# An application of the Colonel Blotto Game

Andrea Maino
*dept. of Finance*
*SFI PhD Program; University of Geneva and GFRI*
Geneva, Switzerland
andrea.maino@etu.unige.ch

***Abstract*—This project proposes a generalization of the Colonel Blotto game with the objective of studying different strategies in order to maximize the likelihood of winning against a group of opponents. We consider a set of players which compete each other in a tournament in a configuration of the game where: the board weighting is not uniform, the player objective is to maximize the average score against the fixed set of opponent and each opponents has a discrete and equal amount of initial endowment of troops. A set of strategies are proposed and analysed using numerical optimization techniques.**

***Index Terms*—Game Theory, zero sum games, Colonel Blotto game, exponential complexity, computational science, numerical algorithms**

## I. Introduction

One of the problems which game theorists face in their profession is the problem of computing Nash Equilibria of zero sum games. In research applications, most of the zero sum games of interest have exponentially many strategies but highly structured payoffs. Among the class of zero sum games a well known game is the Colonel Blotto Game. In the Colonel Blotto game, players distribute a pool of troops among a set of battlefields with the goal of maximizing the likelihood of winning against an opponent or a set of opponents given a specific payoff function establishing the award winning rule of a battlefield. His importance among other zero sum games has grown since his first publication in 1921 by french mathematician Emile Borel because of its applicability with real life situations. Indeed, Colonel Blotto game is used as model of a wide range of applications characterized by limited resources and a set of stylized battlefields where two or more opponents face each other. Examples are U.S. presidential elections, innovative technology competition etc. . The main challenge in resolving a Colonel Blotto game arises from the size of the strategy space for which standard methods for computing equilibria of zero sum games fail to be computationally feasible. Despite the very important applications, to date only few solutions to special variants are known.

### A. An overview of zero sum games

It is well know that every finite game admits a Nash equilibrium i.e. a profile of strategies for which no player can benefit from an unilateral deviation. However, there is no obvious way on how to find a Nash-equilibrium. Computing a Nash equilibrium for a normal form game is know to be a PPAD-complete problem even for two players game as proved by Chen and Deng (2006).
This issue, motivates the study of games for which the equilibrium can be computed efficiently, which often reduces in finding structures in games which can be exploited to admit computational results. The most well known class of such games is the class of zero-sum two players games in which player 2's payoff is the negation of player 1's payoff. The normal form representation of such zero sum game is a matrix which specifies the payoff for player 1. This class of games is well used for modelling perfect competitions among two parties. Using Linear Programming methods, can be shown that because the payoff of a zero sum game is a matrix, a Nash-equilibrium can be computed in polynomial time. Hence, time polynomial is the number of strategies to each player. However, in case where the number of strategies is exponential, the above discussion fails to guarantee efficient computations of equilibrium and an alternative approach is warranted. For more information, refer to Ahmadinejad at al. (2017), Behnezhad et al. (2018), Ferdowsi et. al (2018).

### B. Colonel Blotto Game

In the standard configuration of Colonel Blotto game, two colonels have a fixed pool of troops and face each other over a set of battlefields. The two colonels simultaneously divide their troops between the battlefields at the outset of the game. In the original configuration, a colonel wins a battlefield if the number of his troops dominates the number of troops of the opponent. The final payoff to each colonel is the number of battlefields won which can be weighted in more advanced configurations. An equilibrium in this game is a pair of colonel's strategies, which is (potentially randomized) a distribution of troops across battlefields, such that none of the opponents has incentive to deviate his strategy. Although it is a zero sum game, the number of strategies in Colonel Blotto is exponential in the number of troops. Indeed, the ways to partition n troops among k battlefields is given by the *Stirling partition number*, which is asymptotically exponential in the number of troops, and therefore, traditional approaches for solving zero sum games do not yield computationally efficient results.
Several efforts have been made to understand the equilibria

and solve the equilibrium explicitly. Existing works consider a continuous relaxation of the problem where troops are divisible. A recent breakthrough came in a seminal work by Roberson (2006) which proposes a solution of the equilibrium in case of continuous game and equally weighted battlefields. Another interesting contribution by Behnezhad et al. (2018) which analyses games in which maximizing the expected payoff deviates from the actual goal of the paper. In standard game theory configurations, mixed strategies are often evaluated based on the expected payoff that they guarantee. However, this is not always desirable because maximizing the expected payoff and the likelihood of winning might differ. An application of such configuration is the Colonel Blotto game. They show that maximizing the expected payoff of a player does not necessarily maximize the winning probabilities for certain applications of the game. For instance, in U.S. presidential campaigns, the player's goal is to maximize the probability of winning more than half the votes rather than maximizing the expected number of votes.

## II. Formalization of the Colonel Blotto Game

The simplest version of Colonel Blotto Game can be formalized as follows. Two players simultaneously allocate $a$ and $b$ troops over $K$ battlefields. A pure strategy of player A is a K-partition $x = (x_1, x_2, ..., x_K)$ with $\sum_{i=1}^{K} x_i = a$ and similarly for player B. Defining $u_i^A(x_i, y_i)$ as the payoff of player A from the i-th battlefield, since the Colonel Blotto is a zero sum game, we have that $u_i^A(x_i, y_i) = -u_i^B(x_i, y_i)$. The total payoff of the game for player A is $h_A(x, y) = \sum_i u_i^A(x_i, y_i)$ and similarly for player B. Finally, a mixed strategy of each player is a probability distribution over his pure strategies.

