Computing two-person zero sum games at multiple times the speed of Linear Programming solvers

Debtoru Chatterjee

srz238159@sire.iitd.ac.in

Indian Institute of Technology Delhi https://orcid.org/0000-0001-8478-4452

Girish Tiwari

Indian Institute of Technology Bombay

Niladri Chatterjee

Indian Institute of Technology Delhi

Research Article

Keywords:

Posted Date: August 5th, 2025

DOI: https://doi.org/10.21203/rs.3.rs-7278149/v1

Additional Declarations: The authors declare no competing interests.

Computing two-person zero sum games at multiple times the speed of Linear Programming solvers

Debtoru Chatterjee[1] • Girish Tiwari[2] • Niladri Chatterjee[3]

Debtoru Chatterjee, debtoruchatterjee@gmail.com, a Joint Secretary to the Government of India, is a doctoral candidate at the Indian Institute of Technology, School of Interdisciplinary Research, New Delhi, 110016. He created the algorithm described in the paper, its analysis and presentation, and wrote the text.

Girish Tiwari, gtiwari0802@iitb.ac.in is with the Department of Electrical Engineering, Indian Institute of Technology, Mumbai. He coded the algorithm created by the first author.

Niladri Chatterjee, niladri.chatterjee@maths.iitd.ac.in, is a Professor of Computer Science and Statistics in the Department of Mathematics, Indian Institute of Technology (IITD), New Delhi, 110016, India. He examined and edited the paper.

The above authors acknowledge the assistance of Shri Gourav Dhakad, M.Tech (Computer Science), Laxmi Narayan College of Technology, Bhopal, in writing of the code.

Abstract

It has been proposed in the paper 'Reducing the complexity of computing the values of a Nash Equilibrium' by the authors that the optimal row and column strategies and value of a two-person, zero sum game can be more efficiently computed than by linear programming optimization and other existing methods. That paper had described the algorithm for achieving such a result. The present paper presents a computational validation of the claimed result of the algorithm by test of the speed of its performance in comparison with the speed achieved by the GLOP linear programming solver. Evidence is also adduced in favor of the algorithm's performance much exceeding that of other open-source and commercial LP solvers. While the framework of the Colonel Blotto game has been adopted to carry out the performance tests, the results achieved by the algorithm is generalizable and valid for all two-person, zero sum games.

1. Introduction

A Colonel Blotto game is a two person, zero sum game where the players are required to simultaneously distribute limited resources over more than one object of contest. The player allocating greater resources to the object wins it. Formulated by the French Emile Borel in 1921, it was published by Borel and Jean Ville as an application of the Theory of Probability and of the Game of Chance in 1938. While in Borel's game, the players assign symmetrical resources over a finite number of "battlefields", Gross and Wagner in 1950 characterized the game with asymmetrical resources between the two players. They described the game between the two players where the number of battlefields $n = 2$.

Discrete distribution of resource units across 2 battlefields is daily fare in many security scenarios: thus if Hamas has 4000 attack missiles and Israel has to defend 2 equally important cities with 6000 Iron Dome interceptor missiles, with what probabilities should it allocate its missiles at each city for its defence? What would be the optimal allocation of 500 policemen to the defense of 2 establishments ("battlefields") in a disturbed district to each of which target (the establishment) insurgents will also allocate their strength of, say, 300 optimally? And what would be the value of the game, indicating the probability of the attacker winning? If 5000 peacekeepers in a disturbed territory with 3000 insurgents have to divide between the two tasks of neutralizing insurgents via encounters on the one hand and defusing improvised explosive

devices and taking down insurgent supply chains on the other, what will be the optimal allocation of the peacekeepers for these two tasks ("battlefields") to each of which the insurgents will also allocate their numbers optimally?

Colonel Blotto games are, therefore, pivotal to the strategic allocation of resources by two adversaries to two battlefields, as originally formulated by Borel. As in all two-person, zero sum games with large payoff matrices, however, their computation had hitherto been impeded by inefficient algorithms. In the paper, 'Reducing the complexity of computing the value of a Nash equilibrium' presented at the 36th International Conference of Game Theory, the following algorithm was proposed to compute the value of two person, zero sum games, including Colonel Blotto games, much more efficiently than existing algorithms, including Linear Programming solvers:

$$u_i(a^*) = u_i(a_i, a^*_{-i}) \text{ for every action } a_i \text{ of player } i \quad (1)$$

where,

the row player *R* (attacker) adheres to his action a_j^* in the action profile a^*

Column player *C* (defender) represent player *i*; a_i is a set of active strategies *n* in a mixed strategy game such that each element of the set of active strategies $a_i = \{x_1, x_2 \ldots x_n\}$ is played with the probability 1/n; and

a^* is the action profile of each player given by the Nash equilibrium wherein the active strategies of each player are spread over a probability distribution *determined by the Nash equilibrium*.

We called the above the Equal Probabilities Approach (EPA). In that paper, we also defined a Probabilities Summing to 1 Approach (PSA) wherein:

$$u_i(a^*) = u_i(a_i, a^*_{-i}) \text{ for every action } a_i \text{ of player } i \quad (2)$$

where,

the row player *R* (attacker) adheres to his action a_j^* in the action profile a^*;

Column player *C* represents player *i*; a_i is a set of active strategies *n* in a mixed strategy game such that each element of the set of active strategies $a_i = \{x_1, x_2 \ldots x_n\}$ is played with a probability such that $\sum_{i=1}^{n} x_n = 1$; and

a^* is the action profile of each player given by the Nash equilibrium wherein the active strategies of each player are spread over a probability distribution *determined by the Nash equilibrium*.

We proved in the above referred conference paper that our proposed algorithm (that played each active column strategy with equal probability while each active row strategy was played with a probability determined by the Nash equilibrium) was computationally less complex and therefore, more efficient than Linear Programming solvers, the Lemke Howson algorithm and the PSA approach while calculating the same value as the Nash equilibrium. The theoretical proofs of the above claim are repeated here:

Let's denote the EPA approach as σ_{EPA}, where $\sigma_{EPA,i} = 1/n$ for all *i* column strategies.

The mathematical complexity of the EPA approach can be measured by the number of calculations required to compute $\sigma_{EPA,i} = 1/n$. The complexity is thus $O(1)$. Column does not solve a best-response LP or compute a Nash Equilibrium.
In the PSA approach, Column plays each active strategy with a random probability, subject to the constraint that the probabilities sum to 1:

$$\sum_{j=1}^{m} q_j = 1$$

$$q_j \geq 0, \forall j$$

Where q_j is the probability of column player's strategy. The computational complexity of generating random probabilities that satisfy these constraints is $O(m)$.

Thus, the PSA approach requires more calculations than the EPA approach, *especially when the number of strategies is large,* and is, therefore, more computationally complex than EPA.

Linear programming based complexity of computing the optimal column strategy in a two-person, zero sum game has a bound that is $O((n+m)^{3.5} \cdot \log L)$, according to Karmarkar,

where,

m represents the number of rows (strategies for the row player i.e. attacker),

n represents the number of columns (strategies for the column player i.e. defender),

L represents the number of bits required to represent the input data such as the payoffs in the game matrix.

This complexity formula thus indicates that as **m** and **n** increase (more rows and columns due to larger resource units of row and column player), the complexity grows polynomially. However, as **L** increases (more bits required to represents the input data), *the complexity grows exponentially with respect to L.*

Thus for large, two person zero sum games, including large Colonel Blotto games, the complexity of computing the optimal column (defender) strategy, using Karmarkar algorithm, indeed increases significantly both due to the polynomial growth with respect to **m** and **n** and the *exponential* growth with respect to **L**.

The complexity of eliminating dominated strategies from the payoff matrix of two person zero sum games, including Colonel Blotto games, required by our Equal Probabilities algorithm however, grows *quadratically* with respect to the number of rows and columns, but since the number of comparisons depends on both m and n, *the overall complexity grows polynomially, specifically O(m* n*(m + n)).* Although still a significant growth, *our algorithm does not trigger an exponential growth of complexity that linear programming implies as the games grow larger.*

Hence the EPA approach has a lower computational complexity than linear programming for computing the Nash Equilibrium value of a two-person, zero sum game, particularly large games, as its computation of the column strategy is the simplest possible one, requiring no algebraic optimization that LP needs.

