Computational Optimization of The Prisoner's Dilemma

Babur Kocaoglu

April 30, 2021

1 Introduction

We begin with the simplified prisoner's dilemma. This is defined by a matrix A of values. Example:

$$A = \begin{bmatrix} 4 & 0 & 5 \\ 4 & 4 & 3 \\ 2 & 6 & 2 \end{bmatrix}$$

In the classical prisoner's dilemma, we must pick a row and our opponent must pick a column. Here, though, we intend to play perfectly and selfishly: Our opponent knows what we will do and will do what's best for him. We will do what's best for us (which, in this simplified case, is what's worst for him).

In other words, we pick a row such that, when our opponent picks a column, he is forced to pick his own worst possible outcome. However, since he knows what we're going to do, picking a single row outright may be sub-ideal: as such, we will pick row i with a probability p_i, and our opponent must play accordingly.

Knowing the p_i's, our opponent can generate a value vector by simply taking the row vector $p = [p_1, p_2, ..., p_m]$ and multiplying it by A, taking the weighted mean of the rows. Then, he can pick the highest value, thereby giving himself the highest possible (average) payout. This means that while our actions are probabilistic, his action will be deterministic: whatever distribution we present, we can determine what our opponent will do. This lets us create our optimization equation:

$$\arg_p \min(\max_i (pA)_i)$$

with constraints

$$||p||_1 = 1$$

$$p_i \geq 0 \ \ \forall i$$

Returning to our example matrix, A, this becomes

$$\arg_p \min(\max((pA)_1, (pA)_2, (pA)_3)) = \arg_p \min(\max(4p_1+4p_2+2p_3, 4p_2+6p_3, 5p_1+3p_2+2p_3))$$

and noting that $p_3 = 1 - p_1 - p_2$, it simplifies to

$$\arg_p \min(\max(2p_1 + 2p_2 + 2, -6p_1 - 2p_2 + 6, 3p_1 + p_2 + 2))$$

Now, fix $\max(2p_1 + 2p_2 + 2, -6p_1 - 2p_2 + 6, 3p_1 + p_2 + 2) = C$, and solve for the system of equations

$$\begin{bmatrix} 2p_1 & +2p_2 & +2 & = C \\ -6p_1 & -2p_2 & +6 & = C \\ 3p_1 & +p_2 & +2 & = C \end{bmatrix}$$

$$\begin{bmatrix} 0 & +4p_2 & +12 & = 4C \\ 2p_1 & +2p_2 & +2 & = C \\ 3p_1 & +p_2 & +2 & = C \end{bmatrix}$$

$$\begin{bmatrix} 0 & +4p_2 & +12 & = 4C \\ 6p_1 & +6p_2 & +6 & = 3C \\ 6p_1 & +2p_2 & +4 & = 2C \end{bmatrix}$$

$$\begin{bmatrix} 0 & +4p_2 & +12 & = 4C \\ 0 & +4p_2 & +2 & = C \\ 6p_1 & +2p_2 & +4 & = 2C \end{bmatrix}$$

$$\begin{bmatrix} 0 & +4p_2 & +12 & = 4C \\ 0 & +4p_2 & +2 & = C \\ 6p_1 & -6p_2 & 0 & = 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & +10 & = 3C \\ 0 & +4p_2 & +2 & = C \\ p_1 & -p_2 & 0 & = 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & +\frac{10}{3} & = C \\ 0 & +4p_2 & +2 - \frac{10}{3} & = 0 \\ p_1 & -p_2 & 0 & = 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & +\frac{10}{3} & = C \\ 0 & +4p_2 & -\frac{4}{3} & = 0 \\ p_1 & -p_2 & 0 & = 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & +\frac{10}{3} & = C \\ 0 & +p_2 & -\frac{1}{3} & = 0 \\ p_1 & -p_2 & 0 & = 0 \end{bmatrix}$$

which gives us our answer: $p_1 = p_2 = p_3 = \frac{1}{3}$ and $C = \frac{10}{3}$.

$$p = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

This process, however, is not always so simple. Consider another example.