## III. An application of the Colonel Blotto Game

The proposed configuration of Colonel Blotto game considers a set of opponents, each playing independently and empowered with 100 soldiers, which they can allocate in 10 battlefields numbered from 1 to 10 and each worth 1 to 10 points, corresponding to the identification number of the castle. Battlefields are fought in order, starting from battlefield 1 to 10. For each couple of opponents, the allocation of troops on the battlefields is compared starting from battlefield number 1 and so on. Whoever has the most soldiers on the battlefield wins it (in the case of a tie no one gets points). Moreover, as soon as one player has won three consecutive battlefields, all remaining battlefields are conquered. An example is give in the below table:

TABLE I
Table Type Styles

| **Battlefields** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Colonel 1** | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| **Colonel 2** | 5 | 10 | 15 | 17 | 8 | 5 | 5 | 15 | 18 | 2 |

In this game, Colonel 1 wins battlefields 1,5,6,7,8,9 and 10 for a total score of 46, while Colonel 2, wins battlefields 3 and 4 for a total score of 7. Battlefield 2 is a draw and Colonel 1 wins battlefields 8 and 9 because he has already won the consecutive battlefields 5,6, and 7.
In this configuration of the game, imagine it as a tournament, each Colonel plays against each other and has to select, simultaneously, their allocations on the battlefields in order to maximize the average score of all the games of the tournament.

## IV. Proposed solution

In this section I elaborate the proposed solution to maximize the average score in the tournament. The approach implemented is not based on analytical results from game theory but rather on making a conjecture on which structure of strategies is expected to yield a better average score. Once rationalized the base approach, I propose and implement an algorithm able to optimize, starting on this initial guess of benchmark strategies, the average score given a set of opponents strategies. In particular, starting from a set of player's benchmark strategies which performances are computed against a set of large opponents strategies, a *Stochastic Hill Climbing* optimization algorithm is implemented in order to select from a starting strategy the optimal neighbor strategy.

As will be discussed later, two major challenges arises from this problem. First, the variable space is large and the number of possible strategies is asymptotically exponential in the number of castles and initial endowment of troops which makes the computing effort very high. For instance, given 10 castles and 100 initial troops, the possible game configurations are given by the partitions of a set of n elements in k non empty sets and is given by the *Stirling number of second kind*. Second, the average score, which depends on the player's strategy and on the strategies of the opponents, which are unknown at the time of the resource allocation and represent a discrete domain, is a highly discontinuous function which makes complex the implementation of "standard" optimization algorithms. Therefore it is required to implement an heuristic algorithm which works in such context.

### A. Algorithm Implementation

Assuming that all opponents are rational and have the same objective of maximizing their average score, it is possible to rationalize benchmark strategies from which to create opponent's strategies. At a first look, one may think that in order to maximize the average score, one should be better off by trying to find the strategy which could win against any opponents. However, interestingly, two considerations need to be taken into account. First, such strategy does not exist. Indeed, for any player's configuration there exist at least a configuration which beats such strategy. A similar consideration holds for the average score, meaning that it is unluckily that a global maxima exists and if so would be very difficult to find it given the exponential number of strategies and the high discontinuity of the score function. Second, the fact that castles have different weightings, in combination with the "take all" rule, makes difficult to anticipate which castles might attract the highest average score. Indeed, while the value of winning individual

first castles is low, the "take all" rule make them more valuable on that account compared to last castles which instead have higher individual value but low "take all" rule value. Finally, even if such a preferred configuration of castle existed, rational players would concentrate their choices on such castles which would reduce the effective value of strategies targeting them. This last observation relates to the idea that the value of a strategy depends on ex-ante anticipation of opponents strategies which make the problem more complex. Although relevant, this is not taken into account in this project which focuses instead in an "atomistic" allocation of resources by each player meaning that no "strategic allocation" is taken into account. Not considering the effect of strategy concentration and player strategical anticipation is a good approximation when the number of total tournament players is low compared to the total configuration space. Intuitively, this space is extremely large from a *Manhattan distance* point of view and it is very unlucky without a very high number of participants that players allocate troops in similar configurations from a Manhattan distance perspective.