The aim of the present paper is to *experimentally validate the above theoretical claim and prove that our algorithm drastically outperforms many competing algorithms, including LP solvers like GLOP, GLPK or Karmarkar, in computing the value of two person zero sum games*. It is true that commercial LP solvers like Gurobi may be faster than GLOP, GLPK or KARMARKAR; however, as (unlike in our EPA algorithm) L would likely play a role in its performance similar to its role in Karmarkar's algorithm or in GLPK, Gurobi's performance would inevitably degrade with the increased size of the inputs. Although the experimental proof is furnished in the framework of the Colonel Blotto game, the implementation of the EPA algorithm in python code and its encapsulation in the python library debtoru_solver makes the algorithm available for use across all two person zero sum games. *Thus the performance results, cited in the Colonel Blotto games context, are generalizable to all two person zero sum games*.

The rest of the paper are as follows: section 2 repeats the description of the Colonel Blotto game contained in the conference paper, to show the functioning of our algorithm viz-a-viz the traditional computation of the Nash equilibrium strategies of both row (attacker) and column (defender) players. Section 3 tabulates the *speed of performance* in seconds of our proposed algorithm as well as those of the Linear Programming solver, GLOP, implemented by *Pyspiel,* and the PSA approach, to facilitate comparison. We have not included the Lemke Howson algorithm, which uses a Pivoting Path method in computing the row-column strategies and game value for comparison as it is not in the race anyway, having been outshone by many LP solvers.

The full python code implementation of our proposed algorithm (encapsulated in debtoru_solver) in solving Colonel Blotto games is given in Appendix 'A', that of PSA in Appendix 'B' and that of an LP implementation (*Pyspiel)* for solving the CB game in Appendix 'C'. Section 4 summarizes the results and indicates the direction of further research.

2. The Colonel Blotto game

The following example is taken from the Colonel Blotto games given by Philip D. Straffin (Straffin 1993) under the name of 'Guerrillas vs. Police'. A model is constructed consisting of g guerrillas, p policemen and two government arsenals which the guerrillas seek to capture and the police needs to defend. The guerrillas win if they capture any one of the two arsenals. (They can thus increase their arms to sustain their conflict with government forces). The police, however, win only if they are able to successfully defend both arsenals.

The guerrillas win if $g > p$. The police win when $p \geq 2g$ as they need to defend each of the two arsenals with a force of at least g. The outcomes when $g \leq p < 2g$ are examined below.

Assuming $g = 4$ and $p = 4$, Straffin depicts the following payoffs to the guerrillas (1 for a win 0 for a loss):

Table 1: Symmetric mixed strategy Colonel Blotto game

	4 Police			
4 Guerrillas		4-0	3-1	2-2
	4-0	½	1	1
	3-1	1	½	1
	2-2	1	1	0

The guerrillas can opt to divide 4-0, 3-1, 2-2, its three active strategies. Police can similarly divide 4-0, 3-1 and 2-2 to protect the arsenal(s). The payoffs to the guerrillas are calculated as follows:

Case 1: Guerrillas divide 4-0 and police divide 4-0. The expected payoff is ½, since the guerrillas attack the defended arsenal with probability 0.5 and lose. But also they attack the undefended arsenal with a probability of 0.5 and win.

Case 2: Guerrillas divide 4-0 and police divide 3-1. Here too the expected payoff for guerillas is 1. They win both the arsenals each of which they attack with a probability of ½.

Case 3: Guerrillas divide 4-0 and police divide 2-2. The guerrillas get a payoff of 1. They win both the arsenals each of which they attack with a probability of ½.

Case 4: Guerrillas divide 3-1 and the police divide 4-0. The guerrillas get a payoff of 1. They win when they attack, with probability of ½, the undefended arsenal with a party of 3 guerrillas and when they attack, with a probability of ½, the undefended arsenal with a party of 1 guerrilla.

Case 5: Guerrillas divide 3-1 and the police divide 3-1. The guerrillas get a payoff of ½. They win when the party of 3 guerillas attacks the arsenal defended by 1 policeman with a probability of ½.

Case 6: Guerrillas divide 3-1 and police divide 2-2. The guerrillas get a payoff of 1. They win when they attack each arsenal with a probability of ½.

Case 7: Guerrillas divide 2-2 and police divide 4-0. The guerrillas get a payoff of 1. Each party of 2 guerrillas wins by attacking the undefended arsenal with a probability of ½.

Case 8: Guerrillas divide 2-2 and police divide 3-1. The guerrillas get a payoff of 1. Each party of 2 guerrillas wins by attacking the arsenal defended by 1 policeman.

Case 9: Guerrillas divide 2-2 and police divide 2-2. The guerrillas get a payoff of 0. Each party of 2 guerrillas is defeated by 2 policemen at each arsenal.

The value of the game is calculated by multiplying each payoff by the Nash equilibrium-determined probability with which the guerrillas and police will play their relevant row and column strategy respectively and then summing the products. In this Colonel Blotto game, the value of the game indicates the probability with which the guerrillas would win. For the above game, the strategies (resource distributions across battlefields) of each player, the Nash equilibrium-determined probabilities with which each player will play each of his or her strategies, and the value of the game are given below:

```python
# 4 Guerillas and 4 Policemen
import nashpy as nash
import numpy as np

# Define the game
A = np.array([[0.5, 1, 1],
              [1, 0.5, 1],
              [1, 1, 0]
              ])
rps = nash.Game(A)

# Find Nash equilibria
eqs = rps.support_enumeration()

# Extract and print the first equilibrium
row_player_strategy, column_player_strategy = next(eqs)
print("Row Player Strategy:", row_player_strategy)
print("Column Player Strategy:", column_player_strategy)

# Calculate the value of the game
value_of_game = row_player_strategy.dot(A).dot(column_player_strategy)

print("Value of the Game:", value_of_game)
```

```
Row Player Strategy: [0.4 0.4 0.2]
Column Player Strategy: [0.4 0.4 0.2]
Value of the Game: 0.8000000000000002
```

Fig.1: Nash Equilibrium Analysis for a 3x3 Game Matrix: 4 Guerillas vs. 4 Policemen

Or

Computation of Nash Equilibrium and Game Value in a 4 Guerillas vs. 4 Policemen Strategic Scenario

Nash equilibrium computes a probability of 0.4, 0.4 and 0.2 to each of the guerrillas' 3 strategies. *It also computes a probability of 0.4, 0.4 and 0.2 to each of the 3 police strategies* and computes the value of the game as 0.80.

As per the proposition, the police (the defender) ought to play each of its *active* strategies with a probability of 1/n, where n is the number of active strategies. Hence, in the above game, Police ought to play each of its 3 active strategies with a probability of 0.33 (rounded of to 0.34 for any one of the three strategies. The number of decimal places may be extended without affecting the result computed to any significant decimal place). Using this proposition, the value of the game is recalculated as follows, retaining the payoffs to the guerrillas and the Nash equilibrium-determined probabilities with which the latter (guerrillas) plays each of its strategies:

Table 2: Expected Payoff Components and Total Game Value Calculation

Or

Breakdown of Expected Payoff Calculation for Nash Equilibrium in 4 Guerillas vs. 4 Policemen Game

0.5 x 0.4 x 0.33	=	0.066
1 x 0.4 x 0.33	=	0.132
1 x 0.4 x 0.34	=	0.136
1 x 0.4 x 0.33	=	0.132

0.5 x 0.4 x 0.33	=	0.066
1 x 0.4 x 0.34	=	0.136
1 x 0.2 x 0.33	=	0.066
1 x 0.2 x 0.33	=	0.066
Total	**=**	**0.8**

It is thus seen that our proposed algorithm dispenses with the requirement of the Police to separately compute the Nash-equilibrium probabilities of its strategies and yet achieves the same value of the game while the Guerilla adheres to his Nash-equilibrium strategy.

