The Minimum Unique Positive Integer Game

CO 456 Project 1, Fall 2007

Due Date: October 18 2007, 10 am

In this project you are asked to analyze a strategic game and implement a Java strategy for it.

Format

In this project you can work alone or in a team of 2. Each team submits *one writeup* and *one Java submission*; both members on a team receive the same grade, and are graded the same as the solo teams. You are not permitted to discuss the project with anybody from another team.

If you and another student wish to form a team, notify the email address below. **If you would like to be on a team but need a partner,** email the address below by **October 9** and you will be teamed up on a first-come, first-serve basis. Any requests after that date may not be fulfilled.

`co456@student.math.uwaterloo.ca`

1 The Game

We call the game in this project the *minimum unique positive integer game*, and it works as follows. Each player's action set is the set of all positive integers, which we will denote by $\mathbb{P}$. The person who picks the minimum unique number, out of the chosen numbers, wins. In the event that none of the numbers chosen are unique, then nobody wins. Here are three examples.

- Oskar picks 2, John picks 2, Ernst picks 10. Ernst wins.
- Oskar picks 3, John picks 1, Ernst picks 2. John wins.
- Oskar picks 1, John picks 1, Ernst picks 1. Nobody wins.

We will denote this game by $\mathcal{G}$. The payoff to player i for action profile $a = (a_i, a_{-i})$ is

$$u_i(a) = \begin{cases} 1, & \text{if } (a_j \neq a_i \text{ for each player } j \neq i) \text{ and (for each player } j \text{ such that} \\ & \quad a_j < a_i, \text{ there is a player } k \neq j \text{ such that } a_k = a_j). \\ 0, & \text{otherwise.} \end{cases}$$

It is possible to interpret this as some sort of "creativity market" game: each positive integer represents an idea, and the magnitude of the number represents the complexity of the idea. The consumers will only purchase "unique" products (they don't like rip-offs) but they will only buy the least complicated one (since it will generally be cheapest and easiest to use). However, this is something of a stretch, and we suggest that you consider the game as an interesting but abstract strategic game.

This game has previously been used as the mechanism for a charity lottery. Players were able to submit as many entries as they liked, but at a fixed cost per entry. Out of about 1200 entries, the winning integer was 28. The full details and data are available at `http://web.mit.edu/thinkBIG/challenge/`. If you find any more prior occurrences of this game, we would like to hear about them.

2 Writeup Questions

2.1 Pure Equilibria

In the rest of this project, n denotes the number of players. In this section we ask you to prove some basic facts about the game's pure equilibria.

Question 1 (4 points)**.** *When $n = 2$, prove that an outcome a is a Nash equilibrium for $\mathcal{G}$ if and only if there is a player i for which $a_i = 1$.*

Question 2 (5 points)**.** *When $n = 3$, prove that an outcome a is a Nash equilibrium for $\mathcal{G}$ if and only if both of the following hold:*

1. *there is a player i for which $a_i = 1$, and*
2. *there is a player i for which $a_i \neq 1$.*

2.2 Mixed Equilibria

It is fairly easy to determine all mixed equilibria of $\mathcal{G}$ when there are $n = 2$ players, but for $n \geq 3$ things are much more interesting. Our approach in this project is to look at the special case of *symmetric* Nash equilibria. The following definition is used to make the notation less cumbersome.

Definition 1. *Let $p = (p_1, p_2, \dots)$ be a sequence of reals for which $p_i \geq 0$ for all i, and $\sum_{i \in \mathbb{P}} p_i = 1$. We say that p* represents *the mixed action profile α defined by $\alpha_j(i) = p_i$ for each player j and each $i \in \mathbb{P}$.*

In the rest of this section, we will develop some necessary conditions for p to represent a (symmetric mixed) Nash equilibrium. On the surface, it seems like this game is essentially finite. For example, if there are only three players, why would a rational player pick a large number like 100? (Note, something like this actually occurred in the charity tournament.) Given this view, and since we know that all finite games have mixed equilibria, it seems like there should be a symmetric mixed equilibrium where all players choose only small numbers. In the next problem you will show this is *not* true.

Question 3 (6 points)**.** *Let there be $n \geq 3$ players. If p represents a Nash equilibrium, show that there is no maximum integer k for which $p_k > 0$.*

Hint: suppose for the sake of contradiction that such a k exists. Let α be the mixed action profile represented by p. Show that k is not a best pure response to α_{-1}; then use the Support Characterization.

As a result of Question 3 we know that positive probabilities are assigned to an infinite number of $i \in \mathbb{P}$. Is it possible that an equilibrium strategy "skips" some values? We ask you to show this is not possible.