Following the above reasoning, I generate using six base strategies a set of opponents strategies. Starting from these strategies, random perturbation are computed in order to finally obtain a set of opponents strategies.
The selected strategies are based on the conjecture that first castles are more valuable than last ones give the "take all" rule. Although a more rigorous approach would be needed to validate this conjecture, it seams reasonable on several accounts. I leave to the reader to develop an intuition about it. These base strategies are represented below:

1) $Opponents\_group\_A$: $Base\_opponent\{1, 1, 40, 24, 14, 5, 0, 0, 0, 0\}$;
2) $Opponents\_group\_B$: $Base\_opponent\{1, 1, 35, 25, 20, 3, 0, 0, 0, 0\}$;
3) $Opponents\_group\_C$: $Base\_opponent\{1, 35, 24, 20, 5, 0, 0, 0, 0, 0\}$;
4) $Opponents\_group\_D$: $Base\_opponent\{1, 40, 1, 0, 25, 8, 5, 0, 0, 0\}$;
5) $Opponents\_group\_E$: $Base\_opponent\{15, 35, 25, 5, 0, 0, 0, 0, 0, 0\}$;
6) $Opponents\_group\_F$: $Base\_opponent\{5, 5, 10, 9, 10, 10, 9, 8, 6, 6\}$;
7) $Opponents\_group$: $Base\_opponent\{0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$;

With the exception of the 6th strategy which is the more equilibrated among the other six, the first 5 strategies are mostly focused on the value of the "take all" rule.
All the opponents strategies are collected in a database given by a vector of vectors data structure in cpp. I use the database to select among each group of strategies which ones have the highest average score. These selected strategies are then used as base strategies to search for local maxima using the procedure "$maxima\_spp\_detection\_algorithm$" which implements the *Stochastic Hill Climbing* algorithm on possible alternative configurations based on the Manhattan metrics. The Manhattan metrics is selected given the discrete nature of the space of the strategies configuration.
For each of the initial group of opponents strategies, I obtain a vector of vectors containing local maxima of strategies facing the entire database of opponents. Once the average score is computed with the function "$Check\_multi\_score$", the strategies are sorted based on their relative score using the function "$special\_sort$". "$special\_sort$" allows to sort strategies based on their relative score. These selected strategies are then used to build a database of selected strategies named "Selected Database".
Using the function "$maxima\_detection\_algorithm$", which finds local maxima starting from a base configuration (balanced) of $\{5, 5, 10, 9, 10, 10, 9, 8, 6, 6\}$, I generate a set of strategies optimized to face the opponents strategies of the "Selected Database". The "$Selected\_Maxima$" strategies are sorted and only the best are selected. These "$Selected\_Maxima$" are also evaluated on the full database of opponents. Finally, a cross validation is performed on a random cross validation set of opponents strategies. I compute the score of the selected Maxima on this set of opponents strategies. It is then possible to compute the weighted average score of the selected maxima on the three different opponents databases and sort the strategies on that average score measure. The final output is a vector of vector which have the highest average score on these 3 database of opponents strategies. They represent local maxima in the "small" strategies configuration space considered.

## V. Optimization Strategy

*Stochastic Hill Climbing* is a stochastic variant of the standard *Hill Climbing* algorithm. The stochasticity comes from the random choice of the uphill move among the set of uphill moves. The Hill Climbing algorithm is a local search algorithm. This type of algorithm are well suited to approach optimizations of discrete valued functions. A famous application relates to the "travelling salesman problem". Similarly to other optimization algorithm, the Hill climbing algorithm is guaranteed to find optimal solutions only in convex problems while in general finds only local optima. The simplicity of the implementation makes it often used in artificial intelligence problems. Importantly, the Hill Climbing is an "anytime" algorithm meaning that it returns a possible solution at any intermediate iteration. This property is important in the Colonel Blotto game given the difficulty to asses the local convergence.

In summary, as can be understood from the previous discussion, the choice of Hill Climbing algorithm over more used optimization algorithm of similar nature, for instance the (Stochastic) Gradient Descent algorithm, is given by the discrete nature and high discontinuity of the problem at hand.

## VI. Results

In this section, I show the results from a given configuration of the game. In particular I consider the below set up:

- Initial troops endowment for each player of 100;
- 10 castles with relative weight and "get all" rule;

- Opponents Database strategies:90
- Selected Database size:250
- Cross-Validation Database:1000

The procedure starts by creating the database of opponents strategies by creating board configuration starting from benchmark strategies. In Figure 1, 15 strategies are created from Benchmark Strategy A as illustrated above. Among these

```
==============================


========RESULTS GROUP A:==============


==============================

Group_A[1] = [6, 1, 42, 25, 21, 5, 0, 0, 0, 0]
Group_A[2] = [3, 1, 45, 29, 14, 8, 0, 0, 0, 0]
Group_A[3] = [1, 5, 40, 28, 15, 11, 0, 0, 0, 0]
Group_A[4] = [1, 3, 42, 29, 20, 5, 0, 0, 0, 0]
Group_A[5] = [1, 6, 40, 28, 15, 10, 0, 0, 0, 0]
Group_A[6] = [5, 1, 40, 24, 18, 12, 0, 0, 0, 0]
Group_A[7] = [2, 2, 48, 29, 14, 5, 0, 0, 0, 0]
Group_A[8] = [1, 1, 40, 35, 14, 9, 0, 0, 0, 0]
Group_A[9] = [11, 2, 40, 24, 14, 9, 0, 0, 0, 0]
Group_A[10] = [2, 1, 43, 34, 14, 6, 0, 0, 0, 0]
Group_A[11] = [7, 6, 40, 28, 14, 5, 0, 0, 0, 0]
Group_A[12] = [8, 1, 42, 24, 18, 7, 0, 0, 0, 0]
Group_A[13] = [1, 6, 42, 27, 19, 5, 0, 0, 0, 0]
Group_A[14] = [1, 1, 44, 24, 20, 10, 0, 0, 0, 0]
Group_A[15] = [3, 1, 43, 32, 14, 7, 0, 0, 0, 0]
```

