"Solving" the Blotto Game:

A Computational Approach

Michael D. Wittman

April 27, 2011

Abstract. The Blotto game is one of the classic games of game theory. Two generals are competing in a battle over a certain number of battlefields. Each general has to decide how to divide his available troops amongst the fields. On each field, the side that deployed the most troops wins the battle. The general who divides her troops most effectively and wins the most battles wins the game.

The Blotto game has far-reaching implications wherever multi-front division of resources is involved, including business, logistics, and political campaigns, to name a few. However, even a simple version of the game has millions of possible strategies, and it is difficult or impossible to deduce the best strategy on paper.

This paper describes an agent-based simulational approach to identify best strategies in the Blotto game. Using Python, procedures are created to generate random strategies which are tested against a field of other random selections from the strategy space. Well-performing strategies are retained and used to simulate a round-robin Blotto game tournament amongst only the best strategies. Results from the general simulation and from the round-robin tournament are presented and implications are discussed. Finally, directions for future research into the specifics and behavior of the Blotto game are introduced.

1 Definitions and Literature Review

Let us first begin with a definition of the Blotto game as it will be used in the remainder of this paper. There are several different "canonical" forms of the Blotto game, some of which are zero-sum and some of which are not. This non-zero-sum version of the Blotto game involves elements of the games presented in Roberson (2006) and Arad and Rubinstein (2010).

Definition (Blotto Game): Let $x_1 = (x_1^1, \ldots, x_1^n)$ and $x_2 = (x_2^1, \ldots, x_2^n)$ be vectors of length $n \geq 2$. A Blotto game $\mathrm{B}(x_1, x_2, n)$ is a one-time game between two players represented by their strategy vectors x_1 and x_2. The payoff function for player 1 is defined as:

$$p_1(x_1, x_2) = \sum_{i=1}^{n} f_1(x_1, x_2), \text{where} f_1(x_1, x_2) = \begin{cases} 1 & \text{if } x_1^m > x_2^m \\ 0 & \text{if } x_1^m < x_2^m \\ 0.5 & \text{if } x_1^m = x_2^m \end{cases}$$

Player 2's payoff function $p_2(x_1, x_2)$ is simply $n - p_1(x_1, x_2)$. A player is a winner of a Blotto game if her score is higher than her opponent's. If $p_1(x_1, x_2) = p_2(x_1, x_2) = n/2$, then the game is considered a draw.

The first instances of the Blotto game in the game theory literature came as early as Borel (1921), who described a class of games involving a resource allocation decision over a finite number of fields. These games were refined three decades later in a paper by Gross and Wagner (1950) of the RAND Corporation, who introduced the fictitous "Colonel Blotto" after whom this class of games would later be named.

While the Blotto game was introduced into the literature very early in the development of game theory and even earlier than the famous Prisoner's Dilemma game, it has received considerably less attention than its well-known cousin. This relative neglect in relation to the vast literature surrounding the Prisoner's Dilemma may have been caused by the complexities of the Blotto game and the difficulties inherent in finding its solution. As opposed to the simplest form of the Prisoner's Dilemma game, which contains only two strategies and an easy to analyze 2x2 payoff matrix, even very simple versions of the Colonel Blotto game have much more complex strategy spaces. Table 1 shows the number of possible strategies for some selected simple versions of the Blotto game.

Troops	Fields	Number of Strategies
6	2	7
12	3	91
24	4	2925
50	5	316251
120	6	more than 250 million

Table 1: Number of unique Blotto Strategies for selected parameter values

Table 1 illustrates the complexities of analyzing the Blotto game discretely. Numbers of possible strategies increase factorially as field size increase, and even games with relatively few numbers of fields can still engender payoff matrices that are many thousands of columns and rows wide. Furthermore, Roberson (2006), Tofias (2007), and others have proved that the Blotto game has no Nash Equilibrium in pure strategies. Given any strategy vector and her opponent's strategy, a player can always rearrange her troop assignments to improve her score and win the game. When compared to the simplicity of the Prisoner's Dilemma game and its easy to find equilibria, it is no surprise that the more complex Blotto game was relegated to the game theory sidelines for a number of years.

In fact, the specific equilibrium strategies and states of the Blotto game were left unsolved for 85 years after Borel's first description of the game, and were often seen as unsolvable for either the continuous or discrete case. However, a breakthrough paper by Roberson (2006) successfully found a continuous equilibrium strategy distribution for Blotto games of any size as a function of field size and the total amount of troops available.

Roberson's findings spurred new interest in the Blotto game, especially in the newly popular field of experiemental economics. A number of papers realized the potential for an experimental analysis of the Blotto game and created studies that recruited students, academics, and business professionals to submit strategies for use in a Blotto game tournament. Chowdhury et al. (2009) presented one of the first experimental studies of the Blotto game. The Chowdhury game featured asymmetric allocation of resources and found a consistent left-skewness of troop allocation in the rightmost field. That is, the experimental results showed that humans are more likely to place fewer troops on the last field than the other fields, on average.