We now demonstrate that the proposed algorithm is robust to a larger Colonel Blotto game involving dominating and dominated strategies Assuming $g = 8$ and $p = 9$, the payoffs to the guerrillas are depicted in the following table:

Table 3 Asymmetric mixed strategy Colonel Blotto game

	9 Police					
8 Guerrillas		9-0	8-1	7-2	6-3	5-4
	8-0	½	½	1	1	1
	7-1	1	½	½	1	1
	6-2	1	1	½	½	1
	5-3	1	1	1	½	½
	4-4	1	1	1	1	0

Inputting the payoffs to the guerrilla in the payoffs matrix of the *Nashpy* program, the probabilities attached to the different strategies of the guerrillas and police and the value of the game are computed as follows:

```python
# 8 Guerillas and 9 Policemen
import nashpy as nash
import numpy as np

# Define the game
A = np.array([[0.5, 0.5, 1, 1, 1],
              [1, 0.5, 0.5, 1, 1],
              [1, 1, 0.5, 0.5, 1],
              [1, 1, 1, 0.5, 0.5],
              [1, 1, 1, 1, 0]])
rps = nash.Game(A)

# Find Nash equilibria
eqs = rps.support_enumeration()

# Extract and print the first equilibrium
row_player_strategy, column_player_strategy = next(eqs)
print("Row Player Strategy:", row_player_strategy)
print("Column Player Strategy:", column_player_strategy)

# Calculate the value of the game
value_of_game = row_player_strategy.dot(A).dot(column_player_strategy)

print("Value of the Game:", value_of_game)
```

```
Row Player Strategy: [0.4 0.  0.4 0.  0.2]
Column Player Strategy: [0.  0.4 0.  0.4 0.2]
Value of the Game: 0.8000000000000002
```

Fig.2: Nash Equilibrium and Game Value Computation for 5x5 Matrix: 8 Guerillas vs. 9 Policemen Scenario

Or

Strategic Game Simulation: Nash Equilibrium for 8 Guerillas and 9 Policemen Using Payoff Matrix

After crossing out dominated strategies (implemented in our python code in Appendix 'A' for eliminating weakly dominated strategies) it is seen that the police are left with 3 *active* strategies and the guerrillas are also left with 3 active strategies. The resultant payoff matrix exactly represents the payoff matrix depicted in Table 1. Applying the probability of 1/n to each of n *active* strategies of police and retaining the Nash equilibrium-determined probabilities attached to the strategies of the guerrillas, the same value of the game of 0.80 is achieved.

The table below is a small sample of Colonel Blotto games, both symmetric and asymmetric, and including games with and without dominating strategies. The value of the game achieved by the Nash solution and the proposed algorithm are the same in each case, demonstrating the robustness of the algorithm:

Table 4: Value of Colonel Blotto game

	Proposed algorithm	**Nash solution**
9 Policemen and 8 Guerillas	0.80	0.80
5 Policemen and 5 Guerillas	0.83	0.83
4 Policemen and 4 Guerillas	0.80	0.80
6 Policemen and 5 Guerillas	0.66	0.66
8 Policemen and 8 Guerillas	0.88	0.88
11 Policemen and 10 Guerillas	0.83	0.83
7 Policemen and 6 Guerillas	0.75	0.75

7 Policemen and 7 Guerillas	0.875	0.875
6 Policemen and 6 Guerillas	0.857	0.857
9 Policemen and 9 Guerillas	0.90	0.90
10 Policemen and 9 Guerillas	0.80	0.80

3. Computation of CB game values: A comparison of performance of algorithms

As already mentioned above, for the full python code implementation of our proposed algorithm (encapsulated in debtoru_solver) in solving Colonel Blotto games with two battlefields, the reader is referred to Appendix 'A'. Appendix 'B' shows the PSA implementation, using a randomizer, for computing the value of such Colonel Blotto games. An LP (GLOP) implementation (*Pyspiel)* for solving the CB game is at Appendix 'C'. The three algorithms have been nested within the python code for the game, with a function to also calculate the time taken for computing the values of the Nash Equilibrium in each game (including the optimal row-column strategies). The results for 50 guerrillas attacking 70 policemen tasked with defending two arsenals are also indicated as examples. *They show our algorithm computing the same value as the Nash Equilibrium at a small fraction of the time taken by linear programming solver of Pyspiel i.e. GLOP (Google Linear Optimization Package) and the PSA approach.*

For running the code, the GPU selected is T4. The code is flexible to include any number of battlefields as one of its inputs.

A table indicating the performance metrics of our algorithm vis-à-vis the other algorithms across different samples of police-guerrilla numbers (row-column resources) is given below:

Table 5: Scalability and Performance Evaluation of LP Solver, PSA Approach, and Debtoru_Solver in Computing Game Values
Or
Performance Comparison of Game Solving Algorithms Across Varying Input Sizes in Guerilla-Policemen Strategic Games
Or
Algorithm Efficiency and Game Value Analysis for Large-Scale Guerilla-Policemen Games

Input sizes	**Algorithm**	**Value of Game**	**Average Time to compute**	**debtoru_solver speed vs LP**	**debtoru_solver speed vs PSA**
50 Guerillas and 70 Policemen	LP solver	0.667	0.0151	5.03 times faster	1.53 times faster
	PSA approach	0.667	0.0046		
	debtoru_solver	0.667	0.0030		
80 Guerillas and 110 Policemen	LP solver	0.667	0.038172	11.8 times faster	1.09 times faster
	PSA approach	0.667	0.003546		
	debtoru_solver	0.667	0.003225		
100 Guerillas and 140 Policemen	LP solver	0.667	0.142467	40.32 times faster	1.07 times faster
	PSA approach	0.667	0.003806		
	debtoru_solver	0.667	0.003533		
180 Guerillas and	LP solver	0.5	0.501703	883.27 times faster	1.01 times faster
	PSA approach	0.5	0.000577		

290 Policemen	debtoru_solver	0.5	0.000568		
586 Guerillas and 792 Policemen	LP solver	0.667	22.07438	1410.68 times faster	1.08 times faster
	PSA approach	0.667	0.016929		
	debtoru_solver	0.667	0.015648		
923 Guerillas and 1418 Policemen	LP solver	0.5	169.28742	265757.33 times faster	1.21 times faster
	PSA approach	0.5	0.000776		
	debtoru_solver	0.5	0.000637		
1810 Guerillas and 2999 Policemen	LP solver	0.5	2929.15	4131382.22 times faster	1.13 times faster
	PSA approach	0.5	0.000805		
	debtoru_solver	0.5	0.000709		
3052 Guerillas and 5001 Policemen	LP solver	0.5	Not competitive	times faster	1.05 times faster
	PSA approach	0.5	0.000826		
	debtoru_solver	0.5	0.000785		

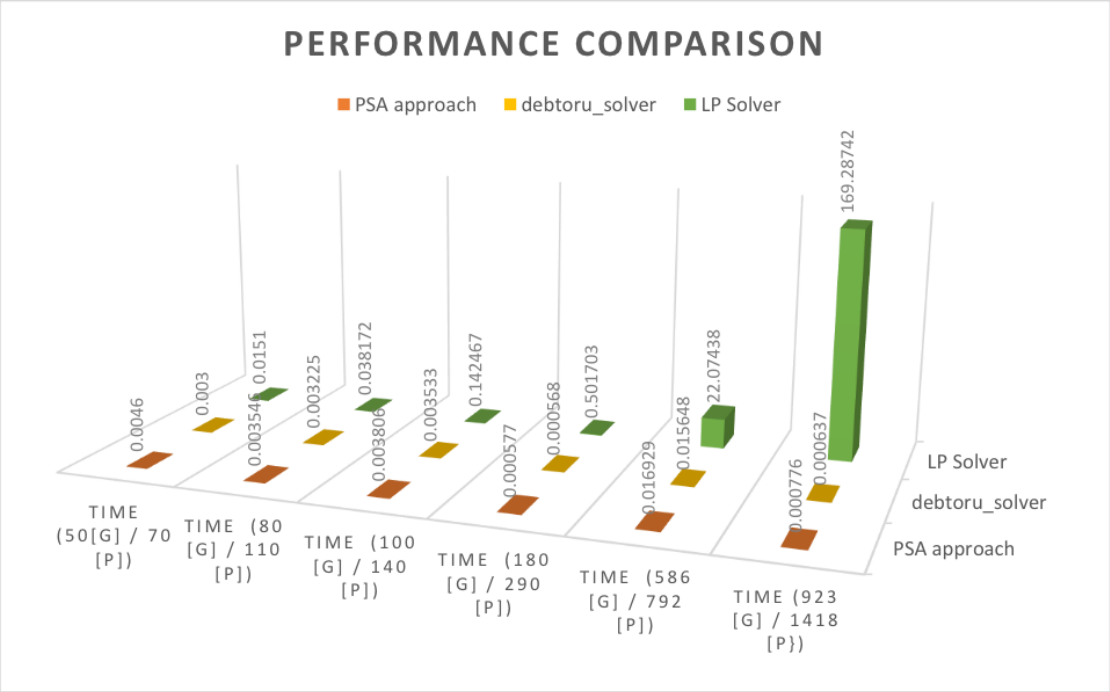

Fig.3: Execution Time Comparison of PSA, Debtoru, and LP Solvers Across Varying Game Sizes

