Writing an AI to Increase the Odds of Winning Three Card War

Jesse Johnson
Brigham Young University
Provo, Utah, United States
jwj39@byu.edu

***Abstract*—This paper describes an algorithm to increase the odds of winning during a game of Three Card War, which is a variation on the popular card game War. The algorithm sees all possible outcomes from playing a specific card and gives a recommendation on which card to play. The code was all in Java, and the algorithm was run 100,000 times to verify its success rate.**

***Index Terms*—java, min-max, algorithms, programming, games**

I. Introduction

This paper explores the application of Linear Programming in solving what move should be taken in a physical card game. Specifically, this game is Three Card War, where Two Players play a game of War but opposed to the entire game being random and determined by the shuffling of a deck of cards, there is now a layer of strategy to the game. Now both players have three cards in their hands and can pick one of them to play against the opposing player. First, we will explore how an AI chooses to make a move in a game where they are playing against only one opponent and then move on to the applications of using an AI to choose the play in Three Card War.

A. Writing an AI to Play Physical Games

Chess is a game of pure strategy. This allows an AI to be able to plan out all possible outcomes from a board and choose an optimum move according to min-max. According to an algorithm shown on cs.cornell.edu,

"In order to save space and time during the min-max search, its optimal not to have separate board instance at each branch. After all, they differ only by the position of one piece. Hence each move contains information not only about which piece was moved from where to where, but also whether it affected castling, en passant, and whether it captured an enemy piece in the process. Thus reversing a move on a board is very simple. The algorithm thus only needs one board object, on which it makes and reverses all the moves it considers during its search" [1].

This considers the values of each piece and the odds that a piece will be taken. In doing this it finds the solution that will give the best outcome for a given situation not knowing what the opponent will choose but taking that into account anyway. Any game where there are 2 players and an amount of choice can be represented as a min-max situation. The more possible outcomes, the longer it will take and the odds can be vastly different.

II. War

Three Card War is a variation on a game most people have played in their lifetime: War. To understand this variation, one must also understand the original game. War is a card game where a standard deck of 52 cards is shuffled and dealt evenly between two players. The players then will play the top card from their deck, and the winning card is then declared the winner and goes into the players discard that played said card along with the card that it defeated. The value for each card is as follows:

2: 2 points
3: 3 points
4: 4 points
5: 5 points
6: 6 points
7: 7 points
8: 8 points
9: 9 points
10: 10 points
Jack: 11 points
Queen: 12 points
King: 13 points
Ace: 14 points

When two cards have the same value, then a "war" is declared. In a war, both players must do the following. Draw the top three cards from their deck and place them face up. These cards go to whoever is the winner of the "war". Next each player flips the next card from their deck and the winner of these final two cards is the winner of the "war". This process continues until a winner is declared for the war. Once a player's deck is completely used up their personal discard pile will be shuffled together and then the game continues until one player has all 52 cards in their deck. This game is completely random, and a winner is decided by how the decks get shuffled at any point in the game. There is no strategy, and because of this the game gets dull for most players before the game is even finished, let alone playing the game again. Because of the random nature of this game the odds of victory for each player is 50:50.

A. Three Card War

Three card war is a variation of war where the game is played mostly the same way with one huge difference: instead of playing the top card from the players deck, each player now has three cards in their hand and chooses which they want to play against their opponent. The remainder of the rules remain the same, including when war is declared and how a war is carried out. This includes all cards being used in war being taken from the deck and not the hand. This now adds a layer of strategy into a game that was purely chance and now somebody that knows what they are doing is more likely to win a game than a player who does not. This suggests a new issue into the equation of the game however: With my specific hand which card should I choose to play?

III. Writing a Program to Store Game Data

The first part of solving this problem was writing a program that takes the game from a beginning state of each player having been dealt half of a full and shuffled deck of cards and keeps track of when scenarios like war happens and who wins a single battle. This meant I had to create a program that made a deck of cards according to the standards where each card has a value and a suit and follow the game logic for playing a game of Three Card War.