More relevant to this paper is the experiment created by Modzelewski et al. (2009), which had participants play a repeated 50-troop 5-field Blotto game against a random selection of other players. The authors of this study found the same bias towards the first four fields as did Chowdhury et al. Modzelewski et al., playing off of Schelling's theory of "focal points" theory, suggests that this bias is due to their test subjects being readers of a left-to-right language system. This study also showed that many of the most popular strategies

were "Big 3" strategies that abandoned two fields in order to stack resources on the remaining three fields in an attempt to win the game. The equal split (10, 10, 10, 10, 10) strategy was another popular choice.

The author participated in one such Blotto tournament created by the Israeli game theorist Ariel Rubinstein and hosted on his gametheory.tau.ac.il homepage. The subsequent paper created from this study, Arad and Rubenstein (2010), serves as the inspiration for this project. The Arad-Rubinstein paper is valuable to a student of computational economics since it is the only experimental paper that offers the precise tournament scores and results for specific strategies in a round-robin Blotto tournament.

In 2008, Arad and Rubinstein commissioned a Blotto tournament using a 120-troop, 6-field model specification. Participants were given instructions containing the rules of the game and an online form through which to enter a strategy selection. Two groups of participants submitted strategies for the game: game theory students from a variety of countries and readers of the Israeli business magazine *Calcalist*. Once strategies were collected from all groups, round-robin tournaments were run using the sets of strategies (each strategy played each other strategy in the set in a Blotto game exactly once) and average scores were tabulated. Strategies were then ranked by average tournament scores. Figure 1 shows the Arad-Rubinstein (2010) results:

Classes' grand tournament								Calcalist tournament							
	1	2	3	4	5	6	Mean Score		1	2	3	4	5	6	Mean Score
1	**2**	**31**	**31**	**31**	**23**	**2**	3.83	1	**2**	**31**	**31**	**31**	**23**	**2**	3.77
2	3	31	31	31	21	3	3.80	2	2	32	31	31	22	2	3.76
3	3	31	3	31	31	21	3.76	3	2	23	31	31	31	2	3.75
4	**1**	**31**	**31**	**31**	**25**	**1**	3.76	4	1	1	32	32	32	22	3.72
5	**2**	**27**	**31**	**31**	**27**	**2**	3.75	5	**1**	**1**	**31**	**31**	**31**	**25**	3.71
6	2	31	23	31	31	2	3.74	6	**2**	**27**	**31**	**31**	**27**	**2**	3.71
7	**1**	**1**	**31**	**31**	**31**	**25**	3.73	7	2	31	1	31	31	24	3.70
8	2	21	32	32	2	31	3.72	8	**1**	**31**	**31**	**31**	**25**	**1**	3.70
9	1	1	31	31	25	31	3.71	9	1	25	31	31	31	1	3.69
10	1	31	31	25	31	1	3.69	10	1	1	34	31	31	22	3.69

Figure 1: The Arad and Rubinstein (2010) Blotto experimental tournament results

Figure 1 highlights a surprising result. In both round-robin tournaments, the same strategy vector (2, 31, 31, 31, 23, 2) had the highest average score, and permutations of this strategy hold other spots on the top-ten list. Why was (2, 31, 31, 31, 23, 2) a winning strategy? Why didn't the most intuitive strategy, the

n-way split (20, 20, 20, 20, 20, 20) appear on the list? Was there something that the best strategies in a round-robin Blotto tournament have in common? These were some questions for which a robust agent-based computer simulation model might be able to supply some answers.

There have been several attempts to create a computer simulation of the Blotto game. Tofias (2007) uses a simulation to generate sets of best strategies for Blotto games in which there is asymmetry between the amount of information available to both players. A Tofias agent generates strategies by using a so-called "genetic algorithm" to create strategies that are best responses to the strategy set being generated by its opponent. Tofias's results show that as the number of fields in a Blotto game increase, it takes longer for the agent with more resources to gain an edge in the repeated game. Yet Tofias and a later paper by Jackson (2010) both struggle with defining appropriate parameters for inclusion of a given strategy into the set of "best strategies."

To my knowledge, this is the first paper that attempts to verify a set of experimental data using simulational techniques. This is surprising, since agent-based modeling seems to be an ideal partner for experimental game theory. If we can correctly use simulational techniques to model human behavior in situations with very well-defined rules and outcomes, perhaps future researchers will be able to use agent-based modeling to solve and model more complex human interactions.

2 A Simulational Approach to the Blotto Game

2.1 Why a Simulational Approach?

In comparison to other games of game theory, the Blotto game is a particularly strong candidate for exploration through a simulational approach. In part, this is due to the difficulty of modeling the game discretely and the sheer number of strategies possible in a Blotto game. Even the solutions to the continuous Blotto game as solved by Roberson (2006) are of little use when looking at an empirical or experimental case such as the Arad-Rubinstein (2010) results.

While humans may have trouble analyzing many thousands of strategies in order to find the best strategies, this is easy work for a computer simulation. Fortunately, there exists a easy-to-learn yet powerful language that makes this type of simulation relatively easy. As Isaac (2008) suggests, Python's flexible data structures and relative ease of coding make the language well-suited for the simulation of evolutionary games. Using Python's mutable list structure, we are easily able to keep track of strategies, collect best strategies and report scores after many runs of the Blotto game, to name just a few conveniences. Additionally, by

using an agent-based framework for our simulation, we can attempt to simulate human performance in a Blotto tournament and replicate the Arad-Rubinstein experimental results with a computer.

