: procedure GET-BEST-VALUE-CANDIDATES(U, UnrankedPlayers, Domain, LowerBound)
2:   returns: the utility for the next-to-be-ranked player and the list of possible candidates

3:   BestValueCandidates ← ∅
4:   for each P_i ∈ UnrankedPlayers do
5:     [v, f] ← MAXIMIZE U(P_i, f) subject to f ∈ Domain and ∀P_k ∈ UnrankedPlayers \ {P_i} : (U(P_k, f) ≥ U(P_i, f))
6:     // P_i cannot be a candidate if LP returned v < LowerBound or INFEASIBLE
7:     if v = LowerBound then
8:       BestValueCandidates ← BestValueCandidates ∪ {P_i}
9:     else if v > LowerBound then
10:      BestValueCandidates ← {P_i}
11:      LowerBound ← v
12:    end if
13:  end for
14:  return (LowerBound, BestValueCandidates)
15: end procedure
```

Figure 3: The naive GET-BEST-VALUE-CANDIDATES procedure used by the LEXIMIN algorithm.

```
1: procedure SUPERSET(Domain, P_i, P_j, LowerBound)
2:   returns: TRUE if the minimum-utility region of P_i is a non-strict superset of that of P_j, within the constraints Domain, else FALSE

3:   D_i = Domain ∩ {f ∈ F | U(P_i, f) = LowerBound}
4:   D_j = Domain ∩ {f ∈ F | U(P_j, f) = LowerBound}
5:   [v_j, f'] ← MAXIMIZE P_j subject to D_i
6:   [v_i, f''] ← MAXIMIZE P_i subject to D_j

7:   // Set-difference is difficult to compute using constraints, but note that D_i \ D_j ≠ ∅ iff v_j > LowerBound
8:   // Likewise, D_j \ D_i ≠ ∅ iff v_i > LowerBound
9:   return v_j ≤ LowerBound ∧ v_i > LowerBound          // That is, D_i \ D_j = ∅ ∧ D_j \ D_i ≠ ∅
10: end procedure
```

Figure 4: SUPERSET procedure used by the LEXIMIN algorithm.

vided it can be next-to-be-ranked, we try to maximize a new variable, subject to the constraints that the utility of each unranked player is higher or equal to that variable. Intuitively, the new variable will hold the best possible utility value for the next-to-be-ranked, whoever it is. Solving this LP provides us with a policy and the final LowerBound for that recursion level. We then populate BestValueCandidates with all players whose utility is LowerBound given that particular policy. Although we might miss some players that meet the "best value" criteria under other policies, they would have failed the "largest domain" test: we only care about those players with maximally large domains and no matter what policy the LP returns, it will be contained in those players' domains. Therefore the improved algorithm is still correct.

This new version replaces |UnrankedPlayers| LP calls with just a single call in the first phase (the "best value" test), and it also has the potential to decrease the number of LPs solved in the second phase[3] (the "largest domain" test). The LPs in the new version of GET-BEST-VALUE-CANDIDATES have $m+1$ variables and $n+1$ constraints. The constraints are: $\langle \sum_{a \in A} f_a = 1 \rangle$; and additionally $\forall k = 1 \ldots n$, $\langle U(P_k, f) = P_k$'s ranking value$\rangle$ if $P_k \in$ RankedPlayers and $\langle U(P_k, f) - y \geq 0 \rangle$ otherwise. Since all LP variables are required to be non-negative, and y will be set in turn to each utility level, a sufficient condition is for coefficients to be non-negative. This is easy to do: add to all coefficients sufficiently large positive constant.

4.3 Computational complexity

The LEXIMIN algorithm makes a linear number of LP calls at each level, before calling itself recursively. Since at least one player is ranked at each level of recursion, the recursion goes at most n levels deep. Therefore our algorithm makes $O(n^2)$ LP calls.[4]

There are two main classes of LP algorithms, each with its own computational complexity. First are variants of the Simplex algorithm, likely the most popular linear programming technique [7], but for which there are no polynomial worst-case guarantees. Some variants of the Simplex algorithm have been proved to run in exponen-

[3] We revisit the example in Figure 1(a) to illustrate the secondary advantage of the new implementation of GET-BEST-VALUE-CANDIDATES. The first version would have returned BestValueCandidates $= \{P_0, P_3, P_4\}$, while the new one returns BestValueCandidates $= \{P_0, P_3\}$ (or $\{P_0, P_4\}$), and so D_0 is compared against only one of D_3 or D_4, but not both.