To create a deck the values are listed above, and suits are as follows: Spades, Diamonds, Hearts and Clubs. There are no duplicates in a deck of cards and each there is a card of each suit that contains each value.

After creating the deck of cards, I then needed to write logic for a player of the game and for their opponent. This was done by creating a class for each that held the cards that were in their discard pile and how to shuffle the discard pile once their deck has been depleted. The initial deck for both the player and the opponent is unknown to the game's logic, but once a card has been played it gets added to the winners discard pile.

Following creating the players, I needed to have the game function in the different scenarios. This was accomplished by creating a game loop that asks for what the player draws and plays as well as what the opposing player plays. This is then fed into an algorithm that determines if a war is declared and then asks for what cards are played in the war and automatically adds the cards into the winner's discard pile.

Upon completion of this game logic a rudimentary version of the game now exists. The values are all that is compared at this point, as the program assumes that everything is input exactly as it was played. This is changed later when running the program with determining which card is the best to play. At this point however I needed to test that it indeed worked as expected. I then played the game with a physical deck of cards to ensure the logic was sound and everything went where it was supposed to when it was supposed to go there. Many attempts later I was satisfied with my results (no more exceptions were thrown, and the game worked until completion multiple times in a row). After this logic was created to determine the odds of winning with each card.

IV. Writing the Logic

When playing Three Card War without having a program to track what is played, the odds of winning at a given time with a specific card are fixed. Because of this a player would theoretically choose the card that gives the most certain outcome. This will cause the worst cards to be given to an opponent while they play their mediocre cards, and the player would then have a higher chance of winning with mediocre cards against the opponent later. This shifts the balance in favor of the player if the opponent doesn't also employ a similar strategy. In both of the following scenarios there is also a skew multiplied by the program to make it slightly less likely to play the highest value cards. This is mostly useful in scenarios where multiple cards have the same odds of winning and we would want to choose the lower valued card in that situation. This skew is:

2: 1.2
3: 1.19
4: 1.18
5: 1.175
6: 1.15
7: 1.125
8: 1.1
9: 1.075
10: 1.05
Jack: 1.025
Queen: 1.0
King: 0.975
Ace: 0.95

A. Linear Problem Without Memory

The following is an example of how a typical player would play the game without having memory of what has previously occurred. This is how most people would play the game.

max $c = (50 - \text{abs}(((1 + h - g) / (1 + g - h))((db - hb)/(ds - h))))$ * skew

s.t. $\sum_{v=2}^{c.value}(4 - h(v)) / (ds - h)$

$(4 - h(c.value))/ (ds - h)$

$\sum_{v=c.value}^{14}(4 - h(v)) / / (ds - h)$

key:
- c = card to play
- h = size of hand
- g = size of opponent's hand
- db = number of cards in the deck that c has a greater value than
- hb = number of cards in players hand that c has a greater value than
- ds = deck size, or 52
- v = card value
- h(v) = number of cards in players hand with specified value

This version, which is how the initial logic of the game would be played, does not include any form of memory of what has happened so far in the game, including who has won what. This is also used for all future instances, as this aspect will not be altered for using the different strategies. This is shown in a cost matrix by:

$$a_{ij} = \begin{cases} 1 : if\ i > j \\ -1 : if\ i < j \\ 0 : if\ i = j \end{cases}$$

The corresponding matrix for a hand containing a 9, a 7 and a Jack is as follows:

	2	3	4	5	6	7	8	9	10	Jack	Queen	King	Ace
9	1	1	1	1	1	1	1	0	-1	-1	-1	-1	-1
7	1	1	1	1	1	0	-1	-1	-1	-1	-1	-1	-1
Jack	1	1	1	1	1	1	1	1	1	0	-1	-1	-1

The chance of winning for each card being:

	W	T	L
9	27/49	3/49	19/49
7	20/49	3/49	26/49
Jack	34/49	3/49	12/49

In this case the chosen card would be the Jack as it has the situation where one of its projected outcomes is larger than any other, or its chance of winning being 34/49 which is the largest individual value on the table.