2.2. Goals of the Project

The goal of this project is to create a computerized model that can simulate round-robin Blotto tournaments using a variety of input parameters in order to see which strategies are most successful in different Blotto games. To do this, we must first devise a way to generate integer-valued Blotto strategy vectors from scratch. Given the unintuitive strategy that was the winner of the Arad-Rubinstein tournament, it should not be left up to the programmer to come up with strategies to test, particularly in complex games. Once a set of strategies is created, it is relatively simple to run a round-robin Blotto tournament. A tournament involves each strategy in the strategy set playing each other strategy in the set, as well as itself, in a Blotto game. From there, results can be collected and patterns can be analyzed.

However, with the ultimate goal of replicating and explaining the Arad-Rubinstein winning strategy, simply generating strategy vectors at random is not sufficient. Even though humans may not analyze every single strategy in the Blotto game before they make their strategy choice, experimental results show us that intuition about the structure of the game is usually enough to discount some strategies immediately. For example, few humans will choose a strategy that concentrates all resources in one field in a multi-field game, as this is a sure-fire loss. Yet when a computer generates random strategies, it is unable to use the same intuition to discount the bad strategies and choose the good ones.

Hence, another major goal of the project is to create selection criteria that allow a computer to separate good strategies (i.e., the ones that humans are more likely to play) from bad ones. By using the computer to generate a set of only good strategies, we can more closely mimic the strategy sets that arise when humans play the game. In this way, we can achieve a better understanding of why the Arad-Rubinstein winning strategy was the victor across two distinct Blotto tournaments.

3 Simulating Blotto Tournaments

3.1. Generating random strategies

One of our first challenges when creating a model of the Blotto game is deciding how to generate random integer-valued strategy vectors from scratch that can be used in the game for a certain set of parameters. These strategy vectors have two properties: they must sum to the total amount of troops or resources (i.e.,

no strategy can use more troops than assigned and no strategy can leave troops unused) and they must be of length n, where n is the number of fields in the given game. The generation of strategies was one of the first computational challenges of the project, thanks to two distinct frames of thought about what a truly "random" strategy vector is.

One way to generate random strategy vectors is to start with a generator function that can create, one at a time, permutations of strategies of length n that sum to the total number of resources available. The generator can then be called by a list to fill that list with every possible strategy vector in the strategy space. A function like Python's random.choice() procedure can then be used to select a strategy out of the list randomly as needed.

This approach has several benefits. By generating the whole space of possible strategies and using random.choice(), we are guaranteed that our selection of a random strategy vector will be uniformly "random" across every strategy in the space (of course, we are bounded by the constraints of "psuedo-randomness" when using computers to generate random events). We can also easily search among all strategies for a given set of parameters in our search to find best strategies for the Blotto game without fear that we might have missed a potentially good strategy due to random error. It is also straightforward to create an ordered list of all strategies to rank the performance of certain strategies against all others.

However, the convenience and thoroughness of the generator approach is undermined by large losses in computational efficiency, especially when dealing with large strategy spaces. Even if we only want a small subset of strategies, we must generate the entire strategy space before we can extract our desired output. In a 120-troop 6-field model like the Arad-Rubinstein game, this means that to generate even one random strategy, we must wait for the system to collect all 250 million strategies in a list. This makes the analysis of even moderately large Blotto games computationally inefficient and begs for a way to generate random strategies on demand.

The code excerpt below shows a procedure that can be used to generate an integer-valued strategy vector of length n that sums to an initial value init on demand.

```
def RandoStrat(init, n):
    """Create a single random integer-valued strategy that sums
    to init with length n"""
    res = init        #create a temporary resource variable
    strat = []        #empty list
    for i in range(n-1):
        rand = random.randint(0, res) #fill each position in the vector
```

```
        # with a random int from 0 to the remaining reserves
        strat.append(rand)              #add this to the strategy vector
        res -= rand                     #reduce reserves by this number
    strat.append(res) #add any remaining reserves as the last variable
    random.shuffle(strat) #shuffle the strategy vector
    return strat
```

This "on demand" procedure for generating strategies makes simulating tournaments with large strategy spaces easy. This procedure can be called any number of times to generate the desired number of random strategies without having to generate the entire strategy soace first. The random nature of the selection process may also include some stochastic behavior to strategy selection that is not present when we generate the entire strategy space. This may be beneficial if we assume that humans do not always choose the best strategies and sometimes make mistakes. Hence, creating our best strategy set with this procedure might be a better approximation of the experimental results.

However, for these results to be robust, this procedure should not be biased towards returning strategies with certain characteristics over others. From looking at the procedure, it is unclear if RandoStrat will indeed produce strategies that are as uniformly distributed as the generator procedure. While aggregate simulational results from best strategies produced with the RandoStrat procedure are similar to those from the generator procedure, testing of the RandoStrat procedure reveals an overrepresentation bias towards strategies that place a large majority of the available resources in one field. These strategies will later be culled out when we select best strategies, so while this bias may affect the specific magnitude of tournament scores, it does not affect the general results we present later.