[4] A tighter upper bound is n^2 LP calls, if the second phase is rewritten into a single loop of |BestValueCandidates| $- 1$ iterations, with 2 LP calls per iteration.

1: **procedure** GET-BEST-VALUE-CANDIDATES(U, UnrankedPlayers, Domain, LowerBound)
2: ***returns:*** *the utility for the next-to-be-ranked player and the list of possible candidates*

3: $[v, f] \leftarrow$ MAXIMIZE y subject to $f \in$ Domain and $\forall P_k \in$ UnrankedPlayers : $(U(P_k, f) \geq y)$
4: BestValueCandidates $\leftarrow \{P_i \in$ UnrankedPlayers$|U(P_i, f) = v\}$
5: **return** (v, BestValueCandidates)
6: **end procedure**

Figure 5: Improved version of GET-BEST-VALUE-CANDIDATES.

tial time in carefully constructed instances. The second class contains linear programming algorithms proven to run in polynomial time if the linear program's coefficients are restricted to fixed-length representation [7]. For our purposes, rational values in our reward function translate directly to fixed-length coefficients in the linear programs. Note, however, that although algorithms in this second class are polynomial in the worst case, Simplex algorithms are usually preferred by practitioners.

All LPs needed have at most $m+1$ variables and $n+1$ constraints. However, the output from solving one LP could appear on the right hand side of a constraint in subsequent LPs, so the bit-length (i.e. the number of bits required to encode an LP) of the sequence of LPs we solve could grow out of control.

One can use Tardos' algorithm [7], which solves an LP over some number of operations bounded by a polynomial[5] in n, m, and the size of the binary representation of matrix of coefficients from the left hand side of the constraints, but not the size of the numbers on the right hand side of the constraints. This makes our algorithm polynomial under the algebraic complexity model, as it executes a polynomial number of arithmetic operations, but we don't have a bound for the operands' bit-lengths.

We can still offer polynomial time under the logarithmic complexity model (the cost of arithmetic operation depends on the bit length of the operands) if a floating-point, fixed-length representation is used. This should be enough for most practical cases, as the user is allowed to set the precision. Our algorithm is guaranteed to finish in polynomial time, as the bit-length of the LPs is constant throughout the process. A large number of polynomial time LP algorithms have been proposed in the literature, and the choosing the right one might depend on the size of the problem. For instance, if there are many players and relatively fewer joint actions, then Karmarker's algorithm may be best. Conversely, if there are many joint actions relative to the number of players, then one might choose some variant of the basic ellipsoid method, whose running time is polynomial in the length of the binary encoding of the LP and $min(m,n)$.

Since one would not be using exact values, one should slightly relax the "=" relation: $a = b$ is true if $|a-b| < \varepsilon$. This affects the algorithm in two places: line 7 in Figure 3 and line 9 in Figure 4. We informally validated this by comparing the approximative solutions produced by our algorithms coupled with the interior-point LP solver from GLPK [6] against exact solutions obtained when coupled with the Simplex algorithm [4] with explicitly represented rational numbers (i.e. fractions). The results showed no difference in the players' ranking order, and only minor propagated differences in the actual values for policies and utilities as would be expected given the change from fractions to decimals.

[5]Tardos' algorithm only works for LPs guaranteed to be feasible, so it can only be use with our second implementation of GET-BEST-VALUE-CANDIDATES (Figure 5).

5 Conclusion

In this paper we proposed an algorithm for finding the set of LEXIMIN optimal solutions with respect to average payoff in infinitely repeated games. The algorithms is more generally applicable to any problem with multiple linear objective functions defined over a multi-dimensional continuous and bounded decision variable space. Our approach relies on solving a quadratic number of LPs, and is polynomial with respect to the algebraic complexity model if the coefficients are rational. It is also polynomial under the logarithmic complexity model if the coefficients use a fixed length, albeit floating point, representation.