B. Linear Program with Memory

Once memory has been added into the mix it gives a much clearer idea of what card should be played. This is the logic before a single shuffle has occurred:

max c = (40 – abs(((1 + h − g) / (1 + g – h))((db – hb − pdb − odb)/(ds − h − pds − ods)))) * skew

s.t. $\sum_{v=2}^{c.value}(4 - h(v) - od(v) - pd(v))\ /\ (49 - \text{pds} - \text{ods})$

$(4 - h(c.value)\ -\ od(c.value)\ -\ pd(c.value))/\ (49 - \text{pds} - \text{ods})$

$\sum_{v=c.value}^{14}(4 - h(v) - od(v) - pd(v))\ /\ /\ (49 - \text{pds} - \text{ods})$

key:

c = card to play

h = size of hand

g = size of opponent's hand

db = number of cards in the deck that c has a greater value than

hb = number of cards in players hand that c has a greater value than

pdb = number of cards in players discard that c has a greater value than

odb = number of cards in players discard that c has a greater value than

pds = player discard size

ods = opponent discard size

ds = deck size, or 52

v = card value

h(v) = number of cards in players hand with specified value

od(v) = number of cards in opponents discard with specified value

pd(v) = number of cards in players discard with specified value

Let's assume that the game has been played for a little while. There is now a discard for each player. The player discard contains: 5, King, King, 6, 4, 4, Jack, 3, 10, 9, 8, 7, Jack, 9, 8, Ace, 2, Queen and the opponent's discard contains 4, Queen, Queen, King, 3, 6, Jack, 2, 2, 4. Using the same cost matrix as without memory but now having the odds from this later one with memory we get the following matrix. The values present are in accordance with the math above.

	W	T	L
9	12/21	1/21	8/21
7	7/21	2/21	12/21
Jack	16/21	0/21	5/21

In this case we would still choose the Jack, as it has a value of 16/21 to win which is by far the furthest from 40%. This may appear to not be affecting the outcome in the overall game, but when comparing the values to that of the previous example by converting all the winning chances to percent's you see that the odds of winning with the Jack has increased to 76% compared to the 69% previously and the other values have dropped. You also may have noticed that we are now using 40 as the base for determining odds. This is because it makes the games go by a bit faster for recording data as will become evident as to why later.

C. Linear Program with Memory After Shuffling

After the deck is shuffled the program's memory now knows exactly what is contained in the combination of the opponent's deck and their hand, but not what is actually in the hand to play. This once again changes the values as it knows the exact number of cards that the opponent has opposed to just knowing what has not appeared yet. Now the linear problem is actually drastically simplified(especially when compared to before shuffling the deck). In the following example all situations referring to the opponent's deck also includes their hand.

max c = (40 – abs(((1 + h − g) / (1 + g – h))((opdb)/(opds)))) * skew

s.t. $\sum_{v=2}^{c.value}(opdb)$ / (opds)

$(opd(v))$/ (opds)

$\sum_{v=c.value}^{14}(opdl)$ / (opds)

key: c = card to play

h = size of hand

g = size of opponent's hand

v = card value

opdb = number of cards in opponents deck that c has a greater value than

opd(v) = number of cards in opponents deck with specified value

opdb = number of cards in opponents deck that c has a lesser value than

opds = opponent deck size

This time we also know that the player has the following cards in their deck but not their hand: 3, Ace, 5, 7, 10, 7, 9, 2, 6. With this and the known cost matrix we now get the following matrix. The values present are in accordance with the math above.

	W	T	L
9	6/12	0/12	6/12
7	4/12	0/12	8/12
Jack	8/12	0/12	4/12

Once again the odds are decreased when compared to previous scenarios. Although it may look like the 7 would be the card to play here (since all values have to be multiplied by the skew) it actually isn't. This is where using 40 for the base to compare to comes into being needed. This is because games can come to a standstill for many turns (sometimes over 1000) and this decreases the average turn count for the game. This then also skews the game further toward winning instead of just having the most likely outcome.