Or

Scalability of Game-Solving Algorithms: PSA vs Debtoru vs LP Solver

4. Discussion

As may be seen from the table, debtoru_solver, implementing our equal probabilities algorithm outclasses the LP solver used by *Pyspiel*, namely GLOP. Yet GLOP has no exponent in a theoretical complexity formula, unlike Karmarkar. GLPK resembles GLOP as it uses the simplex method (with unpredictable iteration counts) and its performance can therefore be considered

comparable to GLOP. On the other hand, GLOP, though based on simplex, is often faster than Karmarkar, especially in zero-sum matrix games. Though Karmarkar has a better worst case complexity than GLOP, GLOP may be actually faster than Karmarkar due to low overhead (avoidance of matrix factorization), lesser number of simplex pivots, warm start by the reuse of previous LP solutions and avoidance of precision-tuning overheads required by Karmarkar.

While the performance of debtoru_solver exceeded that of the PSA approach by between 1% and 21% in the sample, the authors have already proved the superiority of the EPA approach (implemented by debtoru_solver) over the PSA approach on other counts as well in the conference paper "Reducing the complexity of computing the values of a Nash Equilibrium", namely: maximum entropy, robustness to mistakes (Selten's Trembling Hands), minimax regret and robustness to incomplete information (Aumann)

5. Conclusion and further research

Although Gurobi has been reported to be twice as fast as GLPK and sometimes upto 10,000 times faster than open source solvers, we have already seen in our experiment that debtoru_solver outperformed GLOP (which is comparable in performance to GLPK) by over 4131382 times in just a mid-sized zero sum matrix game. According to Netlib LP benchmarks, Gurobi is between 10 to 1000 times faster than GLOP, depending on instance size. This suggests that debtoru_solver, implementing the EPA approach, is probably far more efficient than Gurobi as debtoru_solver has outperformed GLOP by a far bigger margin than has Gurobi, even in our small sample of cases. However, as game structures could be widely divergent in their characteristics, a direct comparison of the two on the same two-person, zero sum games would tender firmer evidence of their respective capabilities.