This work is our initial study in "fair" multiagent interaction: we have started by reducing the problem to a single actor whose decisions affect multiple agents differently. Even this simplification turns out to be non-trivial. In immediate future work, we will examine what (likely significantly greater) tradeoffs must be made in order to extend to the multiple-actor case.

6 Acknowledgments

Our thanks to Liviu Panait, Hakan Aydin, Rob Axtell, and Deepankar Sharma for their insight. Most importantly, we are indebted to Octav Olteanu, to whom we owe thanks for the proof for Lemma 1.

A Appendix

Lemma 1. *There always exists some player P_i in* BestDomainCandidates *such that $D_i \supseteq D_j$ holds $\forall P_j \in$* BestValueCandidates*. Or, equivalently, $\exists P_i$ such that $D_i =$* Domain*.*

Proof. Let v be the optimal payoff value for the next-to-be-ranked player. For every player $P_i \in$ BestValueCandidates[6] that can achieve it, D_i is defined as the set of f values for which P_i is the next poorest: $D_i = \{f | U(P_i, f) = v \wedge \forall P_k : U(P_k, f) = v\}$. We want to prove that $\exists P_j$ so that $D_j = \bigcup D_i$.

We prove this lemma using convex analysis [14]. We make the following notations: D refers to the value of Domain at line 9, while D' is the value of Domain at line 14 in Figure 2. It is trivial to show that $D' = \bigcup D_i$.

First we note that all domains (i.e. D, D' and D_i) are always convex. We prove the convexity by induction. D is initially convex. If D is convex, then so are D' and D_i, because every constraint is a convex set, and the intersection of convex sets is convex. The set $\{f | U(P_i, f) = v\}$ corresponding to a constraint $\langle U(P_i, f) = v \rangle$ is convex because for any 2 points f_1 and f_2 that satisfy it, so does every point in between them ($\alpha U(P_i, f_1) + (1 - \alpha) U(P_i, f_2) = U(P_i, \alpha f_1 + (1 - \alpha) f_2) = v$) because P_i's utility is a linear function. Similarly, one can show that constraints such as $\langle U(P_i, f) > v \rangle$ and $\langle U(P_i, f) \geq v \rangle$ define convex sets, too. At the next level D is one of these D_i sets, so all domains are convex.

If D' consists of a single point then the domain D_i for every candidate P_i is equal to D', which satisfies the lemma. We now treat the case when D' consists of more than a single point.

Given that D' is both convex and non-empty, according to theorem 6.2 in [14], D' has non-empty *relative interior*. The relative interior of a convex set is the set's interior with respect to L, the largest *affine set*[7] it generates. Intuitively, if D' is a point, then L is that point; if D' is a line segment, then L is the line the segment lies on; if D' is a polygon, then L is its supporting plane.

Let λ be the *Lebesgue measure*[8] defined over L. Since D' has a non-empty interior with respect to L, it results that $\lambda(D') > 0$. Since $D_j = \bigcup D_i$, there must exist a P_k such that $\lambda(D_k) > 0$. Otherwise we would run into the following contradiction: $0 < \lambda(D') \leq \sum \lambda(D_i) = 0$.

Since P_k has constant utility throughout D_k, and $D_k \subseteq D' \subset L$ and $\lambda(D_k) > 0$, it follows that P_k has constant utility on the entire affine set L. However D_k can be rewritten as $\{f \in D' | U(P_k, f) = v\}$, so it implies that $D_k = D' = \bigcup D_i$, and the lemma is proved. □

Lemma 2. *Given two arbitrary players $P_i, P_j \in$* BestValueCandidates *such that $D_j \subset D_i$, for every policy $f_j \in D_j$ we can find a leximin superior policy in $D_i \setminus D_j$.*

Since $D_i \setminus D_j \neq \emptyset$, we arbitrarily chose some $f_i \in D_i \setminus D_j$. In a nutshell, if f_i is not leximin superior to f_j, then any policy point f on the open-ended segment (f_i, f_j) must be.