V. Implementation

I implemented the logic for the player in Java. This was because Java allowed me the best access to creating objects and then mapping and outputting data. It was during this portion that I realized the need to adjust the base value for determining odds to 40 from 50. Games were going on considerably longer, and I was trying to record data and it was just taking too long.

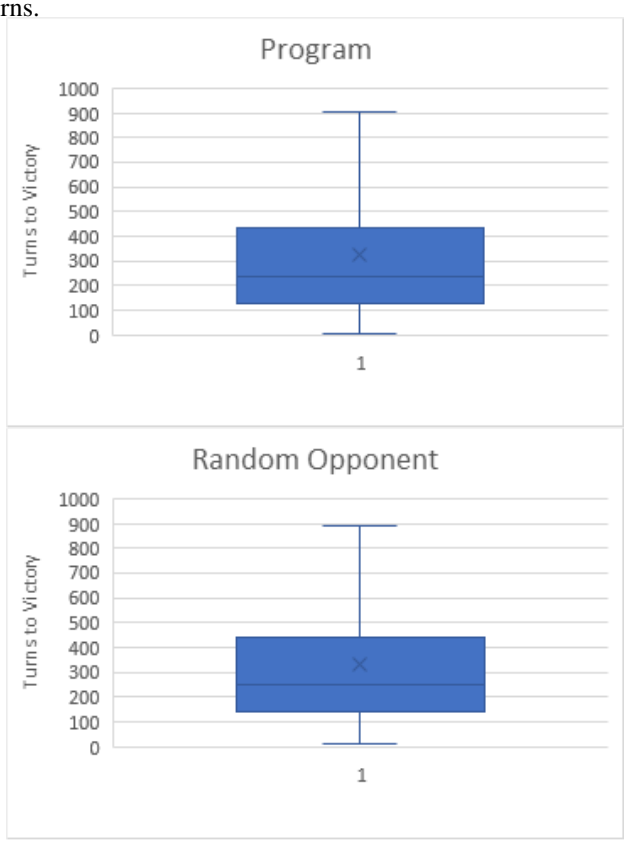
I also found a method to completely automate the entire game from shuffling/dealing to a winner being declared. This was done in 2 different circumstances, both against a typical opponent who would just play a random card from their hand as well as against an opponent attempting to win using the exact same method. This automation allowed for many attempts of the game to be run now that all logic was intact.

This was all automated with an "unknown" deck of cards being given to each player, both being half of a randomly shuffled deck of cards. This way a single attempt would take a fraction of a second opposed to upwards of an hour to simulate.

VI. Results

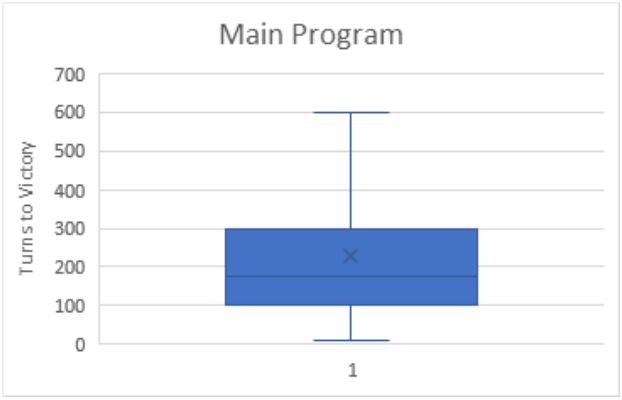
To collect data I ran the program 100,000 times for each implementation, both the program facing an opponent that plays a random card from their hand and an opponent that also plays with the same strategy. This was completed in under 20 minutes. The following data shows the results of each implementation and how long it took a player to win. The data from these trials can be found in Appendix A and Appendix B for the random opponent and the equally skilled opponent respectively.