The following procedure offers another way to generate a list of w strategies with given parameters on demand. Testing of this procedure shows that individual strategies are uniformly distributed and do not show the selection bias that plagues the original RandoStrat procedure.

```
def RandoStrat2(init, n, w):
"""Generate w random strategies of length n that sum to init"""
    strats = []
    potential = []
    while len(strats) < w: #keep going until we put w strategies in strats
        for i in range(n):
            potential.append(random.randint(0, init)) #fill the
        #potential strategy with three random integers
        if sum(potential) == init: #check if they sum to init
                strats.append(potential) #if so, add them to the strategy list
                potential = [] #and clear the potential strategy
```

```
        else:
            potential = [] #otherwise, clear the potential and try again
    return strats
```

While this procedure does not encounter the same bias as the original RandoStrat procedure, it is highly inefficient since it relies on a "guess and check" method of strategy selection. This makes RandoStrat2 especially slow when dealing with large strategy spaces. A further extension of this project might include a procedure of generating random strategies that contains a uniform distribution among strategies in the strategy space while not becoming computationally inefficient when given large parameter values.

3.2. Running the Blotto Tournament

Once we have a set of strategies that we have generated from either of our two methods above, we must then run the Blotto tournament. Fortunately, Python's flexible data structures and mutable lists make running a Blotto game rather trivial. The procedure below implements the Blotto game defined in Section 1 and returns the score of the strategy designated strat1. The opponent's score can be derived by subtracting strat1's score from n, the total number of fields.

```
def BlottoGame(strat1, strat2, n):
        """Plays one Blotto game among strategy vectors strat1
        and strat2 of length n. Returns: score for strat1
        """

        p1score = p2score = 0 #initialize scores
        assert len(strat1) == len(strat2) #make sure that vectors are
        #the same length
        for i in range(n): #check each field and assign scores
                if strat1[i] > strat2[i]:
                        p1score += 1.0
                elif strat1[i] < strat2[i]:
                        p2score += 1.0
                else:
                        p1score += 0.5
                        p2score += 0.5
        return p1score  #return the score of the first strategy vector
```

If we have a list of strategy vectors (a "strategy set"), we can run a round-robin Blotto Tournament amongst all of the strategies in the set by simply using nested for-loops to run a Blotto game for each strategy against every other strategy in the list, including itself. A list keeps track of the score history for the strategy in the tournament. We can then take the average of the scores in the history list to find the strategy's average tournament score in the round-robin tournament. This result is then appended to another

list that can be cross-indexed with the strategy set to find the round-robin tournament score for a specific strategy.

Once we have run a Blotto tournament, it is helpful to sort strategies in order of their scores in the tournament to find out which strategies perfomed the best. Python's tuple format provides a convenient way to sort strategies in order of tournament score while still maintaining their indexing information.

```
def SortScores(n, stratset):
        """Returns a list of tuples of the form
       (scores, index) for each strategy
        in stratset, sorted from best score -> worst score
        """
        score_tuples = []
        for i in range(len(stratset)):
                score_tuples.append((stratset[i], i))
       #append a tuple of the score and the index of that score
        score_tuples = sorted(score_tuples, reverse=True)
       #sort the scores from best to worst (keeping indices intact)
        return score_tuples
```

These tuples can be unpacked in a later procedure to create a list of strategies that are sorted by their performance in the tournament from best to worst.

Now that our strategies are sorted, it might be desirable to create a list of only the best performing strategies and run a round-robin tournament against these best strategies only. Since the experimental results tell us that humans are able to use intuition to avoid very high-variance "bad" strategies, performing a tournament using only the best strategies (however we define what a "best strategy" is) is likely to more closely approximate an experimental Blotto tournament. It is a worthwhile exercise to see if the strategies that perform best against all strategies are the same as the strategies that perform best against only the best strategies (i.e., only the human strategies). This latter type of strategy will be more likely to do well in an experiemental Blotto tournament.

After we have our tournament scores, we can report our results. This computational model reports strategies along with two tournamant results: T-Score and B-Score. T-Score (or Total Score) represents the score of a strategy in a round-robin tournament against a representative sample of all strategies in the space. For small strategy spaces, T-Score is calculated by running a round-robin tournament against all strategies in the space. For large parameter values, T-Score is calculated as the score of a Blotto tournament against 10,000 random strategies that satisfy the parameters.

B-Score, on the other hand, is the score of the strategy in a round-robin tournament against only the

best strategies. There are a variety of ways to define best strategies. In a simple approach, the top third (for instance) of all strategies sorted by T-Score are classified as best strategies and stored in a list. We can then run a round-robin tournament using only the strategies in this list. The average scores of each strategy in this tournament is the B-Score. The differentiation between T-Score and B-Score is meant to signal the divide between tournament performance in a set of random strategies and performance in a set of strategies that would more closely mimic experimental results. As we will soon see, strategies that do well against random strategies (i.e., strategies that have a high T-Score) do not necessarily do well against the best strategy set.

Strategies are also reported with their variance (Var). Var is a standard variance calculation that measures the dispersion of the troops amongst the different fields. A strategy that equally divides all troops across all available fields would have zero variance, for instance. We claim that humans are more likely to avoid extremely high-variance strategies, so strategy sets that attempt to mimic experimental results should also be weighted towards more low-variance strategies than high-variance strategies.