Fig. 1.

strategies, a subset are selected after evaluation the relative score of each strategy in the set of originated strategies. These selected strategies, see Figure 2, represent the "Selected strategies" from Group A which are then used, together with all the selected strategies from the remaining five groups to build the "Selected Database" of strategies. This is obtained by generating strategies starting from these selected strategies and finding local maxima for each group. The best performing of these local maxima are then used to generate the "Selected Database". Figure 3 illustrates the newly generated strategies

```
================================================


========STARTING PLAYER SELECTION STARTEGIES:=============


================================================


===========STARTING STARTEGIES A:================


================================================




Selected_A Strategies

Selected_A[1] score  = 36.3222   Selected_A[1] = [1, 6, 40, 28, 15, 10, 0, 0, 0, 0]
Selected_A[2] score  = 36.9   Selected_A[2] = [11, 2, 40, 24, 14, 9, 0, 0, 0, 0]
Selected_A[3] score  = 37.4111   Selected_A[3] = [7, 6, 40, 28, 14, 5, 0, 0, 0, 0]
Selected_A[4] score  = 37.9   Selected_A[4] = [1, 3, 42, 29, 20, 5, 0, 0, 0, 0]
Selected_A[5] score  = 40.3222   Selected_A[5] = [2, 2, 48, 29, 14, 5, 0, 0, 0, 0]
```

Fig. 2.

from the Group A selected strategies. The best performing will be then selected to form the "Selected Database". Figure

```
New Strategies created starting from the Selected_A ones

Maxima_A[1] score  = 43.1222   Maxima_A[1] = [0, 2, 43, 29, 16, 6, 1, 2, 1, 0]
Maxima_A[2] score  = 43.1222   Maxima_A[2] = [6, 4, 41, 30, 11, 4, 1, 1, 1, 1]
Maxima_A[3] score  = 43.2111   Maxima_A[3] = [0, 2, 43, 31, 16, 3, 2, 1, 1, 1]
Maxima_A[4] score  = 43.3556   Maxima_A[4] = [0, 2, 43, 31, 21, 0, 1, 1, 1, 0]
Maxima_A[5] score  = 43.5667   Maxima_A[5] = [8, 7, 41, 30, 11, 0, 1, 1, 1, 0]
Maxima_A[6] score  = 43.8667   Maxima_A[6] = [0, 6, 37, 29, 14, 9, 1, 2, 1, 1]
Maxima_A[7] score  = 44.3444   Maxima_A[7] = [0, 2, 45, 34, 16, 0, 1, 1, 1, 0]
Maxima_A[8] score  = 44.4222   Maxima_A[8] = [4, 4, 41, 30, 16, 2, 1, 1, 1, 0]
Maxima_A[9] score  = 44.4778   Maxima_A[9] = [4, 3, 44, 29, 11, 6, 1, 1, 1, 0]
Maxima_A[10] score  = 44.5778   Maxima_A[10] = [2, 2, 41, 30, 16, 6, 1, 1, 1, 0]
Maxima_A[11] score  = 44.5889   Maxima_A[11] = [2, 2, 41, 30, 22, 0, 1, 1, 1, 0]
Maxima_A[12] score  = 44.6   Maxima_A[12] = [6, 4, 41, 30, 16, 0, 1, 1, 1, 0]
Maxima_A[13] score  = 44.6778   Maxima_A[13] = [4, 2, 45, 25, 15, 6, 1, 1, 1, 0]
Maxima_A[14] score  = 44.6889   Maxima_A[14] = [4, 4, 46, 29, 11, 3, 1, 1, 1, 0]
Maxima_A[15] score  = 44.7778   Maxima_A[15] = [0, 6, 41, 32, 7, 9, 1, 2, 1, 1]
Maxima_A[16] score  = 44.8444   Maxima_A[16] = [4, 7, 41, 30, 15, 0, 1, 1, 1, 0]
Maxima_A[17] score  = 44.8889   Maxima_A[17] = [4, 4, 46, 27, 15, 1, 1, 1, 1, 0]
Maxima_A[18] score  = 45.1444   Maxima_A[18] = [3, 4, 45, 29, 16, 0, 1, 1, 1, 0]
Maxima_A[19] score  = 45.1889   Maxima_A[19] = [4, 2, 46, 29, 16, 0, 1, 1, 1, 0]
Maxima_A[20] score  = 45.6444   Maxima_A[20] = [6, 2, 42, 21, 15, 9, 1, 2, 1, 1]
Maxima_A[21] score  = 45.6889   Maxima_A[21] = [2, 7, 43, 25, 11, 9, 0, 1, 1, 1]
Maxima_A[22] score  = 45.7111   Maxima_A[22] = [2, 2, 40, 29, 14, 9, 1, 1, 1, 1]
Maxima_A[23] score  = 46.3556   Maxima_A[23] = [2, 2, 43, 25, 11, 12, 1, 2, 1, 1]
Maxima_A[24] score  = 46.3778   Maxima_A[24] = [2, 2, 42, 21, 15, 13, 1, 2, 1, 1]
Maxima_A[25] score  = 46.4444   Maxima_A[25] = [2, 2, 42, 24, 14, 12, 1, 1, 1, 1]
Maxima_A[26] score  = 46.4444   Maxima_A[26] = [2, 2, 45, 21, 11, 13, 1, 2, 2, 1]
Maxima_A[27] score  = 46.6   Maxima_A[27] = [2, 2, 42, 25, 15, 9, 1, 2, 1, 1]
```

Fig. 3.