Appendix

Appendix-A

```python
# New Approach

# Date Jul 02, 2025
# Colonel Blotto Game for 'n' Battlefields, displaying Row and Column Strategies in Wrapped Format
# logic for debtoru_solver as per paper, uniform column probabilities for undominated strategies

import numpy as np
import pyspiel
from open_spiel.python.algorithms import lp_solver
import textwrap
import time

from itertools import product
from open_spiel.python.algorithms.lp_solver import LinearProgram, OBJ_MAX, CONS_TYPE_GEQ, CONS_TYPE_EQ
import cvxopt

def extract_row_payoff_matrix(game):
    return np.array(game.row_utilities())

def weakly_dominated_rows(matrix):
    keep = [1] * matrix.shape[0]
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[0]):
            if i == j:
                continue
            if np.all(matrix[i, :] <= matrix[j, :]) and np.any(matrix[i, :] < matrix[j, :]):
                keep[i] = 0
                break
    return keep

def weakly_dominated_columns(matrix):
    keep = [1] * matrix.shape[1]
    for j in range(matrix.shape[1]):
        for k in range(matrix.shape[1]):
            if j == k:
                continue
```

```
            if np.all(matrix[:, j] >= matrix[:, k]) and np.any(matrix[:, j] > matrix[:, k]):
                keep[j] = 0
                break
    return keep

def ieds_zero_sum_weak_flags(matrix):
    """
    Iterated Elimination of Weakly Dominated Strategies (IEDS) for 2-player zero-sum games.

    Args:
        matrix (np.ndarray): Row player's payoff matrix (m x n)

    Returns:
        reduced_matrix (np.ndarray): Matrix after IEDS
        row_flag (List[int]): 0 if row was eliminated, 1 if retained
        col_flag (List[int]): 0 if column was eliminated, 1 if retained
    """
    # print("\nMatrix input inside IEDS")
    # print(matrix)
    A = matrix.copy()
    orig_rows, orig_cols = matrix.shape
    row_flag = [1] * orig_rows
    col_flag = [1] * orig_cols

    active_rows = list(range(orig_rows))
    active_cols = list(range(orig_cols))

    changed = True
    while changed:
        changed = False

        # Eliminate weakly dominated rows
        current_matrix = A[np.ix_(active_rows, active_cols)]
        row_keep = weakly_dominated_rows(current_matrix)
        if sum(row_keep) < len(row_keep):
            new_active_rows = [r for r, keep in zip(active_rows, row_keep) if keep == 1]
            for r in active_rows:
                if r not in new_active_rows:
                    row_flag[r] = 0
            active_rows = new_active_rows
            changed = True

        # Eliminate weakly dominated columns
        current_matrix = A[np.ix_(active_rows, active_cols)]
        col_keep = weakly_dominated_columns(current_matrix)
        if sum(col_keep) < len(col_keep):
            new_active_cols = [c for c, keep in zip(active_cols, col_keep) if keep == 1]
            for c in active_cols:
                if c not in new_active_cols:
                    col_flag[c] = 0
            active_cols = new_active_cols
            changed = True

    reduced_matrix = A[np.ix_(active_rows, active_cols)]
    return reduced_matrix, row_flag, col_flag

def compute_column_player_value(p0_sol, p1_sol, payoff_matrix):
    """
    Computes the expected value of the game for the row player.

    Parameters:
        p0_sol (np.ndarray): Row player's strategy (size: [num_rows])
        p1_sol (np.ndarray): Column player's strategy (size: [num_cols])
        payoff_matrix (np.ndarray): Row player's payoff matrix (shape: [num_rows x num_cols])
```

```
101
102     Returns:
103         float: Expected value of the game for the row player
104     """
105     p0 = np.array(p0_sol).reshape(-1, 1)  # Column vector
106     p1 = np.array(p1_sol).reshape(1, -1)  # Row vector
107     return float(np.sum(p0 * payoff_matrix * p1))
108
109 def expand_strategy(short_strategy, flag):
110     """
111     Expands a strategy vector to full size using a flag vector.
112
113     Parameters:
114         short_strategy (List[float]): Strategy values for undominated strategies.
115         flag (List[int]): Binary flag (1 = keep, 0 = dominated).
116
117     Returns:
118         np.ndarray: Full strategy with 0s in dominated positions.
119     """
120     full_strategy = np.zeros(len(flag))
121     idx = 0
122     for i, keep in enumerate(flag):
123         if keep == 1:
124             full_strategy[i] = short_strategy[idx]
125             idx += 1
126     return full_strategy
127
128 def solve_with_debtoru_solver(game, row_flag, dominance_mask):
129     assert isinstance(game, pyspiel.MatrixGame)
130     assert game.get_type().information == pyspiel.GameType.Information.ONE_SHOT
131     assert game.get_type().utility == pyspiel.GameType.Utility.ZERO_SUM
132
133     num_rows = game.num_rows()
134     num_cols = game.num_cols()
135     cvxopt.solvers.options["show_progress"] = False
```

```
136
137     # ----------------------------
138     # Row player's LP (OpenSpiel version)
139     # ----------------------------
140     lp0 = LinearProgram(OBJ_MAX)
141     for r in range(num_rows):
142         lp0.add_or_reuse_variable(r, lb=0)
143     lp0.add_or_reuse_variable(num_rows)  # V
144     lp0.set_obj_coeff(num_rows, 1.0)     # max V
145
146     for c in range(num_cols):
147         lp0.add_or_reuse_constraint(c, CONS_TYPE_GEQ)
148         for r in range(num_rows):
149             lp0.set_cons_coeff(c, r, game.player_utility(0, r, c))
150         lp0.set_cons_coeff(c, num_rows, -1.0)  # -V >= 0
151
152     lp0.add_or_reuse_constraint(num_cols + 1, CONS_TYPE_EQ)
153     lp0.set_cons_rhs(num_cols + 1, 1.0)
154     for r in range(num_rows):
155         lp0.set_cons_coeff(num_cols + 1, r, 1.0)
156
157     sol = lp0.solve()
158     p0_sol = sol[:-1]      # row strategy
159     p0_sol_val = sol[-1]   # game value for row player
160
161     # ----------------------------
162     # Column strategy (modified)
163     # ----------------------------
164     matrix = extract_row_payoff_matrix(game)
165     # reduced_matrix, row_flag, dominance_mask = ieds_zero_sum_weak_flags(matrix)
166     # print("\nRow Mask\n",row_flag)
167     # print("\nColumn Mask\n",dominance_mask)
168     num_undominated = sum(dominance_mask)
```

```
169
170     if num_undominated == 0:
171         raise ValueError("All column strategies are dominated — no valid strategy remains.")
172
173     p1_sol = np.zeros(num_cols)
174     uniform_prob = 1.0 / num_undominated
175     for i in range(num_cols):
176         p1_sol[i] = uniform_prob
177
178     # print("\np0_sol\n", p0_sol)
179     # print("\np1_sol\n", p1_sol)
180     # ----------------------------
181     # Return as in original OpenSpiel
182     # ----------------------------
183     p1_sol_val = compute_column_player_value(p0_sol, p1_sol, matrix)  # skipped LP for column player
184
185     p0_sol_full = expand_strategy(p0_sol, row_flag)
186     p1_sol_full = expand_strategy(p1_sol, dominance_mask )
187
188     return p0_sol_full, p1_sol_full, p0_sol_val, p1_sol_val
189
190 def generate_strategies(num_agents, num_arsenals):
191     """
192     Generate all unique descending allocations (canonical form) of num_agents across num_arsenals.
193     Each allocation is a tuple of non-negative integers that sum to num_agents,
194     sorted in descending order. No repeated permutations.
195     The result is sorted in reverse lexicographical order.
196     """
197     strategies = set()
```

```
198
199     # Generate all possible combinations
200     for combo in product(range(num_agents + 1), repeat=num_arsenals):
201         if sum(combo) == num_agents:
202             strategies.add(tuple(sorted(combo, reverse=True)))  # enforce canonical form
203
204     return sorted(strategies, reverse=True)
205
206 def compute_payoff_matrix(num_guerrillas, num_police, num_arsenals):
207     g_strategies = generate_strategies(num_guerrillas, num_arsenals)
208     p_strategies = generate_strategies(num_police, num_arsenals)
209     print("\nRow Player (Guerrilla) Allocation:\n", textwrap.fill(str(g_strategies), width=100))
210     print("\nColumn Player (Police) Allocation:\n", textwrap.fill(str(p_strategies), width=100))
211     num_g = len(g_strategies)
212     num_p = len(p_strategies)
213     payoff_matrix = np.zeros((num_g, num_p))
214     win_count = 0
215
216     for i, g_strat in enumerate(g_strategies):
217         for j, p_strat in enumerate(p_strategies):
218
219             # Compare each guerrilla (row) group against each police (column) group
220             comparisons = [(g, p) for g in g_strat for p in p_strat]
221             win_count = sum(1 for g, p in comparisons if g > p)
222
223             payoff = round(win_count / num_arsenals, 2)
224             if payoff > 1.0:
225                 payoff = 1.0
226
227             payoff_matrix[i, j] = payoff
```

```
    return payoff_matrix

def compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals):
    """
    Compute and display the Nash equilibrium strategies and game value for the Colonel Blotto game.
    """
    payoff_matrix = compute_payoff_matrix(num_guerrillas, num_police, num_arsenals)
    print(f"\nPayoff Matrix for {num_guerrillas} Guerrillas and {num_police} Police:")
    print(payoff_matrix)

    reduced_matrix, row_flag, dominance_mask = ieds_zero_sum_weak_flags(payoff_matrix)
    # print("\nreduced_matrix\n",reduced_matrix)
    # print("\nRow Mask\n",row_flag)
    # print("\nColumn Mask\n",dominance_mask)



    # Create OpenSpiel matrix game
    row_utilities = reduced_matrix.tolist()
    col_utilities = [[-cell for cell in row] for row in row_utilities]  # zero-sum version
    game = pyspiel.create_matrix_game(row_utilities, col_utilities)

    start_time = time.time()
    # Solve using LP solver
    row_strategy, col_strategy, game_value_r, game_value_c = solve_with_debtoru_solver(game, row_flag, dominance_mask)
    end_time = time.time()
    elapsed = end_time - start_time

    # Output strategies and game value
    row_strategy_rounded = [round(p, 3) for p in row_strategy]
    col_strategy_rounded = [round(p, 3) for p in col_strategy]
    game_value_r_rounded = round(game_value_r, 3)
    game_value_c_rounded = round(game_value_c, 3)

    print("\nNash Equilibrium:")
    formatted_row_strategy = [f"{float(x):.3f}" for x in row_strategy]
    print("\nRow Player (Guerrilla) Strategy:\n", textwrap.fill(str(formatted_row_strategy), width=100))

    # print("\nColumn Player (Police) Strategy:\n", textwrap.fill(str(col_strategy_rounded), width=100))
    formatted_col_strategy = [f"{float(x):.3f}" for x in col_strategy]
    print("\nColumn Player (Police) Strategy:\n", textwrap.fill(str(formatted_col_strategy), width=100))

    # print("\nValue of the Game (According to Row strategies of LP_Solver):", game_value_r)
    print("\nValue of the Game (According to Row Prob from LP_Solver and uniform column prob):", game_value_c)
    print(f"\nExecution time of debtoru_solver: {elapsed:.6f} seconds")
if __name__ == "__main__":
    num_guerrillas = int(input("Enter the number of guerrillas (row): "))
    num_police = int(input("Enter the number of police (column): "))
    num_arsenals = int(input("Enter the number of arsenals (battlefield): "))
    #payoff_matrix_test = compute_payoff_matrix(num_guerrillas, num_police)
    #print("Payoff Matrix for {} Guerrillas and {} Police:".format(num_guerrillas, num_police))
    #print(payoff_matrix_test)
    compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals)
```

```
Enter the number of guerrillas (row): 50
Enter the number of police (column): 70
Enter the number of arsenals (battlefield): 2

Row Player (Guerrilla) Allocation:
 [(50, 0), (49, 1), (48, 2), (47, 3), (46, 4), (45, 5), (44, 6), (43, 7), (42, 8), (41, 9), (40, 10),
(39, 11), (38, 12), (37, 13), (36, 14), (35, 15), (34, 16), (33, 17), (32, 18), (31, 19), (30, 20),
(29, 21), (28, 22), (27, 23), (26, 24), (25, 25)]

Column Player (Police) Allocation:
 [(70, 0), (69, 1), (68, 2), (67, 3), (66, 4), (65, 5), (64, 6), (63, 7), (62, 8), (61, 9), (60, 10),
(59, 11), (58, 12), (57, 13), (56, 14), (55, 15), (54, 16), (53, 17), (52, 18), (51, 19), (50, 20),
(49, 21), (48, 22), (47, 23), (46, 24), (45, 25), (44, 26), (43, 27), (42, 28), (41, 29), (40, 30),
(39, 31), (38, 32), (37, 33), (36, 34), (35, 35)]
```

```
Payoff Matrix for 50 Guerrillas and 70 Police:
[[0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
```

```
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]]
```

```
Nash Equilibrium:

Row Player (Guerrilla) Strategy:
 ['0.667', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.067',
'0.067', '0.067', '0.067', '0.067']

Column Player (Police) Strategy:
 ['0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.125', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.125', '0.125', '0.125', '0.125',
'0.125', '0.125', '0.125']

Value of the Game (According to Row Prob from LP_Solver and uniform column prob): 0.6666666816837485

Execution time of debtoru_solver: 0.002876 seconds
```