$$A = \begin{bmatrix} 8 & 3 & 8 \\ 0 & 1 & 5 \\ 5 & 4 & 3 \end{bmatrix}$$

$$\arg_p \min(\max(8p_1 + 5p_3, 3p_1 + p_2 + 4p_3, 8p_1 + 5p_2 + 3p_3)) =$$

$$\arg_p \min(\max(3p_1 - 5p_2 + 5, -p_1 - 3p_2 + 4, 5p_1 + 2p_2 + 3))$$

If we were to solve this system of linear equations, we'd find that either there is no solution which falls within our constraints. The reason is that, for the optimal solution, $p_1 = 0$. Taking this into account, we can then find

$$\arg_p \min(\max(-5p_2 + 5, -3p_2 + 4, 2p_2 + 3))$$

and we note that, as there are three equations but only one variable, only two must equal a constant to be minimized. We could do this by solving all three pairs as individual systems and finding the lowest C, or by noting that $-5p_2+5 > -3p_2 + 4$ for $p_2 < 0.5$ and that, at $p_2 = 0.5$, $-5p_2 + 5 = 2.5 < 4 = 2p_2 + 3$, meaning that all we need to do is set $C = -5p_2 + 5 = 2p_2 + 3$ and see that $p_2 = \frac{2}{7}$ and $C = \frac{25}{7}$.

$$p = \begin{bmatrix} 0 & \frac{2}{7} & \frac{5}{7} \end{bmatrix}$$

And a third example to illustrate this problem even more:

$$A = \begin{bmatrix} 9 & 5 & 9 \\ 7 & 6 & 2 \\ 9 & 0 & 2 \end{bmatrix}$$

$$\arg_p \min(\max(9p_1 + 7p_2 + 9p_3, 5p_1 + 6p_2, 9p_1 + 2p_2 + 2p_3)) =$$

$$\arg_p \min(\max(-2p_2 + 9, 5p_1 + 6p_2, 7p_1 + 2)) =$$

Here, both p_1 and p_3 are zero, meaning p_2 is forced to be 1, giving a solution of 7. It is very important to note that if there are k nonzero p_i's, there are at least k equations in the maximizing group that are equal.

$$p = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$

This effect can be clearly illustrated by these three interactive graphs:
https://www.desmos.com/calculator/w4o5n4k2ec
https://www.desmos.com/calculator/n6mux5pzhl
https://www.desmos.com/calculator/fregr1lpeo

Vary p_2 to see the effect on the three composite equations. Note that we want the minimum valid point, and valid points are on the maximum line and are between the y axis and the vertical line.

The issue here is that we cannot directly solve the system of linear equations without knowing which values are necessarily equal to the maximum C. This means that it is difficult to find the analytic solution, particularly when the matrix is large.

This is what makes the problem suitable for an optimization algorithm.

1.1 Continuous Case

Let's try a more complex model. It could provide more interesting results. Instead of a matrix, let's use a continuously differentiable function $f(x, y) : [0, 1]^2 \to \mathbb{R}$. Here, once again, we pick a probability distribution $p(y) : [0, 1] \to \mathbb{R}^+$ on the rows (now y) and our opponent picks a single x with a maximal value. In other words, we have the problem:

$$\arg_{p(y)} \min(\max_x(\int_0^1 p(y)f(x, y)dy))$$

such that

$$p(y) \geq 0$$

$$\int_0^1 p(y)dy = 1$$

$$0 \leq x \leq 1$$

Graphically, one could think of this as finding a scaling factor for $f(x, y)$ such that we minimize the maximum point on the integral. As an example, take $f(x, y) = x^2 + y^2$. This gives us

$$\arg_{p(y)} \min(\max_x(\int_0^1 p(y)(x^2+y^2)dy)) = \arg_{p(y)} \min(\max_x(\int_0^1 p(y)x^2+p(y)y^2dy)) =$$

$$\arg_{p(y)} \min(\max_x(\int_0^1 p(y)x^2dy + \int_0^1 p(y)y^2dy)) =$$

$$\arg_{p(y)} \min(\max_x(x^2 \int_0^1 p(y)dy + \int_0^1 p(y)y^2dy)) =$$

$$\arg_{p(y)} \min(\max_x(x^2 + \int_0^1 p(y)y^2dy)) =$$

$$\arg_{p(y)} \min(\max_x(x^2) + \max_x(\int_0^1 p(y)y^2dy)) =$$

$$\arg_{p(y)} \min(1 + \int_0^1 p(y)y^2 dy) =$$

$$\arg_{p(y)} \min(\int_0^1 p(y)y^2 dy)$$

and we can see that this gives us $p(y) = \delta(y)$, making $\int_0^1 p(y)y^2 dy = 0$, which is equivalent to just picking $y = 0$ 100% of the time. In other words, the best thing we can do is pick $y = 0$ at which point the opponent picks $x = 1$.