4 illustrates the selected local maxima from the selected group strategies. Only top 300 strategies are selected.

```
****Maxima Detection Algorithm with Selected Database****

maxima_detection_algorithm with 1000 starting_points and 5 restarts
tooks 42588584ticks, or 42.5886seconds to calculate it.



Printing the best 300 choises (if there are more than 300 strategies)
Selected_Maxima[13] score  = 40.376   Selected_Maxima[13] = [22, 8, 12, 8, 32, 10, 3, 3, 2, 0]
Selected_Maxima[14] score  = 40.38   Selected_Maxima[14] = [10, 7, 0, 21, 24, 24, 11, 3, 0, 0]
Selected_Maxima[15] score  = 40.412   Selected_Maxima[15] = [12, 8, 31, 3, 12, 17, 7, 7, 2, 1]
Selected_Maxima[16] score  = 40.416   Selected_Maxima[16] = [16, 8, 12, 10, 33, 7, 7, 3, 2, 2]
Selected_Maxima[17] score  = 40.424   Selected_Maxima[17] = [22, 7, 12, 27, 16, 7, 2, 3, 2, 2]
Selected_Maxima[18] score  = 40.428   Selected_Maxima[18] = [10, 8, 31, 3, 12, 17, 7, 7, 2, 3]
Selected_Maxima[19] score  = 40.488   Selected_Maxima[19] = [20, 8, 12, 8, 32, 7, 7, 3, 3, 0]
Selected_Maxima[20] score  = 40.504   Selected_Maxima[20] = [11, 8, 0, 21, 22, 24, 11, 3, 0, 0]
Selected_Maxima[21] score  = 40.512   Selected_Maxima[21] = [8, 8, 31, 22, 12, 3, 11, 0, 2, 3]
Selected_Maxima[22] score  = 40.536   Selected_Maxima[22] = [11, 7, 12, 21, 22, 15, 7, 3, 1, 1]
Selected_Maxima[23] score  = 40.592   Selected_Maxima[23] = [23, 8, 12, 8, 32, 3, 7, 3, 2, 2]
Selected_Maxima[24] score  = 40.608   Selected_Maxima[24] = [20, 8, 12, 8, 32, 6, 7, 3, 2, 2]
Selected_Maxima[25] score  = 40.624   Selected_Maxima[25] = [8, 8, 31, 15, 12, 5, 16, 3, 2, 0]
Selected_Maxima[26] score  = 40.624   Selected_Maxima[26] = [11, 8, 0, 27, 16, 24, 11, 3, 0, 0]
Selected_Maxima[27] score  = 40.632   Selected_Maxima[27] = [8, 8, 31, 11, 12, 5, 16, 3, 6, 0]
Selected_Maxima[28] score  = 40.64   Selected_Maxima[28] = [19, 8, 12, 10, 32, 5, 7, 3, 2, 2]
Selected_Maxima[29] score  = 40.672   Selected_Maxima[29] = [6, 7, 9, 31, 17, 21, 2, 3, 2, 2]
Selected_Maxima[30] score  = 40.68   Selected_Maxima[30] = [10, 8, 31, 8, 12, 7, 11, 7, 3, 3]
Selected_Maxima[31] score  = 40.7   Selected_Maxima[31] = [10, 8, 31, 10, 12, 5, 11, 7, 6, 0]
Selected_Maxima[32] score  = 40.72   Selected_Maxima[32] = [12, 8, 0, 22, 22, 24, 7, 3, 1, 1]
Selected_Maxima[33] score  = 40.74   Selected_Maxima[33] = [11, 7, 12, 27, 16, 15, 7, 3, 1, 1]
Selected_Maxima[34] score  = 40.752   Selected_Maxima[34] = [20, 8, 12, 8, 32, 10, 2, 3, 2, 3]
Selected_Maxima[35] score  = 40.76   Selected_Maxima[35] = [10, 8, 31, 2, 12, 17, 11, 7, 1, 1]
Selected_Maxima[36] score  = 40.764   Selected_Maxima[36] = [11, 4, 12, 26, 32, 3, 2, 6, 2, 2]
Selected_Maxima[37] score  = 40.772   Selected_Maxima[37] = [10, 8, 31, 3, 12, 17, 12, 4, 1, 2]
Selected_Maxima[38] score  = 40.772   Selected_Maxima[38] = [16, 8, 12, 22, 17, 7, 11, 3, 2, 2]
Selected_Maxima[39] score  = 40.784   Selected_Maxima[39] = [11, 8, 31, 11, 12, 7, 12, 5, 3, 0]
Selected_Maxima[40] score  = 40.784   Selected_Maxima[40] = [11, 8, 0, 22, 28, 21, 7, 3, 0, 0]
Selected_Maxima[41] score  = 40.804   Selected_Maxima[41] = [8, 8, 31, 11, 12, 7, 12, 5, 3, 3]
Selected_Maxima[42] score  = 40.808   Selected_Maxima[42] = [11, 7, 12, 22, 22, 17, 2, 3, 2, 2]
Selected_Maxima[43] score  = 40.812   Selected_Maxima[43] = [11, 8, 0, 26, 22, 21, 7, 3, 1, 1]
Selected_Maxima[44] score  = 40.816   Selected_Maxima[44] = [11, 8, 0, 21, 24, 24, 7, 3, 1, 1]
Selected_Maxima[45] score  = 40.82   Selected_Maxima[45] = [10, 8, 31, 10, 12, 7, 11, 4, 5, 2]
Selected_Maxima[46] score  = 40.832   Selected_Maxima[46] = [10, 8, 31, 11, 12, 7, 11, 4, 3, 3]
Selected_Maxima[47] score  = 40.84   Selected_Maxima[47] = [10, 8, 31, 10, 12, 5, 16, 6, 2, 0]
Selected_Maxima[48] score  = 40.844   Selected_Maxima[48] = [10, 8, 31, 10, 12, 7, 11, 5, 6, 0]
Selected_Maxima[49] score  = 40.864   Selected_Maxima[49] = [16, 8, 0, 11, 34, 24, 7, 0, 0, 0]
Selected_Maxima[50] score  = 40.868   Selected_Maxima[50] = [10, 7, 12, 26, 22, 7, 11, 3, 2, 0]
Selected_Maxima[51] score  = 40.872   Selected_Maxima[51] = [8, 8, 31, 11, 12, 7, 12, 5, 6, 0]
```