Question 4 (6 points)**.** *Let there be $n \geq 3$ players. If p represents a Nash equilibrium, using the result of Question 3, show that $p_i > 0$ for each $i \in \mathbb{P}$.*

Finally, once we have Question 4, there is a method to approximately deduce the p values that form a Nash equilibrium. The following question embodies the first step of this method.

Question 5 (4 points)**.** *Let there be exactly $n = 3$ players. If p represents a Nash equilibrium, using the result of Question 4, show that the following equation holds:*

$$(1 - p_1)^2 = (1 - p_1 - p_2)^2 + p_1^2.$$

3 Code Fragment and Tournament Structure

Your team will be asked to submit a Java code fragment that will be entered in a tournament against the other teams. The simplest way to model the minimum unique positive integer game as a strategic game is with $\mathcal{G}$, where you only pick a single integer; however, we also want to capture some of the complexity of the charity lottery, where you can make multiple choices. Hence, each time we run your code fragment, it will generate `numChoices` positive integers. In each "round" of the tournament, we ask several strategies to generate some lists in this way, and the strategy with the minimum unique integer over all the lists gets 1 point.

Specifically, your code fragment will include a procedure `getChoices` with the following signature: `void getChoices(int numChoices, int[] myChoices, int totalChoices, int numPlayers)`. This procedure should generate `numChoices` integers and place these values in the array `myChoices`. The other two arguments are as follows:

- `totalChoices` is the sum of `numChoices` for all of the different players in that round
- `numPlayers` is the number of distinct strategies that will play in that round

If you forget to fill in some of the array's entries, or put a number that is less than 1, it will be treated as if you had entered $2^{31} - 1$, the maximum `int` value in Java. You can assume $20 \geq$ `numChoices` > 0. **Note, it is never a good idea for your code to enter duplicate values into `myChoices`!** A sample strategy is shown below.

```
public class UniformPlayer extends Player {
    void getChoices(int numChoices, int[] myChoices,
            int totalChoices, int numPlayers) {
        // picks numChoices random numbers from
        // the uniform distribution on [1,..,range]
        int range = Math.max(2*numChoices, 3*numPlayers);

        int choicesLeft = numChoices;
        for (int i=1; i<=range; i++)
            if (Math.random() <= choicesLeft/(1.0*(range-i+1)))
                myChoices[numChoices - choicesLeft--] = i;
    }
}
```

We will leave the 4 sample strategies in the tournament; let n denote the total number of strategies, i.e., four plus the number of teams who submitted code.

The tournament will consist of two equally-weighted halves. In the first half, `numChoices` will always be equal to 1, and `numPlayers` will vary uniformly between 2 and n. In the second half, `numPlayers` will again vary uniformly between 2 and n and additionally, `numChoices` will vary between 1 and 20. In any given round, the different strategies that play will have different values of `numChoices`, although we will ensure that all teams get "equal opportunity" in terms of the overall number of choices (see `runRotatedRounds` in the code or ask for more details). The total number of rounds in each half is $100n(n-1)$.

You can access the code at

`http://www.math.uwaterloo.ca/~dagpritc/co456/Project1Tournament.java`

You should be able to run the code as written, and after typing in your own strategy, running the code will test it against the 4 "default" strategies.

3.1 Submission and Grading

In your writeup, include an answer to the following question.

Question 6 (2 points)**.** *Describe your submission to the tournament, and give us an idea of why you think your strategy has a chance to win.*

You need to send your code fragment to the following email address by 10 AM on October 18.

`co456@student.math.uwaterloo.ca`

Your team should bring one hard copy of your writeup to class on that day.

Your team can obtain a maximum of 27 points on the writeup. Another 5 points will be granted for *submission of a strategy.* (If your strategy fails to compile, or for some reason does not meet the required specifications, you will lose some of these points.) Finally, 8 points will be given based on the performance of your strategy in the tournament. So there are 40 points in total.

- The top team will get 9/8 points
- The remaining top 1/8 of all teams will get 8/8
- The next 1/8 of all teams will get 7/8

⋮

- The bottom 1/8 of all teams will get 1/8.

Technical Requirements

The intent of the tournament is that keeping a "memory," e.g. of the number of matches your strategy has played in so far, is impossible. Likewise, you should not be able to find out anything about your opponents. Hence we make the following restrictions; if you don't know what they mean, don't worry.

Your code should not use the `static` modifier, and you should not read from or write to any global variables, files, ports, pipes, use system calls, etc. In particular, you are not allowed to call `getPlayerClasses` or otherwise use reflection to instantiate the other players' classes. We will re-instantiate your class before every match.