In fact, this generalizes: if $f(x, y) = g(x) + h(y)$,

$$\arg_{p(y)} \min(\max_x(\int_0^1 p(y)f(x, y)dy)) = \arg_{p(y)} \min(\int_0^1 p(y)h(y)dy)$$

which will simply be $p(y) = \delta(y - \arg_i \min(h(i)))$.

However, the nature of this problem makes it difficult to optimize in an algorithmic fashion, and so the rest of the paper will deal with the aforementioned simplified prisoner's dilemma.

2 Algorithm Theory

The optimization algorithm is a non-standard one, as it does not actually move in the direction that minimizes the gradient. The reason is that the calculation to determine the precise direction in which to move that minimizes the gradient is not a trivial one: indeed, determining that would involve a calculation not unlike the one to find the solution to the problem itself.

However, there is an easier way. Given the conditions that $||p||_1 = 1$ and $p_i \geq 0$, we can iterate under this ideal: $p^+ = \frac{p+c}{||p+c||_1}$, essentially adding a small change (c) and then renormalizing. Indeed, any possible change can be reached instantly with this method, except for a change which sets a nonzero element of p to 0, though that can be achieved asymptotically.

Therefore, our task is, given a step size t, to determine the c for each iteration such that, over several iterations, we will inevitably move towards the minimum point with minimal computational complexity so long as t is small enough.

Now, let's consider a generic $c = [c_1, c_2, ..., c_n]$. Given vectors I_i, being the ith row of the identity matrix, we see that $c = \sum_i^n c_i I_i$. Now, this gives us

$$\frac{p+c}{||p+c||_1} = \frac{p+\sum_i^n c_i I_i}{||p+\sum_i^n c_i I_i||_1} = \frac{\left(p+\sum_i^{n-1} c_i I_i\right) + c_n I_n}{||\left(p+\sum_i^{n-1} c_i I_i\right) + c_n I_n||} =$$

$$\frac{\frac{1}{||\left(p+\sum_i^{n-1} c_i I_i\right)||_1} p' + c_n I_n}{||\frac{1}{||\left(p+\sum_i^{n-1} c_i I_i\right)||_1} p' + c_n I_n||} = \frac{p' + ||\left(p+\sum_i^{n-1} c_i I_i\right)||_1 c_n I_n}{||p' + ||\left(p+\sum_i^{n-1} c_i I_i\right)||_1 c_n I_n||} = \frac{p' + c' I_n}{||p' + c' I_n||}$$

where

$$p' = \frac{p+\sum_i^{n-1} c_i I_i}{||p+\sum_i^{n-1} c_i I_i||_1}$$

$$c' = ||\left(p+\sum_i^{n-1} c_i I_i\right)||_1 c_n$$

and we can see that, by induction, any change c is possible through a series of steps where $c = aI_i$ for some scalars a and integers $1 \leq i \leq n$. Moreover, since this is a problem with precisely one minimum value (note that there might be multiple minimum points, just not multiple distinct minimum values), any minimum will do, we know that moving in any direction that decreases the value will move towards a valid minimum. And since we know that we can always move in the minimum direction using a series of orthogonal steps, we need only consider those orthogonal steps, which significantly reduces computational complexity.

In fact, with this in mind, we need only move in the orthogonal direction that

minimizes the value. Eventually, we must converge on a minimum with the greatest possible accuracy that the step size t can allow. Given this step size, we have first to find this:

$$\arg_i \min(\max((p + tI_i)A))$$

which can be accomplished in a single longish line of python code. Now that we have the direction to move in, though, the question remains: how much?