Fig. 4.

The Selected maxima are then tested against the "Selected Database", the generic Database and the cross validation database and only the best scoring strategies are finally selected on each of the opponents sets of strategies. Figure 5 and Figure 6 illustrates this steps.

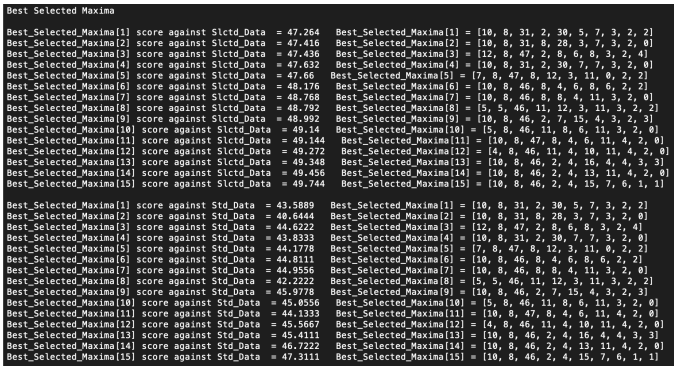


Fig. 5.

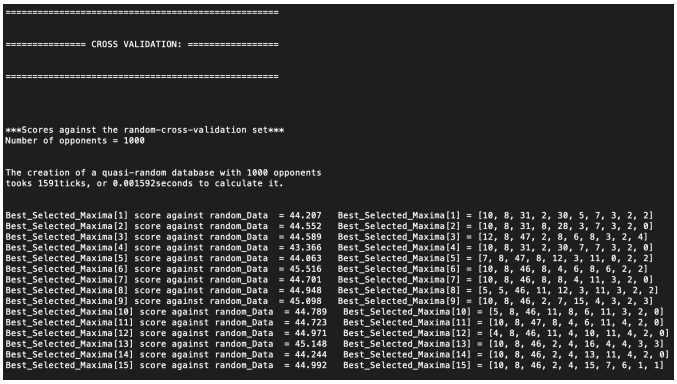


Fig. 6.

Finally the best performing strategies on the three sets of opponents are ranked and identified. Figure 7 illustrates the final results.

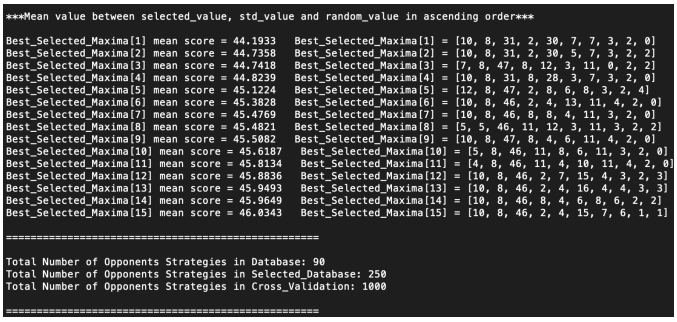


Fig. 7.

## VII. Further Improvements

This is clearly a preliminary version of the analysis of this specific configuration of the Colonel Blotto game from a computational point of view. Several questions remain unanswered and several additional developments would be interesting to explore.