3.3. Collecting best strategies using a learning agent

The goal of creating a strategy set that would closely mimic those created in experimental results with human players offered an interesting challenge. What would be the best way to identify the best strategies out of a pool of random strategies from a big strategy space? The ideal strategy set would contain many low-variance strategies with a few high-variance strategies mixed in, to best mimic the way that humans play the Blotto game. We can create such a strategy set though the creation of an agent that can learn strategies from its opponents and keep those strategies that perform well.

For this approach, we give an agent a random strategy to start. Then the agent plays Blotto games against other random strategies. If the agent is beaten by its opponent, it discards its current strategy and starts using its opponent's strategy. The agent also keeps track of how many games a certain strategy wins in a row. If a strategy passes a threshold parameter of number of consecutive games won before being discarded, it is added to a list of best strategies. The agent can be told to play Blotto games until it produces a pre-determined number of best strategies with a given selection criterion.

The strategy set that is produced by this learning agent has several desirable properties. Since low-variance strategies are more likely to win many games in a row than high variance strategies, the set will be biased towards low-variance strategies if we have strict selection criteria. However, there will always be a few high-variance strategies that "get lucky" and win enough games in a row to be added to the set. This produces a set that is similar in theory to a human-produced experimental result. Furthermore, by changinig

the strength of the selection criterion, we can obtain strategy sets with different proportions of high-variance strategies to low-variance strategies. This sort of manipulation will prove to be very useful in uncovering why a high-variance strategy like (2, 31, 31, 31, 23, 2) performs so well in experimental Blotto tournaments.

4 Results

4.1. Generator function results

Our first experiments with the Blotto tournament simulator involved small parameter values for troop and field sizes. These tournaments were run using the generator function described earlier that generated a list of all strategies in a certain strategy space. In this way, we could test strategies against all others without fears of omitting or overrepresenting certain strategies. These could be seen as the baseline results of a small Blotto tournament with a limited number of strategies.

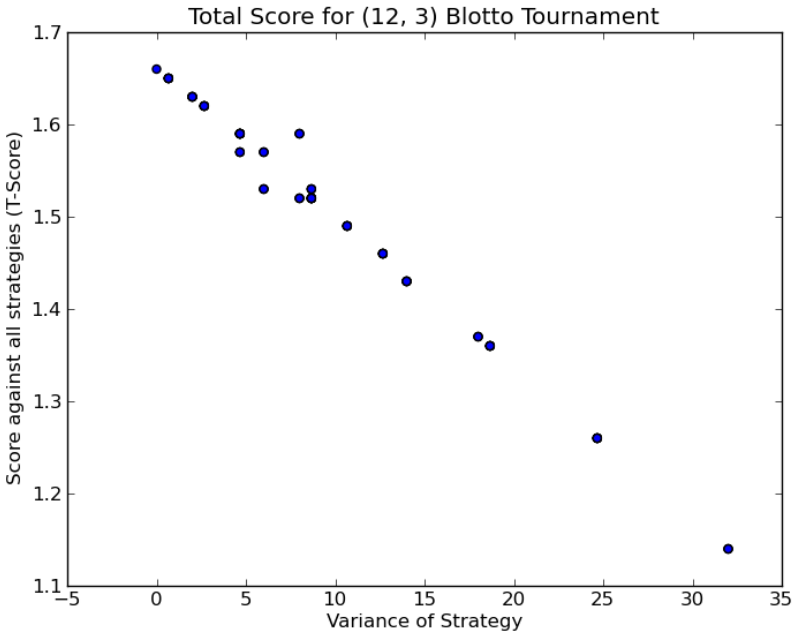

Figure 2: T-Score on Variance: 12 troop, 3 field Blotto Game with Generator

Figures 2 and 3 show the results from the simulation of a round-robin Blotto tournament with 12 troops divided amongst 3 fields. Figure 2 shows T-Score plotted against variance. In this case, T-Score is the strategy's average score in a round-robin tournament against all strategies in the space. Figure 3 shows B-Score plotted against variance, where B-Score is the strategy's round-robin tournament score against "best

