Yale University
CPSC 474/574 - Computational Intelligence for Games
Fall 2020

Assignment 3: Simultaneous Games

Objectives

- to use a linear programming solver to find equilibria for simultaneous games
- to use Barron Theorem 1.3.8 to verify equilibria of simultaneous games

Introduction

Blotto is a simultaneous game for two players in which the players distribute military units over a number of battlefields. Each player has a set number of units to distribute, each unit is equivalent to each other unit, and a unit must be allocated entirely to a single battlefield. Each battlefield is worth a given number of points, and the player who allocates more units to a battlefield wins the points for that battlefield (with the points split evenly in case of a tie, inlcuding a 0-0 tie).

For example, suppose there are four battlefields worth 1, 2, 3, and 4 points respectively and each player has 10 units to distribute. If player 1's distribution is $(1, 0, 4, 5)$ and player 2's distribution is $(1, 1, 3, 5)$ then player 1 wins battlefield 3, player 2 wins battlefield 2, and the players tie on battlefields 1 and 4. Player 1's score is then 5.5 and player 2's score is 4.5.

The objective can be to score more points than your opponent or to score as many points as possible.

Assignment

Write a program called `Blotto` that calculates and verifies equilibrium strategies for Blotto given a point value for each battlefield and the number of units each player has to distribute across those battlefields.

The task for the program to perform (calculate or verify), the parameters of the game (number of units and point value for each battlefield), and the objective (maximize wins or points) will be given as command-line arguments as follows.

- The first command-line argument will be either `--find` or `--verify` to indicate whether to find an equilibrium or verify the equilibrium read from standard input in the format described below. Both arguments can be followed by an optional argument `--tolerance` which it itself followed by a floating point number between 0.0 (exclusive) and 1.0 (inclusive) giving the tolerance (see below). If `--tolerance` is not given then 10^{-6} is used as the default value.
- The next command-line argument will be either `--win` or `--score` to indicate whether to find or verify equilibrium strategies with the objective of maximizing the expected number of wins (with ties counting as ½ a win), or maximizing the expected total points scored.
- For `--find`, the third and fourth command-line arguments will be `--units` and a positive integer giving the number of units to distribute. (For `--verify` the number of units is determined by the values read from standard input.)
- The remaining arguments are positive integers giving the value of each battlefield starting with battlefield 1. The number of battlefields is determined by the number of these arguments and will be positive.

For `--verify`, the mixed strategy to check will be read from standard input where each line gives a unique pure strategy as the number of units to allocate to each battlefield, starting with battlefield 1, and the probability of playing that pure strategy. All values will be separated by a comma with no whitespace. Each number of units will be a positive integer. Each pure strategy will have the same total number of units and the same number of battlefields. Each probability will be a number between 0.0 (exclusive) and 1.0 (inclusive) and the sum of the probabilities will be within the given tolerance of 1.0. Your program should determine whether both players playing the strategy read from input is an equilibrium. Note that because of the imprecision of the linear programming solver and the general imprecision of floating point arithmetic, you will have to allow for some slack in the inequalities (and in the sum of probabilities of the inputs) – as long as none of the inequalities are off by more than the given tolerance then consider the strategy an equilibrium.

Your program should be able to perform any computation up to 10 units and 4 battlefields in a few seconds and up to 15 units over 5 battlefields in several minutes (all times measured in CPU time; on multi-core systems the real time elapsed may be less).

Output

For `--find`, the output should be a mixed strategy in a format readable by `--verify` (the test scripts will run your output through a verifier rather than checking for a character-by-character match with some expected output, so the output needn't be in any specific order) containing only pure strategies with probabilities that exceed the tolerance.

For `--verify`, the output should be either `PASSED` if the strategy is an equilibrium strategy for both players, or any output that does not contain `PASSED` as a substring otherwise (the public tests will discard any lines that do not contain `PASSED` and will perform a character-by-character comparison on the lines that do contain `PASSED`).

Time and Space efficiency

The payoff matrices grow very quickly with the number of battlefields and the number of units, so it is essential to avoid unnecessary work and memory usage. There are two major considerations.

- Time to create the payoff matrix: the matrix will usually be sparse (containing many zeros); can you reduce the amount of time spent calculating the entries in the matrix?
- Space to store the matrix: while you need to store the entire matrix for `--find` since you need to pass it to the solver, for `--verify` you only use each entry once after you've computed it. Since you only need to use each value once, can you avoid storing them all together in one place?

If your implementation passes the public tests within the memory and time limits, then you should be confident that it will pass the private tests.

Examples

Note that there may be multiple equilibria, so your results may not match these for --find. Your solution will processed by a verifier as in the last two examples for grading.

```
[jrg94@cobra code]$ ./Blotto --find --score --units 4 1 2
0,4,1.0
[jrg94@cobra code]$ ./Blotto --find --score --units 5 2 4 5
0,0,5,0.14045462101642173
0,1,4,0.15496211495866577
0,2,3,0.04541673569373383
0,3,2,0.05992422986845587
1,0,4,0.08473480443797905
1,1,3,0.09503788495378677
1,2,2,0.17977268944230548
1,3,1,0.1404546206153363
1,4,0,0.04962114905424416
2,3,0,0.04962114943279007
[jrg94@cobra code]$ ./Blotto --find --tolerance 1e-05 --win --units 5 2 4 5
0,0,5,0.07428558119961398
```

```
0,1,4,0.0628572491429189
0,2,3,0.0514285583023659
0,3,2,0.10857121203986661
1,0,4,0.13142860782053217
1,1,3,0.01714278781855886
1,2,2,0.017143017862373943
1,3,1,0.13142851264458152
2,0,3,0.10857157877490509
2,1,2,0.05142853222724512
2,2,1,0.06285716721342921
2,3,0,0.074285740412162
3,2,0,0.10857132632298054
[jrg94@cobra code]$ ./Blotto --find --win --units 5 2 4 5 | ./Blotto --verify --win 2 4 5
PASSED
[jrg94@cobra code]$ ./Blotto --find --tolerance 1e-05 --win --units 5 2 4 5 | ./Blotto --verify
E[X, (0, 0, 5)] = 5.25999904741337 < 5.499998589596961
```

Submissions

Submit any necessary code, your time log, and a makefile with default target `Blotto`. Note that Python programmers will have to create a script that runs their code with `python3` rather than `pypy3` because the `scipy` module is not available for `pypy3`.