Appendix-B

```
422# New Approach

# Date Jul 02, 2025
# Colonel Blotto Game for 'n' Battlefields, displaying Row and Column Strategies in Wrapped Format
# logic for  PSA=1 for undominated strategies
import numpy as np
import pyspiel
from open_spiel.python.algorithms import lp_solver
import textwrap
import time

from itertools import product
from open_spiel.python.algorithms.lp_solver import LinearProgram, OBJ_MAX, CONS_TYPE_GEQ, CONS_TYPE_EQ
import cvxopt

def extract_row_payoff_matrix(game):
    return np.array(game.row_utilities())

def weakly_dominated_rows(matrix):
    keep = [1] * matrix.shape[0]
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[0]):
            if i == j:
                continue
            if np.all(matrix[i, :] <= matrix[j, :]) and np.any(matrix[i, :] < matrix[j, :]):
                keep[i] = 0
                break
    return keep

def weakly_dominated_columns(matrix):
    keep = [1] * matrix.shape[1]
    for j in range(matrix.shape[1]):
        for k in range(matrix.shape[1]):
            if j == k:
                continue
            if np.all(matrix[:, j] >= matrix[:, k]) and np.any(matrix[:, j] > matrix[:, k]):
```

```
                keep[j] = 0
                break
    return keep

def ieds_zero_sum_weak_flags(matrix):
    """
    Iterated Elimination of Weakly Dominated Strategies (IEDS) for 2-player zero-sum games.

    Args:
        matrix (np.ndarray): Row player's payoff matrix (m x n)

    Returns:
        reduced_matrix (np.ndarray): Matrix after IEDS
        row_flag (List[int]): 0 if row was eliminated, 1 if retained
        col_flag (List[int]): 0 if column was eliminated, 1 if retained
    """
    # print("\nMatrix input inside IEDS")
    # print(matrix)
    A = matrix.copy()
    orig_rows, orig_cols = matrix.shape
    row_flag = [1] * orig_rows
    col_flag = [1] * orig_cols

    active_rows = list(range(orig_rows))
    active_cols = list(range(orig_cols))

    changed = True
    while changed:
        changed = False

        # Eliminate weakly dominated rows
        current_matrix = A[np.ix_(active_rows, active_cols)]
        row_keep = weakly_dominated_rows(current_matrix)
        if sum(row_keep) < len(row_keep):
            new_active_rows = [r for r, keep in zip(active_rows, row_keep) if keep == 1]
```

```
            for r in active_rows:
                if r not in new_active_rows:
                    row_flag[r] = 0
            active_rows = new_active_rows
            changed = True

        # Eliminate weakly dominated columns
        current_matrix = A[np.ix_(active_rows, active_cols)]
        col_keep = weakly_dominated_columns(current_matrix)
        if sum(col_keep) < len(col_keep):
            new_active_cols = [c for c, keep in zip(active_cols, col_keep) if keep == 1]
            for c in active_cols:
                if c not in new_active_cols:
                    col_flag[c] = 0
            active_cols = new_active_cols
            changed = True

    reduced_matrix = A[np.ix_(active_rows, active_cols)]
    return reduced_matrix, row_flag, col_flag

def compute_column_player_value(p0_sol, p1_sol, payoff_matrix):
    """
    Computes the expected value of the game for the row player.

    Parameters:
        p0_sol (np.ndarray): Row player's strategy (size: [num_rows])
        p1_sol (np.ndarray): Column player's strategy (size: [num_cols])
        payoff_matrix (np.ndarray): Row player's payoff matrix (shape: [num_rows x num_cols])

    Returns:
        float: Expected value of the game for the row player
    """
    p0 = np.array(p0_sol).reshape(-1, 1)  # Column vector
    p1 = np.array(p1_sol).reshape(1, -1)  # Row vector
    return float(np.sum(p0 * payoff_matrix * p1))
```

```python

def expand_strategy(short_strategy, flag):
    """
    Expands a strategy vector to full size using a flag vector.

    Parameters:
        short_strategy (List[float]): Strategy values for undominated strategies.
        flag (List[int]): Binary flag (1 = keep, 0 = dominated).

    Returns:
        np.ndarray: Full strategy with 0s in dominated positions.
    """
    full_strategy = np.zeros(len(flag))
    idx = 0
    for i, keep in enumerate(flag):
        if keep == 1:
            full_strategy[i] = short_strategy[idx]
            idx += 1
    return full_strategy

def generate_strictly_positive_probabilities(n: int, min_value=0.0000000001) -> list[float]:
    """
    Generate a random probability vector of length n where:
    - All probabilities are strictly > 0
    - Sum is exactly 1.0 (within rounding error)
    - Probabilities are rounded to 4 decimal places

    Parameters:
        n (int): Length of the vector
        min_value (float): Minimum value for any individual probability (default: 0.01)

    Returns:
        List[float]: A probability vector of length n
    """
    if n <= 0:
```

```python
        return []
    if n == 1:
        return [1.0]

    # Ensure that we can assign at least `min_value` to each of n entries
    remaining_mass = 1.0 - n * min_value
    if remaining_mass < 0:
        raise ValueError(f"min_value too large for vector of length {n}.")

    # Generate n random values that sum to remaining_mass
    raw = np.random.rand(n)
    raw_sum = np.sum(raw)
    scaled_raw = (raw / raw_sum) * remaining_mass

    # Add minimum value to each entry and round
    probs = [np.round(val + min_value, 4) for val in scaled_raw]

    # Fix rounding drift
    total = sum(probs)
    diff = np.round(1.0 - total, 4)
    if diff != 0:
        i = np.argmax(probs)
        probs[i] = np.round(probs[i] + diff, 4)
        if probs[i] <= 0:
            probs[i] = min_value  # Ensure it's still positive

    return probs

def solve_with_debtoru_solver(game, row_flag, dominance_mask):
    assert isinstance(game, pyspiel.MatrixGame)
    assert game.get_type().information == pyspiel.GameType.Information.ONE_SHOT
    assert game.get_type().utility == pyspiel.GameType.Utility.ZERO_SUM

    num_rows = game.num_rows()
    num_cols = game.num_cols()
```

```
    cvxopt.solvers.options["show_progress"] = False

    # ----------------------------
    # Row player's LP (OpenSpiel version)
    # ----------------------------
    lp0 = LinearProgram(OBJ_MAX)
    for r in range(num_rows):
        lp0.add_or_reuse_variable(r, lb=0)
    lp0.add_or_reuse_variable(num_rows)  # V
    lp0.set_obj_coeff(num_rows, 1.0)     # max V

    for c in range(num_cols):
        lp0.add_or_reuse_constraint(c, CONS_TYPE_GEQ)
        for r in range(num_rows):                                   (variable) r: int
            lp0.set_cons_coeff(c, r, game.player_utility(0, r, c))
        lp0.set_cons_coeff(c, num_rows, -1.0)  # -V >= 0

    lp0.add_or_reuse_constraint(num_cols + 1, CONS_TYPE_EQ)
    lp0.set_cons_rhs(num_cols + 1, 1.0)
    for r in range(num_rows):
        lp0.set_cons_coeff(num_cols + 1, r, 1.0)

    sol = lp0.solve()
    p0_sol = sol[:-1]      # row strategy
    p0_sol_val = sol[-1]   # game value for row player

    # ----------------------------
    # Column strategy (modified)
    # ----------------------------
```

```
    matrix = extract_row_payoff_matrix(game)
    # reduced_matrix, row_flag, dominance_mask = ieds_zero_sum_weak_flags(matrix)
    # print("\nRow Mask\n",row_flag)
    # print("\nColumn Mask\n",dominance_mask)
    num_undominated = sum(dominance_mask)

    if num_undominated == 0:
        raise ValueError("All column strategies are dominated — no valid strategy remains.")

    p1_sol = np.zeros(num_cols)
    p1_sol = generate_strictly_positive_probabilities(num_cols)

    # print("\np0_sol\n", p0_sol)
    # print("\np1_sol\n", p1_sol)
    # ----------------------------
    # Return as in original OpenSpiel
    # ----------------------------
    p1_sol_val = compute_column_player_value(p0_sol, p1_sol, matrix)  # skipped LP for column player

    p0_sol_full = expand_strategy(p0_sol, row_flag)
    p1_sol_full = expand_strategy(p1_sol, dominance_mask )

    return p0_sol_full, p1_sol_full, p0_sol_val, p1_sol_val

def generate_strategies(num_agents, num_arsenals):
    """
    Generate all unique descending allocations (canonical form) of num_agents across num_arsenals.
    Each allocation is a tuple of non-negative integers that sum to num_agents,
    sorted in descending order. No repeated permutations.
    The result is sorted in reverse lexicographical order.
```

```
    """
    strategies = set()

    # Generate all possible combinations
    for combo in product(range(num_agents + 1), repeat=num_arsenals):
        if sum(combo) == num_agents:
            strategies.add(tuple(sorted(combo, reverse=True)))  # enforce canonical form

    return sorted(strategies, reverse=True)

def compute_payoff_matrix(num_guerrillas, num_police, num_arsenals):
    g_strategies = generate_strategies(num_guerrillas, num_arsenals)
    p_strategies = generate_strategies(num_police, num_arsenals)
    print("\nRow Player (Guerrilla) Allocation:\n", textwrap.fill(str(g_strategies), width=100))
    print("\nColumn Player (Police) Allocation:\n", textwrap.fill(str(p_strategies), width=100))
    num_g = len(g_strategies)
    num_p = len(p_strategies)
    payoff_matrix = np.zeros((num_g, num_p))
    win_count = 0

    for i, g_strat in enumerate(g_strategies):
        for j, p_strat in enumerate(p_strategies):

            # Compare each guerrilla (row) group against each police (column) group
            comparisons = [(g, p) for g in g_strat for p in p_strat]
            win_count = sum(1 for g, p in comparisons if g > p)

            payoff = round(win_count / num_arsenals, 2)
            if payoff > 1.0:
                payoff = 1.0

            payoff_matrix[i, j] = payoff

    return payoff_matrix
```

```
def compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals):
    """
    Compute and display the Nash equilibrium strategies and game value for the Colonel Blotto game.
    """
    payoff_matrix = compute_payoff_matrix(num_guerrillas, num_police, num_arsenals)
    print(f"\nPayoff Matrix for {num_guerrillas} Guerrillas and {num_police} Police:")
    print(payoff_matrix)

    reduced_matrix, row_flag, dominance_mask = ieds_zero_sum_weak_flags(payoff_matrix)
    # print("\nreduced_matrix\n",reduced_matrix)
    # print("\nRow Mask\n",row_flag)
    # print("\nColumn Mask\n",dominance_mask)



    # Create OpenSpiel matrix game
    row_utilities = reduced_matrix.tolist()
    col_utilities = [[-cell for cell in row] for row in row_utilities]  # zero-sum version
    game = pyspiel.create_matrix_game(row_utilities, col_utilities)

    start_time = time.time()
    # Solve using LP solver
    row_strategy, col_strategy, game_value_r, game_value_c = solve_with_debtoru_solver(game, row_flag, dominance_mask)
    end_time = time.time()
    elapsed = end_time - start_time

    # Output strategies and game value
    row_strategy_rounded = [round(p, 3) for p in row_strategy]
    col_strategy_rounded = [round(p, 3) for p in col_strategy]
    game_value_r_rounded = round(game_value_r, 3)
    game_value_c_rounded = round(game_value_c, 3)

    print("\nNash Equilibrium:")
    formatted_row_strategy = [f"{float(x):.3f}" for x in row_strategy]
    print("\nRow Player (Guerrilla) Strategy:\n", textwrap.fill(str(formatted_row_strategy), width=100))
```

```
306
307     # print("\nColumn Player (Police) Strategy:\n", textwrap.fill(str(col_strategy_rounded), width=100))
308     formatted_col_strategy = [f"{float(x):.3f}" for x in col_strategy]
309     print("\nColumn Player (Police) Strategy:\n", textwrap.fill(str(formatted_col_strategy), width=100))
310
311     # print("\nValue of the Game (According to Row strategies of LP_Solver):", game_value_r)
312     print("\nValue of the Game (According to Row Prob from LP_Solver and PSA=1):", game_value_c)
313     print(f"\nExecution time of PSA: {elapsed:.6f} seconds")
314 if __name__ == "__main__":
315     num_guerrillas = int(input("Enter the number of guerrillas (row): "))
316     num_police = int(input("Enter the number of police (column): "))
317     num_arsenals = int(input("Enter the number of arsenals (battlefield): "))
318     #payoff_matrix_test = compute_payoff_matrix(num_guerrillas, num_police)
319     #print("Payoff Matrix for {} Guerrillas and {} Police:".format(num_guerrillas, num_police))
320     #print(payoff_matrix_test)
321     compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals)
```

```
Enter the number of guerrillas (row): 50
Enter the number of police (column): 70
Enter the number of arsenals (battlefield): 2

Row Player (Guerrilla) Allocation:
 [(50, 0), (49, 1), (48, 2), (47, 3), (46, 4), (45, 5), (44, 6), (43, 7), (42, 8), (41, 9), (40, 10),
(39, 11), (38, 12), (37, 13), (36, 14), (35, 15), (34, 16), (33, 17), (32, 18), (31, 19), (30, 20),
(29, 21), (28, 22), (27, 23), (26, 24), (25, 25)]

Column Player (Police) Allocation:
 [(70, 0), (69, 1), (68, 2), (67, 3), (66, 4), (65, 5), (64, 6), (63, 7), (62, 8), (61, 9), (60, 10),
(59, 11), (58, 12), (57, 13), (56, 14), (55, 15), (54, 16), (53, 17), (52, 18), (51, 19), (50, 20),
(49, 21), (48, 22), (47, 23), (46, 24), (45, 25), (44, 26), (43, 27), (42, 28), (41, 29), (40, 30),
(39, 31), (38, 32), (37, 33), (36, 34), (35, 35)]
```

```
Payoff Matrix for 50 Guerrillas and 70 Police:
[[0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
```

```
[1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
```

```
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]]
```

```
Nash Equilibrium:

Row Player (Guerrilla) Strategy:
 ['0.667', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.067',
'0.067', '0.067', '0.067', '0.067']

Column Player (Police) Strategy:
 ['0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.175', '0.000',
'0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.000', '0.043', '0.174', '0.170', '0.114',
'0.023', '0.149', '0.153']

Value of the Game (According to Row Prob from LP_Solver and PSA=1): 0.6666666803086407

Execution time of PSA: 0.004614 seconds
```