## VIII. Conclusions

In this project I have introduced the Colonel Blotto game and highlighted its importance among the class of zero sum games. Its relevance arises from the possibility of rationalizing an array of empirical application into the game Blotto game framework.

In the second section the discussion moves on the challenges of finding optimal strategies in a tournament of Blotto Games. In this configuration, a set of players face each other given an initial endowment of troops and a fixed number of castles. The game rules are different from the traditional Blotto game and introduces different scoring weights to each caste. This weighting scheme makes more difficult to anticipate the exact value of each castle such that it is not easy to rationalize concentration of strategies around specific castles. The winning strategy is the one which maximizes the average score against a fixed set of opponents. Therefore, this project proposes a procedure to find such strategy by rationalizing possible opponents strategies and finding local maxima among possible strategies chosen by the player.

The challenges of the optimization of the scoring function are twofold. First, the configuration space is large and exponential in the number of castles and initial endowment which make difficult to apply numerical procedures from a computing complexity point of view. Second, the discrete nature of strategies configuration and the high discontinuity of the score function make less obvious the implementation of optimization algorithm. To that end, in this project I proposed to implement a Stochastic Hill Climbing algorithm to find local maxima facing a set of opponents strategies.

This project clearly represent a first attempt to approach this interesting problem and further research is needed.

## IX. Appendix A: Code examples and Algorithms

### A. Stochastic Hill Climbing Algorithm

```
////////////////////////////////

/*
//Function that looks for the highest local
    starting from a given point assigned in
    Strategy;
//Implements the stochastic hill climbing
    algorithm;
// The choice of the stochastic hill climbing
    algorithm is given by the high
    discontinuity of the score function;
*/

vector<int>
    stochastic_hill_climbing(vector<int>
    Strategy, vector<vector<int> >
    &Opponents_List, double &strategy_score)
{

   //map<vector<int>, double> dictionary;
   strategy_score =
       check_score(Strategy,Opponents_List);
   bool change = true;
   while(change)
   {
      change = false;
      vector<vector<int> > Neighbors =
          find_neighbors(Strategy);
```

```
      random_shuffle ( Neighbors.begin(),
          Neighbors.end() ); //we mix the
          order of strategies close to A
      for(int i=0;i<Neighbors.size() &&
          change==false;i++)
      {
         double neighbor_value =
             check_score(Neighbors[i],Opponents_List);
         if(strategy_score<neighbor_value)
         {
            Strategy = Neighbors[i];
            strategy_score = neighbor_value;
            change = true;
         }
      }
   }
   return Strategy;
}

////////////////////////////////
```

### B. Check Victory

```
////////////////////////////////

/* opponents_group generates a set of
    allocations
based on an initial structure and adding some
    random modifications;
The initial structure is given as input;
*/

vector<vector<int> > opponents_group(int
    n,vector<int> &Base,int low, int high)
{
   int missing_soldiers=100;
   for(int i=0;i<Base.size();i++)
   {
      missing_soldiers -= Base[i];
   }
   vector<vector<int> > Opponents;
   for(int i=0;i<n; i++)
   {
      vector<int> Base_opponent = Base;
      int cnt = missing_soldiers;
      while(cnt>0)
      {
         int k = rand_int(low,high);
         if(k<5)
         {
            int l = rand_int(0,min(cnt,70 -
                Base_opponent[k]));
            Base_opponent[k] += l;
            cnt          -= l;
         }
         else if(k>7)
         {
            int l = rand_int(0,min(cnt,10 -
                Base_opponent[k]));
            Base_opponent[k] += l;
            cnt          -= l;
         }
         else
         {
            int l = rand_int(0,min(cnt,25 -
                Base_opponent[k]));
            Base_opponent[k] += l;
            cnt          -= l;
         }
      }
      Opponents.push_back(Base_opponent);

   return Opponents;
}

////////////////////////////////
```

### C. Check Victory

```
////////////////////////////////

/* Function which establishes the score on a
    match
  given two configurations of the troops in
     the castles;


*/

int check_victory(vector<int> &A, vector<int>
    &B)
{  //function that calculates the result of
    the first player in a 1vs1
   int win_strk_1=0;
   int win_strk_2=0;
   int point_1=0;
   int point_2=0;
   int i=0;
   while (win_strk_1<3 && win_strk_2<3 &&
       i<10){
      if(A[i]>B[i]){
         ++win_strk_1;
         point_1 += i+1;
         win_strk_2=0;
      }
      else if(A[i]<B[i]){
         ++win_strk_2;
         point_2 += i+1;
         win_strk_1=0;
      }
      else{
         win_strk_1=0;
         win_strk_2=0;
      }
      i++;
   } //end of while we go out if we have a
       streak or if we have already checked
       all the castles
   if (i<10){
      if(win_strk_1==3){
         for(int j=i+1; j<=10;j++){
            point_1 += j;
         }
      }
      else if(win_strk_2==3){
         for(int j=i+1; j<=10;j++){
            point_2 += j;
         }
```

```
      }
      else cout << "Error in implementation,
          need to debug!"<< endl;

   }

   return point_1;

}

////////////////////////////////
```