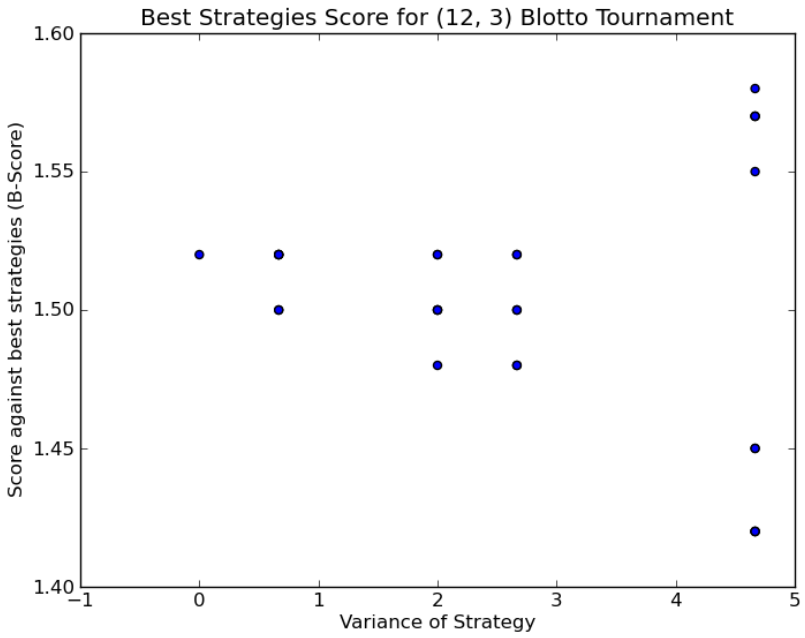

Figure 3: B-Score on Variance: 12 troop, 3 field Blotto Game with Generator

strategies." In this case, best strategies are the strategies whose T-Score was among the top third in the overall tournament.

Figure 2 shows a strong linear relationship between variance and performance against a set of all strategies. This is a universal result that we found amongst all strategy spaces, regardless of initial parameter values. This provides a straightforward decision rule for the best strategy in Blotto tournaments against a strategy set that represents a true random sampling of available strategies; to maximize T-Score, choose the strategy with the lowest possible variance. In games in which the number of troops is divisible by the number of fields, the strategy that equally splits all troops across all available fields will result in the highest T-Score.

It is harder to draw conclusive results from Figure 3. The data points seem to form a "trumpet-bell" shape, with low-variance strategies having an average performance against best strategies. As we increase the variance, we find that some strategies do very well (the strategy that ends up winning this best-strategy tournament has one of the highest variances in the set) and some do very poorly. This figure raises more questions than it answers. Why do some high-variance strategies do well and some do poorly? Is there a way to predict which of the high-variance strategies will do well *a priori*? Furthermore, this strategy set does not accurately represent the Arad-Rubinstein experimental set since it assumes that humans have a degree

of rationality sufficient enough to only select among the best-performing third of all strategies. To analyze the experimental results, we must rely on the strategy sets provided by our learning agent.

4.2 Learning agent results

In order to ensure that our learning agent provides robust results, we must first ensure that the data from small-parameter Blotto tournaments matches that of the generator function. Figures 4 and 5 show the simulation results from a group of 200 best strategies produced by a learning agent in a 60 troop, 3 field Blotto game. Strategies needed to win 20 games in a row against random strategies to be included in the list of best strategies.

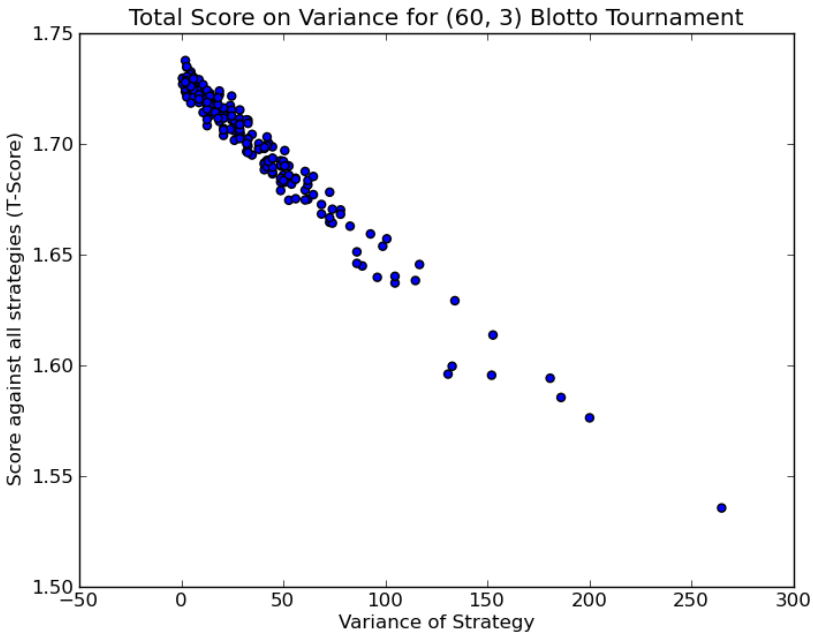

Figure 4: T-Score on Variance: 60 troop, 3 field Blotto Game with learning agent, 20-game inclusion criterion

Figure 4 shows that a learning agent produces the same linear relationship between T-Score and variance that we saw when we ran the simluation using a generator function. The B-Score plot in Figure 5 also appears to maintain the same "trumpet-bell" shape, with low-variance strategies performing at an average level, some high-strategies performing well, and some high-variance strategies performing very poorly. However, using a learning agent gives us flexibility to adjust selection criteria to create best-strategy sets with different properties. Figure 6 shows the B-Score graph from a simulation similar to the simulation from Figure 5, except the number of fields in the game has been doubled from 3 in Figure 5 to 6 in Figure 6.