Let $v =$ LowerBound. $f_j \in D_j \Rightarrow U(P_i, f_j) = U(P_j, f_j) = v$ and $f_i \in D_i \setminus D_j \Rightarrow U(P_i, f_i) = v$ and $U(P_j, f_i) > v$. We already showed in the proof for Lemma 1 that D_i is convex, so $f \in D_i$. Since $U(P_j, \cdot)$ is a linear functional, and $U(P_j, f_j) = v$ and $U(P_j, f_i) > v$, then it must hold that $U(P_j, f) > v$, so $f \in D_i \setminus D_j$.

The sorted utility vectors corresponding to the policies f and f_j are identical for the first $|$RankedPlayers$|$ positions. After that, for f_j there are at least two players ranked at value v. f is leximin superior to f_j, because there cannot be a second player $P_k \in$ UnrankedPlayers to be ranked, alongside P_i, at value v under policy f without being ranked at the same value for policy P_j. That is because $U(P_k, f) = v$ and $U(P_k, f) > v$ would make $U(P_k, f_i) < v$, which is impossible: D_i is the set of policies where all unranked players get at least as much as P_i, which is v. Therefore, either f_i or f is leximin superior to f_j, which concludes the proof for Lemma 2.

Theorem 1. *The algorithm in Figure 2 always terminates and returns the set of leximin-optimal policies.*

Proof. We start by proving that the invariant in Equation 1 holds at the beginning of each LEXIMIN call. Initially there are no ranked players and LowerBound $= -\infty$, so Domain $= \{a \in A | f_a \geq 0 \wedge \sum_{a \in A} f_a = 1\}$ therefore the invariant holds. Whenever a new player P_j is ranked, LowerBound is equal to that player's ranking value and its domain D_j becomes the new Domain, which satisfies the invariant for the next recursion level. Therefore Equation (1) holds all throughout recursion.

A similar argument guarantees Domain $\neq \emptyset$ is always true. At the time of the first procedure call Domain $\neq \emptyset$. After the first loop LowerBound $> -\infty$, as all players get a finite average payoff out of every policy. If P_i is the last player to increase LowerBound, $D_i \neq \emptyset$, since the linear program for optimizing P_i found a point in D_i. Therefore BestValueCandidates $\neq \emptyset$ and $\forall P_i \in$ BestValueCandidates $\Rightarrow D_i \neq \emptyset$, and one of them will be passed on to the next recursive call as Domain.

[6] In this lemma all $P_i \in$ BestValueCandidates, so we will omit it.

[7] An affine set $S \subseteq \mathbb{R}^m$ -also referred to in the literature as affine variety, linear variety, affine manifold or flat, is the locus of points satisfying a set of polynomial equations. A more intuitive definition would be that "a set $S \subseteq \mathbb{R}^m$ is affine if it contains the line through any two points in it" [5]. For example, points, lines and planes are 0, 1 and 2-dimensional affine sets.

[8] The Lebesgue measure quantifies the "size" of subsets of $\mathbb{R}^p$. In $\mathbb{R}$ it measures segments' lengths; in $\mathbb{R}^2$ it measures area and volume in $\mathbb{R}^3$. For example, the Lebesgue measure of segment $[a, b]$ is equal to $b - a$ in $\mathbb{R}$ but is equal with zero in $\mathbb{R}^k$, $k \geq 2$.

A set D_i is guaranteed to contain at least one point (the one found by the LP call) such that all the ranked players keep their optimal ranking and all the unranked players are sure to receive at least LowerBound. Therefore in all calls after the first one Domain is actually some set D_i from the previous iteration, and since it is not empty, one can always propose a candidate to BestValueCandidates set by selecting the player with the smallest payoff value in some point from Domain. If BestValueCandidates $\neq \emptyset$, then Domain $\neq \emptyset$ after the domain restriction at line 14. We proved that if Domain $\neq \emptyset$ on some recursion level then Domain $\neq \emptyset$ on the next level. Therefore, by induction, Domain can never be empty. This also imply that at line 14 BestValueCandidates cannot be empty either.

Since there are a finite number of players and at every recursive call we rank at least a player, we are guaranteed that the LEXIMIN algorithm eventually stops and a complete ranking of players is produced. The rest of this proof consists of demonstrating that the algorithm returns the optimal solution(s).