The obvious solution is, of course, letting $p^+ = \frac{p+tI_i}{1+t}$. However, this does not provide optimal convergence speeds. A significantly faster algorithm is $p^+ = \frac{p+cI_i}{1+c}$ where

$$c = \max_j(\max((p + tI_i)A) - (\max((p + tI_j)A)))$$

The algorithm above requires one to set an update rate, t, from which the actual updates to our probability vector, c, are determined. This allows for two degrees of freedom in optimizing our algorithm - we may modulate both t and c.
Towards this latter goal, consider the function

$$g(x) = \frac{1}{q + N^z \tanh \frac{hx}{L}}$$

N and L are the dimension of the input matrix and the number of iterations chosen, respectively, and q, z, h are positive real numbers defined by a user.
$g(x)$ allows us to define a sequence $g(0), \ldots, g(i), \ldots, g(L)$, where $g(i)$ will be used to scale the step $c = \max_j(\max((p + tI_i)A) - (\max((p + tI_j)A)))$.
For small x, $g(x) < 1$, and as $x \to hL$ $g(x) \to \frac{1}{q+N^p}$. Using this function, we augment our update rate: $p^+ = \frac{p+g(x)cI_i}{1+g(x)c}$. Adding this multiple of $g(x)$ significantly increases step sizes taken early in the process, while limiting step sizes taken later. For a proper selection of q, p, h this allows us to operate on much larger matrices than the standard algorithm. See, using the standard algorithm we have no choice but to select an extremely small t, and an extremely large number of iterations. For a matrix A with dimension N, as $N \to \infty$, the expected value of an entry $p_i \to 0$. For larger matrices, the individual entries of p must be more finely tuned.

Another refinement is to subtly increase t through multiplying by a factor $1 + b$ (for a small real number b) if no decrease in $\max(pA)$ is seen after a set number of iterations. This serves to counterbalance the consistent decrease in adjustment size given by $g(x)c$. If the scaling factor shrinks too quickly, we increase t, giving our descent enough momentum to potentially explore new directions of change in p.

3 Implementation and Experimentation

We chose to implement our algorithm in Python (3.8). Full code is provided.[1]

Throughout presentation of our algorithms, we will focus on two 7×7 matrices:

$$A_1 = \begin{bmatrix} 9.89 & 5.49 & 2.81 & 0.77 & 4.45 & 4.73 & 0.49 \\ 1.63 & 1.16 & 6.27 & 8.56 & 6.50 & 9.91 & 4.70 \\ 6.18 & 2.87 & 9.76 & 6.73 & 4.41 & 2.90 & 5.10 \\ 1.12 & 2.27 & 4.79 & 2.43 & 3.90 & 8.20 & 0.75 \\ 9.23 & 2.25 & 7.64 & 1.12 & 6.01 & 4.07 & 8.37 \\ 2.50 & 4.58 & 5.57 & 2.52 & 1.10 & 7.27 & 3.10 \\ 8.26 & 4.52 & 0.94 & 8.87 & 7.42 & 1.22 & 8.56 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 3.75 & 9.51 & 7.32 & 5.99 & 1.56 & 1.56 & 0.58 \\ 8.66 & 6.01 & 7.08 & 0.21 & 9.70 & 8.32 & 2.12 \\ 1.82 & 1.83 & 3.04 & 5.25 & 4.32 & 2.91 & 6.12 \\ 1.40 & 2.92 & 3.66 & 4.56 & 7.85 & 2.00 & 5.14 \\ 5.92 & 0.47 & 6.08 & 1.71 & 0.65 & 9.49 & 9.66 \\ 8.08 & 3.05 & 0.98 & 6.84 & 4.40 & 1.22 & 4.95 \\ 0.34 & 9.09 & 2.59 & 6.63 & 3.12 & 5.20 & 5.47 \end{bmatrix}$$

Both of these matrices were created by selecting entries from a uniform random distribution on the interval $(0, 10)$ - they, therefore, should be expected to have a minimal opponent maximum near to 5. For the sake of demonstration, A_1 was selected for performance considered "typical", whereas A_2 was selected for sub-optimal performance under our naive algorithm.

[1] see appendix sections 1.1.2, 1.1.3, and 1.1.4.

Algorithm 1: Naive Optimization

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, number of iterations L, perturbation size t
Output: Approximation of $\arg_{\mathbf{p}} \min(\max_i(\mathbf{pA})_i)$
Set $\mathbf{p} = [1, \ldots, 1] \in \mathbb{R}^{1 \times N}$
Set $\mathbf{p} = \frac{\mathbf{p}}{||\mathbf{p}||_1}$
for *i = 1, ... , L* **do**
  Set $max = 0$
  Set $c = 0$
  Set $min = \infty$
  for *j = 1, ... , N* **do**
    Set $m = \max(\mathbf{p} + t\mathbf{e}_j^T)$
    if $m < min$ **then**
      Set $min, c = m, j$
    if $m > max$ **then**
      Set $max = m$
  end
  Set $\mathbf{p} = \frac{\mathbf{p}+(max-min)\mathbf{e}_j^T}{||\mathbf{p}+(max-min)\mathbf{e}_j||_1}$
end
Return $\mathbf{p}$