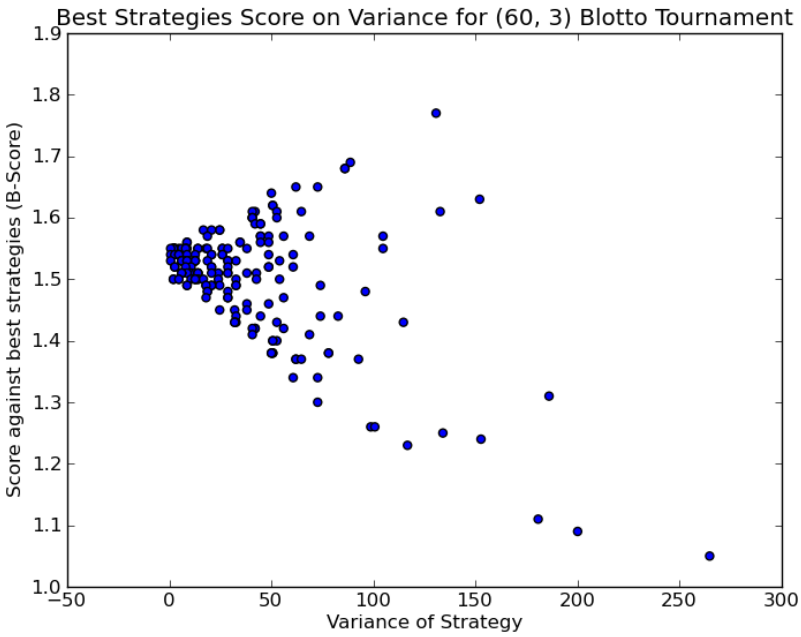

Figure 5: B-Score on Variance: 60 troop, 3 field Blotto Game with learning agent, 20-game inclusion criterion

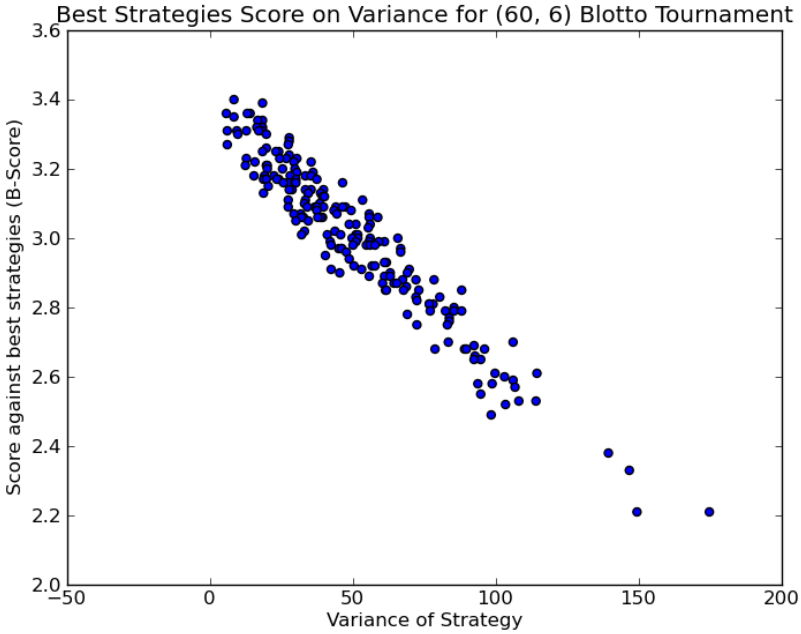

Figure 6: B-Score on Variance: 60 troop, 6 field Blotto Game with learning agent, 20-game inclusion criterion

In Figure 6, the scores of a tournament run with only the best strategies seems to mimic the linear relationship that we obtained when we ran a tournament across a random sampling of *all* strategies (the T-Score graph). This suggests that our set of best strategies in Figure 6 has a similar makeup to a pool of random strategies taken from the strategy space. Therefore, we need a metric that measure the makeup of a best strategy set to see if it contains a mix of high-variance and low-variance strategies (i.e., it behaves like a random set of strategies) or if it contains mostly low-variance strategies (i.e., a set that is more close to the experimental results).

4.3. Strategy Set Variance

The Strategy Set Variance (SSV), the average of the variances of the strategies in a strategy set, is one way to measure the strategic makeup of a given set.

Definition (Strategy Set Variance): Let $S = \{s_1, \ldots, s_n\}$ be a strategy set, and let σ_i represent the variance of strategy i. Then

$$SSV(S) = \frac{1}{n} \sum_{i=1}^{n} \sigma_i$$

Given two strategy sets, the set with the lower SSV will likely contain a higher proportion of low-variance strategies. We can raise or lower the SSV of the best strategy set created by the learning agent simulation by adjusting various parameter values. All else being held equal, we can lower the SSV of a strategy set by:

- Decreasing the number of fields available
- Decreaseing the amount of troops/resources available
- Strengthening the selection criterion (e.g., require a higher winning streak for addition to the strategy space)

Therefore, by adjusting these parameters, we can perform experiments to see how specific strategies perform in round-robin Blotto tournaments against strategy sets containing mostly random high-variance strategies and strategy sets containing mostly strong, low-variance strategies.

4.4. Testing the Arad-Rubinstein (2010) result

We now arrive at a procedure we can use to attempt to answer our original question: why did the strategy (2, 31, 31, 31, 23, 2) do so well in the Arad-Rubinstein (2010) experimental results? We claim that

since humans are more likely to avoid extremely high-variance strategies, the experimental strategy set of Arad-Rubinstein will likely have a low SSV. Therefore, as we decrease the SSV of a simulated strategy set, we should see the (2, 31, 31, 31, 23, 2) strategy perform better. Since we wish to keep the troop and field parameters constant throughout the experiment, we can adjust the SSV of our simulated best strategy sets by altering the selection criterion.