Appendix-C

```python
23# Date Jul 02, 2025
# Colonel Blotto Game for 'n' Battlefields, displaying Row and Column Strategies in Wrapped Format
#LP_Solver
import numpy as np
import pyspiel
from open_spiel.python.algorithms import lp_solver
import textwrap
import time

from itertools import product

def generate_strategies(num_agents, num_arsenals):
    """
    Generate all unique descending allocations (canonical form) of num_agents across num_arsenals.
    Each allocation is a tuple of non-negative integers that sum to num_agents,
    sorted in descending order. No repeated permutations.
    The result is sorted in reverse lexicographical order.
    """
    strategies = set()

    # Generate all possible combinations
    for combo in product(range(num_agents + 1), repeat=num_arsenals):
        if sum(combo) == num_agents:
            strategies.add(tuple(sorted(combo, reverse=True)))  # enforce canonical form

    return sorted(strategies, reverse=True)

def compute_payoff_matrix(num_guerrillas, num_police, num_arsenals):
    g_strategies = generate_strategies(num_guerrillas, num_arsenals)
    p_strategies = generate_strategies(num_police, num_arsenals)
    print("\nRow Player (Guerrilla) Allocation:\n", textwrap.fill(str(g_strategies), width=100))
    print("\nColumn Player (Police) Allocation:\n", textwrap.fill(str(p_strategies), width=100))
    num_g = len(g_strategies)
    num_p = len(p_strategies)
    payoff_matrix = np.zeros((num_g, num_p))
    win_count = 0
```

```
    for i, g_strat in enumerate(g_strategies):
        for j, p_strat in enumerate(p_strategies):

            # Compare each guerrilla (row) group against each police (column) group
            comparisons = [(g, p) for g in g_strat for p in p_strat]
            win_count = sum(1 for g, p in comparisons if g > p)

            payoff = round(win_count / num_arsenals, 2)
            if payoff > 1.0:
                payoff = 1.0

            payoff_matrix[i, j] = payoff

    return payoff_matrix

def compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals):
    """
    Compute and display the Nash equilibrium strategies and game value for the Colonel Blotto game.
    """
    payoff_matrix = compute_payoff_matrix(num_guerrillas, num_police, num_arsenals)
    print(f"\nPayoff Matrix for {num_guerrillas} Guerrillas and {num_police} Police:")
    print(payoff_matrix)

    # Create OpenSpiel matrix game
    row_utilities = payoff_matrix.tolist()
    col_utilities = [[-cell for cell in row] for row in row_utilities]  # zero-sum version
    game = pyspiel.create_matrix_game(row_utilities, col_utilities)

    # Solve using LP solver
    start_time = time.time()
    row_strategy, col_strategy, game_value_r, game_value_c = lp_solver.solve_zero_sum_matrix_game(game)
    end_time = time.time()
    elapsed = end_time - start_time
```

```
    # Output strategies and game value
    row_strategy_rounded = [round(p, 3) for p in row_strategy]
    col_strategy_rounded = [round(p, 3) for p in col_strategy]
    game_value_r_rounded = round(game_value_r, 3)
    game_value_c_rounded = round(game_value_c, 3)

    print("\nNash Equilibrium:")
    print("\nRow Player (Guerrilla) Strategy:\n", textwrap.fill(str(row_strategy_rounded), width=100))
    print("\nColumn Player (Police) Strategy:\n", textwrap.fill(str(col_strategy_rounded), width=100))
    print("\nValue of the Game (Expected Guerrilla Payoff_Row):", game_value_r_rounded)
    print("\nValue of the Game (Expected Guerrilla Payoff_Col):", game_value_c_rounded)
    print(f"Execution time of pure lp_solver: {elapsed:.6f} seconds")

if __name__ == "__main__":
    num_guerrillas = int(input("Enter the number of guerrillas (row): "))
    num_police = int(input("Enter the number of police (column): "))
    num_arsenals = int(input("Enter the number of arsenals (battlefield): "))
    #payoff_matrix_test = compute_payoff_matrix(num_guerrillas, num_police)
    #print("Payoff Matrix for {} Guerrillas and {} Police:".format(num_guerrillas, num_police))
    #print(payoff_matrix_test)
    compute_nash_equilibrium(num_guerrillas, num_police, num_arsenals)
```

```
Enter the number of guerrillas (row): 50
Enter the number of police (column): 70
Enter the number of arsenals (battlefield): 2

Row Player (Guerrilla) Allocation:
 [(50, 0), (49, 1), (48, 2), (47, 3), (46, 4), (45, 5), (44, 6), (43, 7), (42, 8), (41, 9), (40, 10),
(39, 11), (38, 12), (37, 13), (36, 14), (35, 15), (34, 16), (33, 17), (32, 18), (31, 19), (30, 20),
(29, 21), (28, 22), (27, 23), (26, 24), (25, 25)]

Column Player (Police) Allocation:
 [(70, 0), (69, 1), (68, 2), (67, 3), (66, 4), (65, 5), (64, 6), (63, 7), (62, 8), (61, 9), (60, 10),
(59, 11), (58, 12), (57, 13), (56, 14), (55, 15), (54, 16), (53, 17), (52, 18), (51, 19), (50, 20),
(49, 21), (48, 22), (47, 23), (46, 24), (45, 25), (44, 26), (43, 27), (42, 28), (41, 29), (40, 30),
(39, 31), (38, 32), (37, 33), (36, 34), (35, 35)]
```

```
Payoff Matrix for 50 Guerrillas and 70 Police:
[[0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1.  1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 1. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5 0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0. ]
 [1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.5
  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0. ]]
```

```
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  0.5 0.5 0.5 0.5 0.5 0.5 0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  1.  0.5 0.5 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
[1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
 1.  1.  1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]]

Nash Equilibrium:

Row Player (Guerrilla) Strategy:
 [0.238, 0.088, 0.065, 0.054, 0.048, 0.042, 0.039, 0.039, 0.054, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.122, 0.065, 0.055, 0.049, 0.044]

Column Player (Police) Strategy:
 [-0.0, -0.0, -0.0, -0.0, 0.0, 0.0, 0.0, 0.02, 0.026, 0.03, 0.034, 0.037, 0.041, 0.045, 0.047,
0.049, 0.053, 0.061, 0.078, 0.146, 0.0, 0.0, -0.0, -0.0, 0.0, 0.0, 0.0, 0.0, 0.071, 0.052, 0.047,
0.044, 0.042, 0.04, 0.037]

Value of the Game (Expected Guerrilla Payoff_Row): 0.667

Value of the Game (Expected Guerrilla Payoff_Col): -0.667
Execution time of pure lp_solver: 0.015649 seconds
```