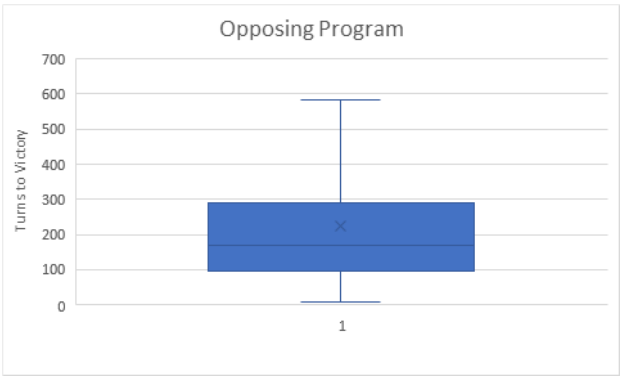
First will be the data against a random opponent. Opposed to the typical game of war and by extension a random game of Three Card War where the odds of winning a game are 50:50, the odds are now skewed in the favor of the player. Out of the 100,000 trials run, 55,668 of them the player won. Considering error into the equation we can assume that the odds of the player winning with this method would be 55:45. This is not a drastic change, but having the player's odds of winning in a randomized game is a victory (no pun intended). The average turns it took for the player to win was 326, and the randomized opponent took 335 turns on average. The data is visualized below in a box and whiskers plot that removes the outlying data, as sometimes it would go on for up to 3,000 turns.



When one player using this strategy plays another we get a completely different situation. Now the first player won 43,887

times, or about 44:56 odds of victory. This could be from any number of things including the random nature of shuffling a deck of cards, or maybe the opponent had some insight on what the player was doing. This could have been checked on if I had more time to complete this and it wasn't so difficult to identify what was causing this. Regardless, here is the interesting part. The average turns for the player to win was now 228, and the opponent at 222. This is considerably lower than the random opponent, probably because both are trying their hardest to win the game now, but the difference is still jarring. The data is visualized below also in a box and whiskers plot that removes the outlying data.I was able to identify the issue, and the opponent knew what was in the players hand at all times when picking their card. This one I only ran 10,000 times because it took quite a while. This surprisingly didn't effect the overall outcome of this experiment too much, and once it was fixed the odds became 49:51 with average turns to victory being 270 and 312 for player and opponent respectively.





I proceeded to run the code again (specifically the client versus a random opponent, and now only 10,000 attempts) with both 4 cards in hand and with 5 cards in hand. This was accomplished with only minor edits to the code, where 3s became 4s and 5s. This does technically make it a different game, but I was curious about the results. With 4 card war the odds of the player winning were 57:43 and 5 cards was 57:43. This proves that in the game when there is a little more strategy and less randomness it is easier to get the desired outcome, but there was little improvement over 3 card war, and both 4 and 5 cards were the same. The results from these are found in Appendix D and Appendix E respectively.

VII. Conclusion

This experience has shown that linear programming can be used to implement an AI. This AI changes the odds of winning the game from being even to being slightly weighted in favor of the player using it (although only slightly). Due to the random nature of this game already, there is no way to guarantee victory like when playing a game that has much lower luck and higher strategy such as chess. Getting fixed values from a random source however and choosing the outcome that is most likely to occur is something that is quite interesting and not something that I would have been able to think of on my own.

VIII. Code

The code can be found in Appendix C.

Appendix A:
https://drive.google.com/file/d/1rqgnzBWwP3IvZgQxZFv3YjdwQcPSFQLQ/view?usp=sharing

Appendix B:
https://drive.google.com/file/d/1rhSB5cfF52rKIU5OU3ek9_VMuHNmd4S0/view?usp=sharing

Appendix C:
https://drive.google.com/file/d/1jBe65fdG8iHfWhREKxN-UXRZd08DVtX1/view?usp=sharing

Appendix D:
https://drive.google.com/file/d/15aN00AUVSehhDgr34twKaNZZ60wqBb62/view?usp=sharing

Appendix E:
https://drive.google.com/file/d/1lyHwcnGN-9xCwE4m965PPEIYwWMVc2Uy/view?usp=sharing

References

[1] "AI chess algorithms." Cornell.edu, https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html