Table 2 shows the results of 120-troop 6-field Blotto tournaments with a variety of selection criteria. We first generate 100 best strategies using a learning agent and the given selection criterion. Then the strategy (2, 31, 31, 31, 23, 2) is introduced into the set of best strategies. We then run a round-robin Blotto tournament using this strategy set and report the SSV of the strategy set, the B-Score of the (2, 31, 31, 31, 23, 2) strategy, and its rank in the tournament out of the 101 strategies in the set.

Selection Criterion	SSV	B-Score	Rank (out of 101 strategies)
5 games won	282	3.28	13th
20 games won	185	3.09	32nd
30 games won	161	3.19	6th
50 games won	129	3.20	7th
100 games won	86	3.26	1st
150 games won	71	3.47	1st

Table 2: Performance of (2, 31, 31, 31, 23, 2) in various Blotto tournaments
Note: random.seed(111111)

As Table 2 shows, strengthening the selection criterion leads to strategy sets with lower SSV values. When the Arad-Rubinstein strategy is introduced into these strategy sets, it generally does very well as the SSV is lowered. In Blotto tournaments with selection criteria of over 100 games won, the Arad-Rubinstein strategy wins nearly every time. This offers a potential explanation for why (2, 31, 31, 31, 23, 2) was the winning strategy in the experimental tournament. Furthermore, since the experimental B-Score for the Arad-Rubinstein strategy (3.81) was higher than the simulated B-Score for the 150 games won strategy set (3.47), this suggests that the experimental strategy set might have had an even lower SSV than the simulated models presented here.

5 Future extensions

This project offers many opportunities for future research to extend and confirm the results presented here and to explore more of the details of the Blotto game. While the simulational approach to the Blotto game seems to offer a potential explanation for why some strategies fare better than others in Blotto tournaments, it offers no explanation for why some high-variance strategies do better than others.

For example, (2, 31, 31, 31, 23, 2) is clearly an above-average strategy; it ranks among the top third of strategies in any Blotto tournament that we simulated. What features of this strategy cause it to perform so well when compared to other similarly high-variance strategies? More importantly, is there another strategy that can be found to be more successful than the Arad-Rubinstein result in the 120 troop, 6 field game? Could the properties of the Arad-Rubinstein result be used to generalize a model that predicts the most successful strategies for any set of parameters? Any of these questions would be a worthwhile avenue of research.

Tofias (2007) suggests that giving an element of bounded rationality to the agents in the simulation might be a worthwhile addition. Tofias claims that humans playing the games have some intuition about which strategies are the best, but are unable to rank them in order like a computer could. Therefore, a procedure which could mimic this bounded rationality by allowing agents to pick from a set of predetermined best strategies with a weighted probability distribution might more closely mirror experimental results.

Finally, along with tweaks to the program to improve computational efficiency, future researchers might want to start to explore the ideas of k-level human reasoning that seem to interest Arad and Rubinstein. When humans play the Blotto game, they often attempt to think one or more moves "ahead" of their opponent, reacting preemptively to perceived threats. A possible extension of this model would include agents with k-level reasoning capabilities who are somehow able to form predictions about their opponent's moves and select strategies that are best responses to their guesses of their opponent's strategies. Modeling these interactions is a promising and exciting future for agent-based modeling. Using ABM to explain and model research in the emerging field of experimental game theory seems to be a fruitful partnership that would help reveal the ways that humans act in well-defined strategic situations.

References

[1] Arad, Ayala and Ariel Rubinstein. 2010. "Colonel Blotto's Top Secret Files: Multidimensional Iterative Reasoning in Action." Working Paper.

[2] Borel, Émile. 1921. "La theorie du jeu les equations integrales a noyau symetrique." *Comptes Rendus de lAcademie* 173: 1304-1308. English translation by Savage, L (1953), The theory of play and integral equations with skew symmetric kernels, *Econometrica* 21.1: 97-100.

[3] Chowdhury, Subhasish, Dan Kovenock, and Roman M. Sheremeta. 2009. "An Experimental Investigation of Colonel Blotto Games." CESifo Working Paper Series No. 2688.

[4] Gross O. A. and R.A. Wagner. 1950. A Continuous Colonel Blotto Game. Technical Report RM-408. *RAND Corporation*, Santa Monica, CA.

[5] Isaac, Alan. 2008. "Simulating Evolutionary Games: A Python-Based Introduction." *Journal of Artificial Societies and Social Simulation* 11.3: Article 8.

[6] Jackson, Elais. 2010. "Adaptation and Learning in Blotto Games." Working Paper.

[7] Modzelewski, Kevin, Joel Stein and Julia Yu. 2009. "An Experimental Study of Classic Colonel Blotto Games." Working Paper.

[8] Roberson, Brian. 2006."The Colonel Blotto game." *Economic Theory* 29: 1-24.

[9] Tofias, Michael. 2007. "Of Colonels and Generals: Understanding Asymmetry in the Colonel Blotto Game." Working Paper.