Executing Algorithm One with perturbation size .00001 on A_1 and A_2, we plot $\max\{\mathbf{p}A\}$ as a function of iteration number:

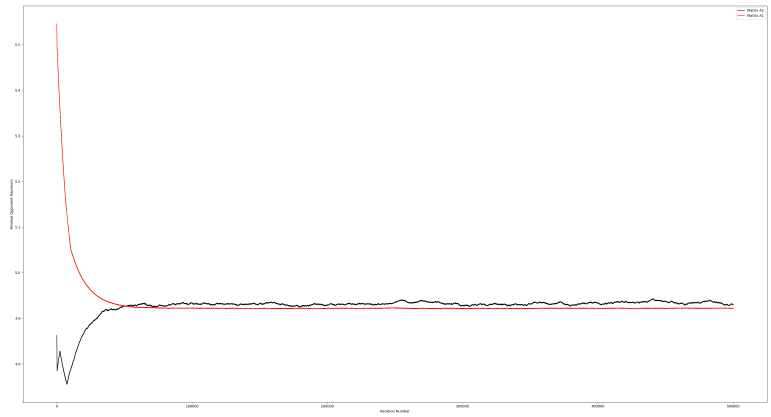

Figure 1: Algorithm 1 Demonstration

For cost matrix A_2, the algorithm "bounces" away from a good value, and settles to a sub-ideal value. This "bouncing" may be counteracted by lowering perturbation size, however the decrease necessary substantially slows execution.

Algorithm Two seeks to counteract this, as described in section 3:

Algorithm 2: Optimization with Scaling Factor

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, number of iterations L, perturbation size t, scaling parameters z, q, h

Output: Approximation of $\arg_{\mathbf{p}} \min(\max_i(\mathbf{pA})_i)$

Set $\mathbf{p} = [1, \ldots, 1] \in \mathbb{R}^{1 \times N}$

Set $\mathbf{p} = \frac{\mathbf{p}}{||\mathbf{p}||_1}$

for i = *1, ... , L* **do**

  Set $max = 0$

  Set $c = 0$

  Set $min = \infty$

  for j = *1, ... , N* **do**

    Set $m = \max(\mathbf{p} + t\mathbf{e}_j^T)$

    if $m < min$ **then**

      Set $min, c = m, j$

    if $m > max$ **then**

      Set $max = m$

  end

  Set $\mathbf{p} = \frac{\mathbf{p}+(max-min)\mathbf{e}_j^T}{||\mathbf{p}+(max-min)\mathbf{e}_j^T||_1} \times \frac{1}{q+N^z \tanh \frac{hi}{L}}$

end

Return $\mathbf{p}$

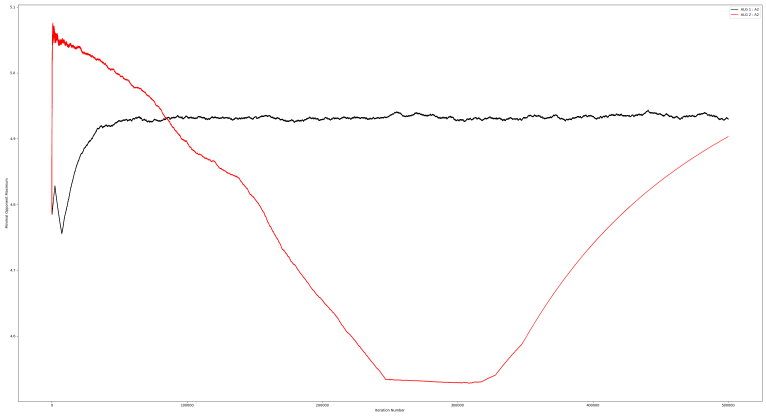

Figure 2: Algorithm 1 (B) and 2 (R) ($z = 1, q = .01, h = 1$), Matrix A_2

Similarly, Algorithm Two provides substantial performance boosts on matrix

A_1 (and all other matrices tested on).