At every procedure call we determine the highest possible payoff for the next-to-be-ranked player, because every unranked player is considered for that position. Thus BestValueCandidates contains the optimal choice for the next-ranked player.

When there are multiple candidates, we rank those whose associated domains include the domains of all the other candidates. It is always possible to apply this strategy (see Lemma 1) and it leads to the set of leximin optimal solutions (see Lemma 2). When multiple candidates have maximally large domains, those candidates can be ranked in any order, since no other player can be selected until all players in BestDomainCandidates are ranked. Instead of ranking each one in a separate recursive call, we chose to rank them all at once.

Since our algorithm is always able to determine correctly which is should be the next-ranked player(s), we conclude that it always finds the set of optimal solutions. □

References

[1] Sylvain Bouveret and Michel Lemaître. Comparison of two constraint programming algorithms for computing leximin-optimal allocations. In *Proceedings of the Workshop on Modelling and Solving Problems with Constraints*, Riva del Garda, Italy, August 2006.

[2] Steven J. Brams. Fair division. In Barry R. Weingast and Donald Wittman, editors, *Oxford Handbook of Political Economy*. Oxford University Press, 2005.

[3] Yann Chevaleyre, Paul E. Dunne, Ulle Endriss, Jérôme Lang, Michel Lemaître, Nicolas Maudet, Julian Padget, Steve Phelps, Juan A. Rodríguez-Aguilar, and Paulo Sousa. Issues in multiagent resource allocation. *Informatica*, 30:3–31, 2006.

[4] George B. Dantzig and Mukund N. Thapa, editors. *Linear Programming 2: Theory and Extensions*. Springer-Verlag, 2003.

[5] Haitham Hindi. A tutorial on convex optimization. In *Proceedings of the American Control Conference*, volume 4, pages 3252–3265, June 2004.

[6] Andrew Makhorin. GNU Linear Programming Kit (GLPK) v4.11. www.gnu.org/software/glpk/glpk.html, 2006.

[7] Nimrod Megiddo. On the complexity of linear programming. In T. F. Bewley, editor, *Advances in Economic Theory — The Fifth World Congress*, volume 12 of *Econometric Society Monographs*, pages 225–258. Cambridge University Press, Cambridge, UK, 1987.

[8] Paul Morris, Robert Morris, Lina Khatib, Sailesh Ramakrishnan, and Andrew Bachmann. Strategies for global optimization of temporal preferences. In *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 2004.

[9] Dritan Nace and James B. Orlin. Lexicographically minimum and maximum load linear programming problems. Technical Report 4584-05, MIT Sloan, 2005.

[10] Wlodzimierz Ogryczak, Michal Pióro, and Artur Tomaszewski. Telecommunications network design and max-min optimization problem. *Journal of Telecommunications and Information Technology*, 3, 2005.

[11] Jos A. M. Potters and Stef H. Tijs. The nucleolus of a matrix game and other nucleoli. *Mathematics of Operations Research*, 17(1):164–174, 1992.

[12] Božidar Radunović and Jean-Yves Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness with Applications. *ACM/IEEE Transactions on Networking*, 2006. Forthcoming.

[13] Jack Robertson and William Webb. *Cake-Cutting Algorithms: Be Fair If You Can*. A. K. Peters, Ltd., 1998.

[14] R. Tyrrell Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, New Jersey, 1970.

[15] Katja Verbeeck, Ann Nowé, Johan Parent, and Karl Tuyls. Exploring selfish reinforcement learning in repeated games with stochastic rewards. *The Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS)*, 2006. Forthcoming.

[16] Katja Verbeeck, Johan Parent, and Ann Nowé. Homo egualis reinforcement learning agents for load balancing. In *Innovative Concepts for Agent-Based Systems: First International Workshop on Radical Agent Concepts WRAC*, volume 2564 of *Lecture Notes in Computer Science*, pages 81–91, January 2003.

[17] Ronald R. Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):183–190, 1988.

[18] Yunkai Zhou. *Resource Allocation in Computer Networks: Fundamental Principles and Practical Strategies*. PhD thesis, Drexel University, 2003.