### D. Sorting Algorithm

```
////////////////////////////////
/*
Function used to sort vectors A based on rule
    based on second vector B;
It is used to sort strategies based on score;
*/
void special_sort(vector<vector<int> > &A,
    vector<double> &B)
{  // function that takes as input 2 vectors
    of the same length and that sorts both
    through the values of the second
   vector<Mixed_vec> C (A.size());
   for(int i=0;i<A.size();i++)
   {
      for(int j=0;j<10;j++)
      {
         C[i].first[j] = A[i][j];
      }
      C[i].second = B[i];
   }
   sort(C.begin(), C.end(),
       [](const Mixed_vec& l, const
           Mixed_vec& r)
       {
       return l.second < r.second;
       }
      );
   for(int i=0;i<A.size();i++)
   {
      for(int j=0;j<10;j++)
      {
        A[i][j] = C[i].first[j] ;
      }
      B[i] = C[i].second;
   }

}
////////////////////////////////
```

### E. Check Multi Score

```
////////////////////////////////
/*
//Function that computes the average score
    for a set of Player strategies
  against opponents strategies;
*/
vector<double>
    check_multi_score(vector<vector<int> >
    &A, vector<vector<int> > &Opponents_List)
{  //Function that returns the score of
    several strategies against a group of
    opponents
   vector<double> Scores(A.size());
   for(int j=0;j<A.size();j++)
   {
      long long int score=0;
      for(int i=0;i<Opponents_List.size();i++)
      {
         score +=
             check_victory(A[j],Opponents_List[i]);
      }
      Scores[j] =
          double(score)/Opponents_List.size();
   }

   return Scores;
}

////////////////////////////////
```

### F. Search Maxima

```
////////////////////////////////

/*
This function takes a strategy as input, the
    number of search implemented in the
    heuristic optimization via stochastic
    hill climbing
a vector of vectors containing opponents
    strategies and a minimum target score to
    be obtained by the search for maxima.
Given the high dimentionality of the variable
    space, it is difficult to assess
    convergence of the score of a given
    strategy;

The idea of combining the while and for loop
    arises because either is very difficult
    to find better strategies or you find
    more easily.
*/

void search_maxima(vector<int>
    initial_strategy, int number_restarts,
    vector<vector<int> > &Opponents_List,
    vector<vector<int> > &Maxima, double
    &min_score)
{  //Function that looks for local maxima
    starting from a given point using several
    times (times=number_restarts)
    stochastic_hill_climbing
   int counter = 0;
   while(Maxima.size()==0 &&
       counter<number_restarts)
   {
      ++counter;
      double strategy_score = 0;
      vector<int> Temp_strategy =
          stochastic_hill_climbing(initial_strategy,
          Opponents_List, strategy_score);
```

```
      if(strategy_score > min_score) // If
          the score of the strategy is less
          then the min_score
         {                        //
             Temp_strategy is automatically
             discarded
            Maxima.push_back(Temp_strategy);
         }
   }

   // Print Intermediate result:
   //cout << "\n Local Maximum found :"<<
       Maxima.size() << endl;
   //cout << Maxima.size() << endl;


   for(int i=counter;i<number_restarts;i++)
      {
         double strategy_score = 0;
         vector<int> Temp_strategy =
             stochastic_hill_climbing(initial_strategy,
             Opponents_List, strategy_score);
         if(strategy_score > min_score) // If
             the score of the strategy is
             less then the min_score
         {                        //
             Temp_strategy is automatically
             discarded
            bool existing = false;
            for(int j=0;j<Maxima.size() &&
                existing==false;j++)
            {
               if(Temp_strategy==Maxima[j])
                   existing=true;
            }
            if(existing==false)
                Maxima.push_back(Temp_strategy);
         }

      }

}

////////////////////////////////
```

```
{  //Algorithm that looks for local maxima
    starting from random guessing
   vector<vector<int> > Maxima;
   for(int i=0;i<number_initial_strategies;
       i++)
   {
      vector<int>
          Base_start{5,5,5,5,5,5,5,5,0,0};
      int cnt=60;
      while( cnt>0)
      {
         int k = rand_int(0,9);
         if(k<8)
         {
            int l = rand_int(0,min(cnt,10));
            Base_start[k] += l;
            cnt        -= l;
         }
         else
         {
            int l = rand_int(0,min(cnt,5 -
                Base_start[k]));
            Base_start[k] += l;
            cnt        -= l;
         }


      }


      search_maxima(Base_start,
          number_restarts, Opponents_List,
          Maxima, min_score);

   }

   return Maxima;
}

////////////////////////////////
```

### G. Maxima Detection Algorithm

```
////////////////////////////////

/*
Function which search for local maxima
    starting from a Benchmark configuration
    and adding noise on the initial benchmark
strategies. Then Searches for local maxima
    given set of opponents strategies;
The search for maxima uses the
    search_maxima() function which implements
    the stochastic hill climb algorithm.
*/

vector<vector<int> >
    maxima_detection_algorithm(int
    number_initial_strategies, int
    number_restarts, vector<vector<int> >
    &Opponents_List, double min_score=0)
```