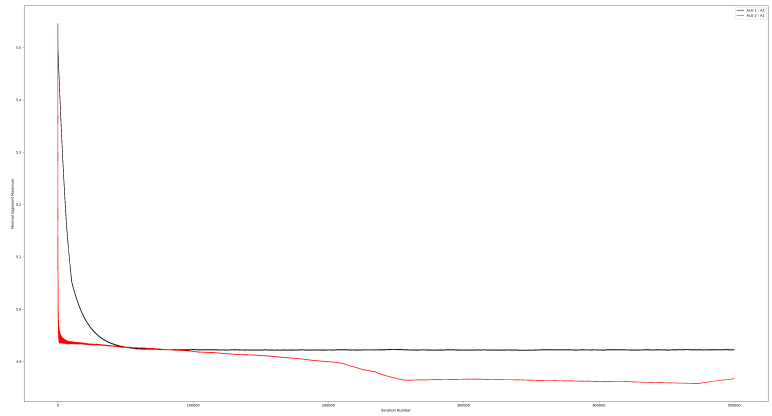

Figure 3: Algorithm 1 (B) and 2 (R) ($z = 1, q = .01, h = 1$), Matrix A_1

Algorithm Two has substantial efficacy in application to high dimensional matrices, computation on which is normally prohibitively expensive. Below, it is shown to converge to an inexact, but reasonable value within 25,000 iterations on a 20×20 matrix with entries drawn from a uniform distribution on the interval $(0, 10)$. Perturbation size $t = .0001$ is used.

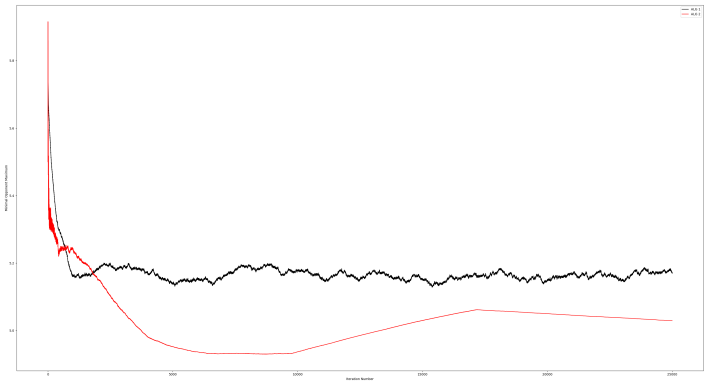

Figure 4: Algorithm 1 (B) and 2 (R) ($z = 1, q = .01, h = 1$), 20×20 random matrix

Algorithm 3: Optimization with Scaling Factor and Perturbation Boosting

Input: $\mathbf{A} \in \mathbb{R}^{n \times n}$, number of iterations L, perturbation size t, scaling parameters z, q, h, perturbation boost b

Output: Approximation of $\arg_{\mathbf{p}} \min(\max_i(\mathbf{pA})_i)$

Set $\mathbf{p} = [1, \ldots, 1] \in \mathbb{R}^{1 \times N}$

Set $\mathbf{p} = \frac{\mathbf{p}}{||\mathbf{p}||_1}$

for *i = 1, ... , L* **do**

  Set $max = 0$

  Set $c = 0$

  Set $min = \infty$

  for *j = 1, ... , N* **do**

    Set $m = \max(\mathbf{p} + t\mathbf{e}_j^T)$

    if $m < min$ **then**

      Set $min, c = m, j$

    if $m > max$ **then**

      Set $max = m$

  end

  Set $\mathbf{p} = \frac{\mathbf{p}+(max-min)\mathbf{e}_j^T}{||\mathbf{p}+(max-min)\mathbf{e}_j^T||_1} \times \frac{1}{q+N^z \tanh \frac{hi}{L}}$

  if ***p*** *has not improved in* $L/100$ *iterations* **then**

    $t = (1 + b) \times t$

end

Return $\mathbf{p}$

Though on our matrices A_1 and A_2 a substantial benefit is not observed from Algorithm Three, there are cases where it speeds the convergence of Algorithm Two substantially via correcting for over-adjustments made in the use of scaling factors:

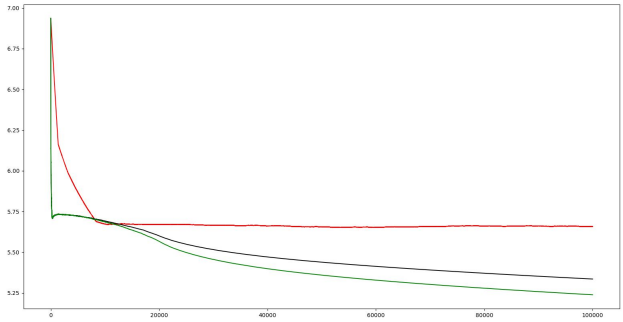

Figure 5: Algorithm 1 (B), 2 (R), 3 (G) ($z = 1, q = .01, h = 1, b = .01$), 10×10 Random Matrix

4 Future Work

In the research, design, and testing of this algorithm, it is clear that there are many unanswered questions. Among them are:

- The Line: What is the strange mark that all algorithms eventually converge to? Why is it so sub-optimal? What is with the strange curve back to it? What does it represent in terms of the matrix's properties?
- Is There a Rhythm: When we zoomed in on the line, we noticed unusual, almost-periodic behavior. What causes this, can we predict it, and is it yet another example of chaos theory messing with our day-to-day lives?
- The Continuous Case: As further elaborated on in our report, we can think of an infinite-dimensional case with an analytic function defined over the one-by-one square. All cases we've seen so far are trivial: do nontrivial ones exist? If so, can we even optimize for them?
- Special Matrices: What happens with Hankel Matrices? Redheffer Matrices? Definite matrices? Frobenius Matrices?
- The N-Player Case: So far, we've only dealt with two players. But what if we made the value matrix N-dimensional, where there are N players making a decision?
- More Matrix Properties and Transformations: Is there any relation between the optimizations for a matrix and its transpose? What about a matrix and its inverse? Or a matrix and its eigenvalue decomposition? Furthermore, what does applying a convolution to the matrix do? If we blur a matrix, does it smooth out the result?
- Measuring Metaparameters: Each of the metaparameters we use has a number of oftentimes unintuitive results. What does each metaparameter do, exactly, what are the "best" values we can set for them in the general case, and what are the limits beyond which they do more harm than good?
- Different Perturbations: What if, instead of using the trivial basis, we used the matrix's own eigenspace?
- Different Problems: Could this algorithm be used for non-prisoner's dilemma-related problems? As an example, could we optimize for, say, the best Gaussian-style Blur formula over a certain area that preserves the elementwise-average or the Frobenius Norm? Are there applications beyond convolutions and matrices?
- The Classic Case: It might not be possible to find the exact solution to the classic prisoner's dilemma through optimization, but could how close could one get?
- Why is the convergence so random? Even tiny changes in matrices can lead to significantly different long-term behavior.
- Why does the initial convergence not reach the minimum? The long-term appears to converge to a suboptimal value, which does not make sense given what we know about the algorithm and associated proof.

5 Appendix

5.1 Python Code

5.1.1 Import and Type Declarations

```
import numpy as np
import matplotlib.pyplot as plt
from typing import TypeVar, Tuple, List
```

5.1.2 Algorithm 1

```
def alg1(cost_array: Array, max_iter: int, t: float):
    #
    # cost_array : NxN array of floating point numbers representing gain
    # max_iter   : the integer at which iteration stops, regardless of convergence
    # t          : floating point perturbation size
    #
    # Computes the ideal probability vector, p, that should be shown to an opponent
    # so as to minimize the maximal gain he could achieve by selecting the optimal
    # column of the input array
    #
    # return: probability vector p, list of opponent optimal values at each iteration

    N = cost_array.shape[0]                                 # dimension of input
    test_vector = np.array([1 for _ in range(N)])/N         # instantiate test vector
    values      = [max((test_vector @ cost_array))]         # instantiate result list

    # store best prob vector, smallest opponent max
    p = test_vector
    op_max = max((test_vector @ cost_array))

    for i in range(max_iter):

        # examine effect of perturbation in each column of test vector
        max_col, minmax, maxmax = 0, float('inf'), 0
        for col in range(N):
            test_vector[col] += t
            m = max(test_vector @ cost_array)
            test_vector[col] -= t
            # update tracker variables
            if m < minmax:
                max_col, minmax = col, m
            if m > maxmax:
                maxmax = m
        test_vector[max_col] += (maxmax - minmax)

        if test_vector[max_col] < 0:
            test_vector[max_col] = 0
        test_vector = test_vector/np.sum(test_vector)

        values.append(max((test_vector @ cost_array)))
        if values[-1] < op_max:
            op_max, p = values[-1], test_vector
    return p, values
```