References

1. Arad A, Rubinstein A (2012) Multi-dimensional iterative reasoning in action: The case of the Colonel Blotto game. J Econ Behav Organ 84(2):571–585
2. Hillman AL, Riley JG (1989) Politically contestable rents and transfers. Econ Polit 1(1):17–39
3. Roberson B (2006) The Colonel Blotto game. Econ Theory 29(1):1–24
4. Kovenock D, Roberson B (2012) Coalitional Colonel Blotto games with application to the economics of alliances. J Public Econ Theory 14(4):653–676
5. Kovenock D, Mauboussin MJ, Roberson B (2010) Asymmetric conflicts and endogenous dimensionality. Korean Econ Rev 26:287–305
6. Borel E, Ville J (1938) Applications de la théorie des probabilités aux jeux de hasard. J Gabay
7. Borel E (1921) La théorie du jeu et les équations intégrales à noyau symétrique. C R Acad Sci 173:1304–1308
8. Baye MR, Kovenock D, De Vries CG (1996) The all-pay auction with complete information. Econ Theory 8(2):291–305
9. McDonald J, Tukey J (1949) Colonel Blotto: a problem in military strategy. Fortune, June 1949
10. Chai P, Chuang J (2011) Colonel Blotto in the phishing war. Decis Game Theory Secur 201–218
11. Chia PH (2012) Colonel Blotto in web security. In: Proceedings of the Eleventh Workshop on Economics and Information Security (WEIS), Rump Session
12. Straffin PD (1993) Game theory and strategy. Mathematical Association of America, Washington, DC
13. Chowdhury S, Kovenock D, Sheremeta R (2013) An experimental investigation of Colonel Blotto games. Econ Theory 52(3):833–861
14. Rinott Y, Scarsini M, Yu Y (2012) A Colonel Blotto gladiator game. Math Oper Res 37:574–590
15. Daskalakis C, Goldberg PW, Papadimitriou CH (2006) The complexity of computing a Nash equilibrium. In: Proceedings of STOC 2006
16. von Stengel B (2002) A note on the accuracy of the support enumeration algorithm. Int J Game Theory 31:213–220
17. Savani R, von Stengel B (2006) Computing Nash equilibria: Approximation and smoothed analysis. In: Proc. of the 47th IEEE Symposium on Foundations of Computer Science (FOCS)
18. Kask K, Goldberg PW, von Stengel B (2011) The Lemke–Howson algorithm is not strongly polynomial. Math Oper Res 36(4):594–609
19. Deng X, Papadimitriou CH, Teng SH (2002) On the complexity of Nash equilibrium computation. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC), pp 123–132
20. von Stengel B (1999) A counterexample to the Lemke–Howson algorithm. Econ Theory 14(2):327–330
21. Savani R, von Stengel B (2004) Computing Nash equilibria in polymatrix games. In: Proceedings of the 3rd International Conference on Algorithms and Complexity (CIAC), pp 57–68
22. Osborne MJ, Rubinstein A (1994) A course in game theory. MIT Press, Cambridge

23. Papadimitriou CH (2001) Algorithms, games, and the internet. In: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC)
24. Chen X, Deng X, Teng SH (2009) Settling the complexity of computing Nash equilibrium. J ACM 56(3):1–57
25. Selten R (1975) Reexamination of the perfectness concept for equilibrium points in extensive games. Int J Game Theory 4(1):25–55
26. Nisan N, Roughgarden T, Tardos É, Vazirani VV (2007) Algorithmic game theory. Cambridge Univ Press, Cambridge
27. Cover TM, Thomas JA (2006) Elements of information theory. Wiley-Interscience, Hoboken
28. Aumann RJ, Maschler M (1995) Repeated games with incomplete information. MIT Press, Cambridge
29. Ahmadinejad AM, Dehghani S, Hajiaghayi MT, Lucier B, Mahini H, Seddighin S (2016) From duels to battlefields: Computing equilibria of Blotto and other games. CoRR abs/1603.03146. https://arxiv.org/abs/1603.03146
30. Codenotti B, De Rossi LC, Pagan M (2008) An experimental analysis of the Lemke–Howson algorithm. Math Oper Res 33(4):924–949
31. Savani R, von Stengel B (2004) Exponentially many steps for finding a Nash equilibrium in a bimatrix game. In: Proc. 45th IEEE Symposium on Foundations of Computer Science (FOCS), pp 258–267