5.1.3 Algorithm 2

```
def alg2(cost_array: Array, max_iter: int, t: float, z: float, q: float, h: float):
    #
    # cost_array : NxN array of floating point numbers representing gain
    # max_iter   : the integer at which iteration stops, regardless of convergence
    # t          : floating point perturbation size
    # z          : exponent of N in scaling sequence
    # q          : offset in scaling sequence
    # h          : factor in tanh numerator in scaling sequence
    #
    # Computes the ideal probability vector, p, that should be shown to an opponent
    # so as to minimize the maximal gain he could achieve by selecting the optimal
    # column of the input array
    #
    # return: probability vector p, list of opponent optimal values at each iteration

    N = cost_array.shape[0]                              # dimension of input
    test_vector = np.array([1 for _ in range(N)])/N      # instantiate test vector
    values      = [max((test_vector @ cost_array))]      # instantiate result list

    # store best prob vector, smallest opponent max
    p = test_vector
    op_max = max((test_vector @ cost_array))

    # creates a smoothing sequence to scale movements by
    # avoids storing sequence in memory
    scalers = ((q + (N**z)*np.tanh(h*i/(max_iter))) for i in range(max_iter))

    for i in range(max_iter):

        # examine effect of perturbation in each column of test vector
        max_col, minmax, maxmax = 0, float('inf'), 0
        for col in range(N):
            test_vector[col] += t
            m = max(test_vector @ cost_array)
            test_vector[col] -= t
            # update tracker variables
            if m < minmax:
                max_col, minmax = col, m
            if m > maxmax:
                maxmax = m

        test_vector[max_col] += (maxmax - minmax)/next(scalers)
        test_vector = test_vector/np.sum(test_vector)
        if test_vector[max_col] < 0:
            test_vector[max_col] = 0

        values.append(max((test_vector @ cost_array)))
        if values[-1] < op_max:
            op_max, p = values[-1], test_vector
    return p, values
```

5.1.4 Algorithm 3

```
def alg3(cost_array: Array, max_iter: int, t: float, z: float, q: float, h: float, b: float):
    #
    # cost_array : NxN array of floating point numbers representing gain
    # max_iter   : the integer at which iteration stops, regardless of convergence
    # t          : floating point perturbation size
    # z          : exponent of N in scaling sequence
    # q          : offset in scaling sequence
    # h          : factor in tanh numerator in scaling sequence
    # b          : factor to increase t
    #
    # Computes the ideal probability vector, p, that should be shown to an opponent
    # so as to minimize the maximal gain he could achieve by selecting the optimal
    # column of the input array
    #
    # return: probability vector p, list of opponent optimal values at each iteration

    N = cost_array.shape[0]                              # dimension of input
    test_vector = np.array([1 for _ in range(N)])/N      # instantiate test vector
    values      = [max((test_vector @ cost_array))]      # instantiate result list

    # store best prob vector, smallest opponent max
    p = test_vector
    op_max = max((test_vector @ cost_array))

    # creates a smoothing sequence to scale movements by
    # avoids storing sequence in memory
    scalers = ((q + (N**z)*np.tanh(h*i/(max_iter))) for i in range(max_iter))
    j = 0 # used to track updates on t

    for i in range(max_iter):

        # examine effect of perturbation in each column of test vector
        max_col, minmax, maxmax = 0, float('inf'), 0
        for col in range(N):
            test_vector[col] += t
            m = max(test_vector @ cost_array)
            test_vector[col] -= t
            # update tracker variables
            if m < minmax:
                max_col, minmax = col, m
            if m > maxmax:
                maxmax = m

        test_vector[max_col] += (maxmax - minmax)/next(scalers)
        test_vector = test_vector/np.sum(test_vector)
        if test_vector[max_col] < 0:
            test_vector[max_col] = 0

        values.append(max((test_vector @ cost_array)))
        if values[-1] < op_max:
            op_max, p = values[-1], test_vector
            j = i
        if j + max_iter/100 < i:
            j, t = i, t*(1+b)
    return p, values
```

6 References

(1) Mas-Colell, Andreu & Whinston, Michael D. & Green, Jerry R., 1995. "Microeconomic Theory," OUP Catalogue, Oxford University Press, number 9780195102680.
(2) Osborne & Ariel Rubinstein, 1994. "A Course in Game Theory," MIT Press Books, The MIT Press, edition 1, volume 1, number 0262650401, September.